Document No.: P0669R0
Date: 2017-06-19
Revises: (None)
Reply to: Guy Davidson
<guy@creative-assembly.com>
Michael B. McLaughlin
<mikebmcl@gmail.com>
Audience: LEWG

Why We Should Standardize 2D Graphics for C++

Table of Contents

Introduction **3**

Summary of the History of 2D Computer Graphics **3**

A Tour of P0267 **5**

Colors 5

Linear algebra 5

Geometry 6

Paths 6

Brushes 6

Surfaces 7

Concerns and Replies **8**

“Game developers will never use it” 8

“2D graphics is already available in various other hardware accelerated libraries that everyone uses” 8

“No one will ever use it” 8

“It's just a toy and not worth spending time standardizing it” 9

“It will never be performant enough to be useful in the real world” 9

Future Plans and Goals **9**

A. Text rendering 9

B. Input 9

C. Goals of a TS 9

Acknowledgements **10**

I. Introduction

P0267 “A Proposal to Add 2D Graphics Rendering and Display to C++” (including its N number predecessors) has been reviewed and revised since 2014; first by SG13 and subsequently by LEWG. The proposal is likely to be forwarded from LEWG to LWG in Toronto.

This paper is meant to:

1. Provide an overview of 2D computer graphics and of P0267;
2. Explain why C++ should have a standardized 2D graphics API;
3. Lay out concerns that have been raised and provide the authors’ replies to them; and,
4. Set forth the roadmap and goals envisioned by the authors.

II. Summary of the History of 2D Computer Graphics

Computer graphics first appeared in the 1950s. The first displays were oscilloscopes which could be used to plot points and lines. *Spacewar!*, which was first released in 1962 for the DEC PDP-1, is widely recognized as the first computer game that was distributed to and played at multiple computer facilities.

As computers became less exotic and computer time became less expensive, games and puzzles became a typical way for students to learn programming. The first commercial video game, Pong, was a TTL device which rendered a small number of lines and points to a CRT device.

In 1974 the Evans and Sutherland frame buffer[1] debuted which allowed the display of 512x512 pixel images. Although enormously expensive at $15,000, prices were driven down over time. This allowed raster displays to become commonplace and to be incorporated into the home-brew computers of the late 1970s.

With the introduction of VRAM in the mid-1980s frame buffers became significantly cheaper and, as a result, available pixel and colour resolutions increased. Today’s consumer graphics cards can deliver resolutions exceeding 14 million pixels with 24 bits of colour information, suitable for the latest crop of 5K monitors.[2]

In the 1980s, software programs started to rely on 2D graphics for intuitive feedback of information using spatial contexts on screens. The widespread introduction of home computers to the market made 2D graphics a familiar experience. Many of today’s programmers had their first experience of programming on these machines, learning to code by writing graphics demonstrations and simple games.

During the 1990s graphics co-processors started to appear; these were add-in cards which provided additional computing power for performing vector calculations. They often contained their own separate RAM used for their own frame buffer to preserve locality in hardware. Over the past twenty years available resolutions have come to greatly exceed the typical resolution of a domestic television set.

Rendering images has spawned a large field of academic study, advanced by Bresenham's line drawing algorithm in 1965, and also by K. Vesprille's dissertation on the B-Spline approximation form in 1975. As colour depths and available grey scales have increased, font rendering and anti-aliasing have become rich areas of investigation. There are entire conferences devoted to rendering, for example SIGGRAPH.

Many 2D graphics libraries have been released. Some are very feature-rich, requiring many hundreds of hours of study and use to master. Some support hardware acceleration using the graphics co-processors that have become ubiquitous in desktop and laptop computers. Some support only one OS or only one type of GPU. From PostScript through to Skia, at the heart of all of them is an API for plotting points, drawing lines and curves, displaying bitmaps and rendering text. These operations have remained essentially unchanged for nearly 40 years, and it is these operations that we propose to standardise.

What these libraries lack is a standard C++ API. The C++ Standard Library and the C++ Technical Specifications all provide standard C++ APIs. Some features, such as atomics, rely on hardware support and quality of implementation to determine their performance. The same will be true for 2D graphics. Two of the goals of creating a 2D graphics TS are to get feedback from implementers regarding changes that could be made to allow them to provide better performance and to get feedback from users regarding improvements in usability and requests for additional features.

Today, computer graphics are pervasive in modern life, and have replaced console-style I/O for basic user interaction on many mainstream platforms. For example, writing a simple `cout << "Hello, world!";` statement doesn't do anything useful on many tablets and smartphones. The absence of a standard 2D graphics library leaves standard C++ output stuck in the 1970s. C++ has modernised, evolved, and continues to develop and improve as new features are added. C++ remains a vibrant, evolving, and highly relevant programming language. Clearly, the time is overdue for standardizing 2D graphics.

[1] A frame buffer is a dedicated piece of RAM containing a bitmap that drives a video display from a memory buffer containing a complete frame of data.

[2] HD monitors have a pixel resolution of 1920 x 1080. 4K monitors have twice the pixel resolution in each axis, 3840 x 2160, and therefore four times as many pixels. 5K monitors have a pixel resolution of 5120 x 2880, which is a little over seven times as many pixels as an HD monitor.

III. A Tour of P0267

There are effectively six components to P0267 as of P0267R5. The three main components are *paths*, *brushes*, and *surfaces*. The three supporting components are *colors*, *linear algebra*, and *geometry*.

A. Colors

Colors should be pretty self-explanatory, but the actual formal specification of color, via the scientific field of colorimetry, is rather complex. The proposal provides some background on this without aiming to be a treatise on the topic.

The library provides a class representing a premultiplied RGBA color. The class itself provides a large number of predefined, named colors drawn from CSS Color Module Level 3[3] along with a predefined color named `transparent_black`, which is useful when working with premultiplied alpha.

It does not currently include any types for HSL/HSV, CMYK, CIE L*a*b*, or other color formats or any conversion functions. If such features, or a subset of them, prove to be highly desirable, they can easily be added.

Recognizing that the data in most modern image files is encoded in the sRGB color space, this is also addressed to ensure that composing graphics data produces correct results.

B. Linear algebra

The library includes a `vector_2d` type and a `matrix_2d` type. These classes are in no way exhaustive nor do their uses match with their names (either in the mathematical usage or the physics usage). However they do provide the functionality expected by 2D graphics developers and the naming conventions (e.g. using `vector_2d` as both a (mathematical) vector and a coordinate point in a Cartesian plane) are neither surprising nor confusing in the context of 2D graphics.

The `vector_2d` class provides a two component vector that provides the basic arithmetic operations (+ - * /) as well as computations of dot products, magnitude, magnitude squared (avoids a sqrt calculation and lets it be constexpr), and the ability to produce a unit vector equivalent of the vector.

The `matrix_2d` class provides a 3x3 matrix that only exposes its first two columns. The third column is always the same regardless of the various affine transformations that are useful in 2D graphics. Similarly, when a `vector_2d` is transformed by the matrix, the non-observable third column is ignored.

The matrix supports scaling, rotation, translation, shearing (both x-axis and y-axis), and reflection. When multiplying or composing the matrix, the order to produce the most commonly desired results is scale * rotate * translate. Matrix multiplication is not commutative such that order matters. The desired composition order is the same, e.g. m.scale(...).rotate(...).translate(...). If any of those transformation isn't needed, it can be dropped provided that the order remains the same for the other transformation. That said, the "SRT" order is not the only useful order. Other orders and the inclusion of the other transformations will produce other results that are desirable in various applications.

C. Geometry

The paths component deals with composable geometry. As such, there are only two geometrical objects that are needed in the library, a `circle` type and a `rectangle` type. Providing these types rather than requiring the use of paths to create them provides usability improvements and potential performance improvements.

D. Paths

Paths provide vector graphics functionality. Specifically, the paths component provides lines, quadratic Bézier curves, cubic Bézier curves, and elliptical arcs. It supports the use and modification of a transformation matrix. It also supports specifying the points that are used in either absolute coordinates or relative coordinates as well as specifying whether a path should be open or closed.

An open path ends wherever it ends and the beginning and end of the path are rendered based on a line cap (none, round, or square). A closed path creates a line to the point where the path began and when the path is rendered, that join point is rendered as a line join (mitered, rounded, or beveled) in the same way as other components of the path are rendered at the points where they join.

The functionality provided in the path API is essentially the same as all other vector graphics APIs. It is not missing any of the fundamentals. Additionally, it enables further operations, such as creating B-splines by chaining cubic Bézier curves together.

E. Brushes

Brushes are used when painting an entire surface, to determine the color of paths when they are drawn (stroked) or filled, as a mask by using the brush's transparency (alpha) data to control what is painted and how much of the color is transferred to the destination surface.

There are four types of brushes: solid color, linear gradient, radial gradient, and surface.

The solid color brush is a uniform color. It is not required to be opaque; it allows translucency. The name simply means that the color is consistent.

The linear gradient brush uses a line along with color stops (a combination of a floating point value and a color) to create a source of color data that determines the color for a point in the brush by interpolating between the color stops along the line. The radial gradient brush is similar, but is defined by two circles which do not need to have the same radius. It also interpolated between the color stops along the area that stretches between the two circles.

The exact details of how it does this is beyond the scope of this paper but is well-defined in P0267 and can be viewed and experimented with using the reference implementation of P0267.

The final brush type is the surface brush. It is a brush that allows pixel data (such as a digital image) to be used and displayed.

F. Surfaces

Surfaces are the core of the API in many ways, though they would be almost useless without the other components. Some other terms used in graphics for what this API calls surfaces are textures and render targets. These objects are where graphics can be loaded from image files, saved to image files, drawn to using various operations in combination with brushes and, almost always, paths, and shown to users via graphical output devices (the graphics equivalent of the consoles and terminals that iostreams use to communicate with users).

There are two main surfaces: the `image_surface` type and the `display_surface` type. Both of these derive from the `surface` type, which provides core functionality common to the two main surface types. There is also a type called `mapped_surface` that allows the direct manipulation of individual pixel data contained within a surface. As such it has no data of its own.

There are four drawing operations for surfaces: painting, stroking, filling, and masking. Painting does not use a path. It simply transfers the contents of a brush to a surface in the method described by a compositing operator. Like all drawing operations, there are various state data objects that are passed along with the brush when a call to perform a painting operation is made. The compositing operator is one such piece of state data and it is part of the state data for all of the drawing operations. Each drawing operation has its own set of state data objects that it takes. These objects are contained within an `optional` object such that reasonable default state data can be used making fewer mandatory arguments for each operation.

The stroking operation draws paths using the supplied brush. One element of the state data that is passed to the surface's stroke function is the width of the path as it is stroked. Other state data controls various other aspects of how a stroking operation draws paths.

The filling operation fills a path using the supplied brush. Part of its state data is a fill mode that determines which of two algorithms is used to determine whether a point is inside the area that should be filled.

The masking operation is similar to the painting operation, except that it uses the alpha data of an additional brush, the mask brush, to determine whether or not a point within the brush should be drawn and if so how much its color should contribute to the result of drawing the brush to the surface.

(Note: When text rendering is added, it will be a fifth drawing operation. Adding it will not require modifying, adding overloads, or removing any of the functions in the current proposal.)

[3] Tantek, Çelik et al., *CSS Color Module Level 3 -- W3C Recommendation* Copyright © 2011 W3C (MIT, ERCIM, Keio)

IV. Concerns and Replies

A. “Game developers will never use it”

There are many different kinds of game developer. Some studios have dedicated graphics teams, pushing the hardware to the limit in search of the highest frame rate and greatest fidelity. Others produce puzzle games for phones at the rate of one a month which make little demand on the host system.

Additionally, there are many parts of C++ that game developers don’t use: not just language features like RTTI and exception handling but also library features which depend on them, or which are implemented in too general a fashion and can be optimised for a particular use.

The quality of implementation of the library will decide which game developers will use it. But all that aside, game development is not the only domain that can benefit from a 2D graphics API. There are many other types of application that make use of 2D graphics. Simply being able to draw a chart or a graph is utility enough. Game developers do not decide what is suitable for the library any more than CAD software developers.

B. “2D graphics is already available in various other hardware accelerated libraries that everyone uses”

This in fact supports the argument for standardisation: there are indeed various libraries, and it can take a significant amount of study to decide which library to use, to learn the library, and possibly to discover it was the wrong choice. Also, some libraries are not appropriate for some situations for reasons of licensing or legal indemnity. A graphics library as part of standard C++ would create a starting point for every application, and an opportunity to optimise where necessary.

Also there is no reason for a vendor not to implement the library making use of available hardware acceleration. There is no reason for this library to be any less performant than the host of libraries that already exist, nor for it to take advantage of particular knowledge of the target platform.

C. “No one will ever use it”

The authors have already received encouragement from educational institutions about the teaching potential for this library. As mentioned earlier, writing code which produces graphical output is a traditional route to learning the art of programming. By providing a standard 2D API this approach and its benefits become available to the student immediately and consistently between institutions. This argument is a more general version of point A and for the same reasons it does not stand.

D. “It's just a toy and not worth spending time standardizing it”

This argument again assumes insufficient performance to be useful. Even if it does only end up as a teaching device that in itself is immensely useful. However, as described in previous responses, there is no reason for such a library to be insufficiently performant for a significant set of applications. This library is either sufficient or a starting point for optimisation.

Once an API is standardised, GPU vendors are in a position to provide versions for a narrower set of targets than the compiler vendor’s implementation, as indeed are other experts in this domain.

E. “It will never be performant enough to be useful in the real world”

This argument is another version of point A. The “real world” has many different requirements of its 2D library, some more intensive than others. The authors believe that the set of applications that can be served by a standard library is extensive.

V. Future Plans and Goals

A. Text rendering

The ability to render and compose text is extremely important. It is also a somewhat difficult topic. The API that existed in P0267R2 was removed because it had significant shortcomings. Design work is already in progress to provide a proper text rendering API. The intention is to submit it as a proposed modification that would augment P0267. As of now, the re-addition of text rendering is expected to be a pure superset of the existing 2D functionality.

B. Input

Formal work has not yet begun on an input API. This is intentional. When the idea of proposing a standardized 2D graphics API was first considered, it was decided that, given the scope of the effort, output should be the sole focus of initial efforts, with work on an input API begun only after the output API stabilized. This assessment has not changed. Input is planned and will be proposed in due time. For now, the output API alone provides a lot of useful functionality.

C. Goals of a TS

If there is a 2D Graphics TS based on P0267 (with amendments), the following items are the major goals that the authors hope will be answered:

- What, if any, problems exist in the API that prevent or otherwise limit optimization opportunities?
- What, if anything, could be improved to aid in teachability?
- What changes could be made to improve usability?
- Is the API missing any features (excluding any known issues) that prevent you from using it? (Note: If you cannot replicate a feature using the API, please include that fact; features that cannot be replicated using the existing API will be considered first, but all reports will be considered.)

VI. Acknowledgements

The authors extend their thanks to Beman Dawes, Pete Becker, Tony Van Eerd, David Sankel, Jeffrey Yasskin, Jens Maurer, Peter Sommerlad, the Standard C++ Foundation, and many others for their efforts, feedback, and support.