Lean Programming Interfaces for Classic Graphics

Peter Occil

This version of the document is dated 2026-08-18.

Peter Occil

Notes on developing an application programming interface (API) for classic graphics with as few entry points as possible. As given in my **specification for such graphics**[1], *classic graphics* generally means 2-D or 3-D graphics achieved by video games before the year 2000 and the rise of programmable "shaders". In general, pre-2000 graphics involve a game screen of 640 × 480 or smaller, up to 12,800 3-D polygons at a time (fewer if the game screen is smaller), and tile- or sprite-based 2-D graphics.

1 Goal: An Open-Source Engine or API for Classic Graphics

It would be of interest to write a free and open-source graphics engine that implements this specification and renders graphics in software (with a minimum of source code and dependencies),[2] or to establish a lean API for this specification. The following are examples:

- *Quake* (1996), *Quake II* (1997), and *Quake III Arena* (1999) popularized the practice of using only a subset of the OpenGL 1.1 programming interface for a game's graphics rendering[3].
- The **API reference**[4] for the two-dimensional (2-D) game engine *Pyxel*. But, in addition to the efforts there, a minimal version of the Python language runtime and nonreliance on hardware acceleration (notably on an OpenGL implementation) would be worthwhile.
- The OpenGL ES 1.1 specification (either the Common profile or the Common-Lite profile with fixed-point but not floating-point arithmetic) is an approximation of pre-2000 3-D graphics.
- BGI, a 2-D graphics API (`graphics.h`) by what was then known as Borland.

The graphics engine is intended to run even on computers from around 2005 (and maybe even on older computers), and with low resources, and so to enable video games that run with acceptable performance on those computers.

[1] https://peteroupc.github.io/graphics.html

[2] In this document, "rendering in software" means that the rendering of graphics does not rely on a video card or graphics accelerator or a graphics API (such as GDI, OpenGL, or Direct3D) that is provided by the operating system, with the sole exception of sending a finished game screen image to the player's display (such as through GDI's `StretchDIBits` or copying to VGA's video memory). (Implementing a subset of OpenGL ES 1.1 or OpenGL 1.1 without relying on a video card or graphics accelerator is allowed.)The following are examples of a graphics library that follows the spirit, even if not the letter, of the classic-graphics specification: ***Tilengine***, ***kit***, ***DOS-like***, ***raylib*'s `rlsw` software renderer**. Michal Strehovský published an **interesting technique to create small game applications**, and so did **Jani Peltonen**. https://github.com/megamarc/Tilengine https://github.com/rxi/kit/ https://github.com/mattiasgustavsson/dos-like https://github.com/raysan5/raylib https://migeel.sk/blog/2024/01/02/building-a-self-contained-game-in-csharp-under-2-kilobytes/ https://www.codeslow.com/2019/12/tiny-windows-executable-in-rust.html

[3] See, for example, "Optimizing OpenGL drivers for Quake3" by John Carmack, referenced in .plan, April 30, 1999.

[4] https://github.com/kitao/pyxel?tab=readme-ov-file#api-reference

2 2-D Graphics

The simplest way to proceed is to give the application a *frame buffer*, a block of system memory consisting of a rectangular array of pixel samples. The nature of each pixel sample depends on the game's needs; for example, it may be an 8-bit index to a color palette, or a 16-bit color value. In this case, the application must render graphics in software.[5]

Note: Under the classic-graphics specification, the frame buffer has no more than 307,200 pixels.

A tile- and sprite-based API suggested by the classic-graphics specification is yet to be determined.

The following is a sketch of what could be included in a lean API for copying and stretching 2-D images as well as for geometric drawing.[6] (For such an API, antialiasing support is optional, and it's enough for the API to draw to images stored in system memory, and not also drawing to video memory or directly to the screen.)

- Getting and setting pixel values of an image.
- Filling with a solid color a rectangular area (whose edges have integer coordinates) of an image.
- Copying a rectangular area (whose edges have integer coordinates) of an image onto another image, with optional nearest-neighbor scaling. The copying can optionally exclude transparent pixels or pixels of a certain color.
- Filling 2-D paths with a solid color, with even/odd or nonzero winding order. 2-D paths are sequences of path segments (line segments, quadratic Bézier curves, cubic Bézier curves, and elliptical arcs).
- Drawing simple outlines of 2-D paths with a solid color.[7]

[5]In this document, "rendering in software" means that the rendering of graphics does not rely on a video card or graphics accelerator or a graphics API (such as GDI, OpenGL, or Direct3D) that is provided by the operating system, with the sole exception of sending a finished game screen image to the player's display (such as through GDI's `StretchDIBits` or copying to VGA's video memory). (Implementing a subset of OpenGL ES 1.1 or OpenGL 1.1 without relying on a video card or graphics accelerator is allowed.)The following are examples of a graphics library that follows the spirit, even if not the letter, of the classic-graphics specification: ***Tilengine***, ***kit***, ***DOS-like***, ***raylib*****'s `rlsw` software renderer**. Michal Strehovský published an **interesting technique to create small game applications**, and so did **Jani Peltonen**. https://github.com/megamarc/Tilengine https://github.com/rxi/kit/ https://github.com/mattiasgustavsson/dos-like https://github.com/raysan5/raylib https://migeel.sk/blog/2024/01/02/building-a-self-contained-game-in-csharp-under-2-kilobytes/ https://www.codeslow.com/2019/12/tiny-windows-executable-in-rust.html

[6]It is unclear whether to include any of the following in the lean 2-D graphics API; this will depend on how heavily pre-2000 or pre-1995 video games (spanning PCs, home computers, game consoles, and arcade machines) made use of them.a. 2-D affine transformations (which keep parallel lines parallel; examples are scalings, shears, and rotations).b. Translations, rotations and scalings, but no other 2-D affine transformations.c. Translations, rotations, scalings, and reflections, but no other 2-D affine transformations.d. So-called "**raster operations**" (bit-by-bit operations between two images), such as those found in the Windows API (for example, `BitBit`) and in the "Microsoft C runtime" (for example, `_putimage`, `_setwritemode`).e. **Alpha compositing** while drawing 2-D graphics.f. Drawing one image part over another (Porter and Duff's "over" operation), with optional translucency, but no other nontrivial alpha compositing. The images may have translucent (semitransparent) or transparent pixels.g. Same as (f), except the images can't have translucent pixels.h. Drawing dashed 2-D paths.i. Filling rectangular areas with a repeating image pattern without transparent or translucent pixels.j. Filling rectangular areas with a repeating image pattern. The image may have transparent pixels.k. Filling arbitrary 2-D paths with a repeating image pattern. The image may have transparent pixels.l. Drawing a solid color over an image, with optional translucency, but no other nontrivial alpha compositing. This implements "fading in" and "fading out".Note on points "a" to "c": Unlike the Windows NT GDI, the 16-bit GDI did not support arbitrary affine transformations; notably, it did not support rotations or shears (for references, see "Further Reading").Note on point "h": Dashed 2-D paths are also known as "styled lines". Simple outlines of 2-D paths are dashed ("styled") by leaving out pixels (examples include the `_getlinestyle` method of the "Microsoft C runtime", or the **sophisticated approach** in the Windows NT GDI [see "Further Reading"] that depends on the horizontal and vertical spacing between pixels). More general outlines of a 2-D path are dashed by leaving out parts of the path (an example is **found in Windows NT's model**).Note on points "i" to "k": Windows's `CreatePatternBrush` function for creating repeating image patterns supported only 8 × 8 or smaller images in Windows 95 and Windows 3.x. https://learn.microsoft.com/en-us/windows/win32/gdi/raster-operation-codes https://ciechanow.ski/alpha-compositing/ https://learn.microsoft.com/en-us/previous-versions/windows/drivers/display/styled-cosmetic-lines https://learn.microsoft.com/en-us/previous-versions/windows/drivers/display/geometric-wide-lines

[7]In this document, a *simple outline* of a 2-D path (including a line segment, curve, or arc) is drawn by approximating the path with a sequence of points at integer coordinates and coloring the pixels at those points. (For example, if the path is a line segment whose endpoints have integer coordinates, an algorithm by Bresenham draws a simple outline of it.) A "thicker" than simple outline can be drawn by approximating the 2-D path with line segments, then drawing filled circles

- Flood filling colored areas of an image.
- Optionally, "inverting" the colors or color indices of an image's rectangular area (whose edges have integer coordinates).
- Optionally, fading an image to or from black.[8]

A leaner API could provide for the following instead:

- Getting and setting pixel values of an image.
- Filling the following figures with a solid color.
  - Rectangles whose edges have integer coordinates, and ellipses that are tightly contained in them.
  - Polygons with integer coordinates and even/odd or nonzero winding order. The API can choose to support arbitrary polygons, convex polygons only, or monotone-vertical polygons only.[9]
- Drawing simple outlines of line segments with a solid color, supporting only integer coordinates.[10]
- Flood filling colored areas of an image.
- Optionally:
  - "Inverting" the colors or color indices of an image's rectangular area (whose edges have integer coordinates).
  - Drawing simple outlines of elliptical arcs with a solid color, supporting only integer coordinates.[11], [12]

The following is not included in either API.

- Color palette functions. Images may or may not have a color table, and the application is assumed to handle colors on an image-by-image basis. (There is the matter of sending finished images to the user's display, which may vary in the number of colors it supports, but the lean 2-D API need not be concerned about this.)
- Text rendering, since the needs of applications in supporting writing systems and languages vary, as do approaches to rendering text.[13]

---

around each segment's endpoints, then drawing filled rectangles that follow the path of each line segment. Thus, a lean graphics API need not support nonsimple outlines of paths. See also Ron Gery, "Primitive Cool", Microsoft Developer Network, Mar. 17, 1992; J.E. Bresenham, "Algorithm for computer control of a digital plotter." *IBM Systems Journal* 4.1 (1965): 25-30, **https://doi.org/10.1147/sj.41.0025** .

[8]This is not required for games that display no more than 256 colors at a time, since the game is assumed to have a game screen image with a color table, so that the fade effect can be implemented by altering the color table.

[9]A "monotone-vertical" polygon is one that changes direction along the y-axis exactly twice, whether or not the polygon is self-intersecting. Every convex polygon is monotone-vertical. See chapter 41 of *Michael Abrash's Graphics Programming Black Book Special Edition*, 1997.

[10]In this document, a *simple outline* of a 2-D path (including a line segment, curve, or arc) is drawn by approximating the path with a sequence of points at integer coordinates and coloring the pixels at those points. (For example, if the path is a line segment whose endpoints have integer coordinates, an algorithm by Bresenham draws a simple outline of it.) A "thicker" than simple outline can be drawn by approximating the 2-D path with line segments, then drawing filled circles around each segment's endpoints, then drawing filled rectangles that follow the path of each line segment. Thus, a lean graphics API need not support nonsimple outlines of paths. See also Ron Gery, "Primitive Cool", Microsoft Developer Network, Mar. 17, 1992; J.E. Bresenham, "Algorithm for computer control of a digital plotter." *IBM Systems Journal* 4.1 (1965): 25-30, **https://doi.org/10.1147/sj.41.0025** .

[11]In this document, a *simple outline* of a 2-D path (including a line segment, curve, or arc) is drawn by approximating the path with a sequence of points at integer coordinates and coloring the pixels at those points. (For example, if the path is a line segment whose endpoints have integer coordinates, an algorithm by Bresenham draws a simple outline of it.) A "thicker" than simple outline can be drawn by approximating the 2-D path with line segments, then drawing filled circles around each segment's endpoints, then drawing filled rectangles that follow the path of each line segment. Thus, a lean graphics API need not support nonsimple outlines of paths. See also Ron Gery, "Primitive Cool", Microsoft Developer Network, Mar. 17, 1992; J.E. Bresenham, "Algorithm for computer control of a digital plotter." *IBM Systems Journal* 4.1 (1965): 25-30, **https://doi.org/10.1147/sj.41.0025** .

[12]Elliptical arcs were not available in Windows CE version 2.0 and earlier, unlike Windows 95, Windows NT, and Windows 3.x. See Jon Christiansen, "Microsoft Windows CE Graphics Features".For drawing circular arcs, see, for example, Jack Bresenham. 1977. A linear algorithm for incremental digital display of circular arcs. Commun. ACM 20, 2 (Feb. 1977), 100–106, **https://doi.org/10.1145/359423.359432**; elliptical arcs: J. R. Van Aken, "An Efficient Ellipse-Drawing Algorithm," in IEEE Computer Graphics and Applications, vol. 4, no. 9, pp. 24-35, Sept. 1984, **https://doi.org/10.1109/MCG.1984.275994**.

[13]**Text rendering** is essentially the drawing of *glyphs* of text. *Glyphs* are graphics that represent elements of text (examples

3 3-D Graphics

Unlike with today's programmable "shaders", classic 3-D video-game graphics support only simple, yet admirable capabilities for real-time rendering of three-dimensional scenes, as well as a low scene complexity by today's standards (fewer than 20,000 triangles per frame).

This section gives suggestions for a lean API supporting 3-D graphics. Any implementation of it should render graphics in software[14] and optionally with hardware acceleration. As with 2-D, the 3-D API is allowed to support drawing to images stored in system memory without also supporting drawing to video memory or directly to the screen.

The **classic-graphics specification**[15] recognizes the following **3-D graphics capabilities**[16] as within its spirit:

- Z buffering (depth buffering).
- Bilinear filtering.
- Flat shading and Gouraud shading.
- Perspective correction.
- Per-vertex specular highlighting.
- Per-vertex depth-based fog.
- Line drawing.
- Two-texture blending.
- Edge antialiasing (smoothing).
- MIP mapping.
- Source and destination alpha blending.

These features were more or less supported by 3-D graphics accelerators offered to consumers from 1995 to 1999, and by widely distributed 3-D game consoles throughout the 1990s. The *PC 99 System Design Guide* sections 14.27 to 14.34 (except for the screen resolution, frame rate, and double buffering requirements) are also in scope.

Stencil buffers, bump mapping, environment mapping, and three- or four-texture blending are borderline capabilities.

3.1 Suggested C-Language Functions

```
DrawTrianglesOneTex(State3D *state, float* vertices, uint32_t numvertices,
   uint32_t * indices, uint32_t numindices, Texture *texture);
```

are letters, digits, the *i*'s dot, and the *f*-*f*-*l* combination), and a *font* is a collection of these glyphs along with rules for drawing and placing them. Glyphs are often but not always mapped one-to-one to letters or other writing symbols. Before 2000, there were three kinds of fonts for screen display: raster fonts (the glyphs are images); vector fonts (the glyphs are made of line segments and/or curves; Hershey, Modern, and Script are examples); and outline fonts (the glyphs are filled 2-D paths; TrueType fonts are examples). (Later developments saw (1) the addition of scalable colored graphics, especially *emoji*, to outline fonts and (2) **"subpixel" antialiasing** of glyphs, such as the ClearType technology announced in November 1998.) The conversion of text to glyphs and the positioning of such glyphs is often nontrivial. https://behdad.org/text2024

[14]In this document, "rendering in software" means that the rendering of graphics does not rely on a video card or graphics accelerator or a graphics API (such as GDI, OpenGL, or Direct3D) that is provided by the operating system, with the sole exception of sending a finished game screen image to the player's display (such as through GDI's `StretchDIBits` or copying to VGA's video memory). (Implementing a subset of OpenGL ES 1.1 or OpenGL 1.1 without relying on a video card or graphics accelerator is allowed.)The following are examples of a graphics library that follows the spirit, even if not the letter, of the classic-graphics specification: ***Tilengine***, ***kit***, ***DOS-like***, ***raylib*'s `rlsw` software renderer**. Michal Strehovský published an **interesting technique to create small game applications**, and so did **Jani Peltonen**. https://github.com/megamarc/Tilengine https://github.com/rxi/kit/ https://github.com/mattiasgustavsson/dos-like https://github.com/raysan5/raylib https://migeel.sk/blog/2024/01/02/building-a-self-contained-game-in-csharp-under-2-kilobytes/ https://www.codeslow.com/2019/12/tiny-windows-executable-in-rust.html

[15]https://peteroupc.github.io/graphics.html

[16]https://peteroupc.github.io/graphics.html#3_D_graphics

Draws a sequence of triangles. The `vertices` array is a rectangular array of numbers organized into "vertex blocks". The number of `float`s pointed to must equal the number of `float`s per vertex block times `numvertices`. The number of indices (`numindices`) must be a multiple of 3. `float` is a number in IEEE 754 binary32 format.[17]

> **Note:** As given in the classic graphics specification, the number of vertices per frame should be no more than 38,400 for a screen resolution of 640 × 480.

There are several possibilities for "vertex blocks":

- Each "vertex block" has eight `float` values: the x-, y-, and z-coordinates; the normal vector's X, Y, and Z components; and the texture coordinates (U and V).
- Each "vertex block" has five `float` values: the x-, y-, and z-coordinates; and the two texture coordinates. This is for games that render only textured triangles and calculate their own lighting.

`state` is the 3-D graphics state, to be determined. This state will include the game's frame buffer, and possibly additional parameters yet to be determined.

`texture` is information about the texture, to be determined. This information will include the texture's image data and a blending operation (add, modulate, etc.).

```
DrawTriangleFanOneTex(State3D *state, float* vertices, uint32_t numvertices, Texture *texture);
```

Draws a triangle fan. `vertices`, `texture`, and `numvertices` are as in `DrawTrianglesOneTex`. Moreover, `numvertices` must be 3 or greater.

```
DrawTriangleStripOneTex(State3D *state, float* vertices, uint32_t numvertices, Texture *texture);
```

Draws a triangle strip. `vertices`, `texture`, and `numvertices` are as in `DrawTrianglesOneTex`. Moreover, `numvertices` must be 3 or greater.

```
DrawTrianglesTwoTex(State3D *state, float* vertices, uint32_t numvertices,
   uint32_t * indices, uint32_t numindices, Texture *texture1, Texture *texture2);
DrawTriangleFanTwoTex(State3D *state, float* vertices, uint32_t numvertices,
   Texture *texture1, Texture *texture2);
DrawTriangleStripTwoTex(State3D *state, float* vertices, uint32_t numvertices,
   Texture *texture1, Texture *texture2);
```

Like the corresponding `...OneTex` versions, with the following exceptions. Each vertex block has ten `float` values: the x-, y-, and z-coordinates; the normal vector's X, Y, and Z components; the texture coordinates (U and V) for `texture1`; and the texture coordinates for `texture2`. These functions are suggested here because some games from the late 1990s rely on so-called *light-map* textures and two-texture blending rather than in-game lighting calculations.

```
DrawTriangles(State3D *state, float* vertices, uint32_t numvertices,
   uint32_t * indices, uint32_t numindices);
DrawTriangleFan(State3D *state, float* vertices, uint32_t numvertices);
DrawTriangleStrip(State3D *state, float* vertices, uint32_t numvertices);
```

Draws triangles without textures. Like the corresponding `...OneTex` versions, with the following exceptions. Each "vertex block" has six `float` values: the x-, y-, and z-coordinates; and a color's red, green, and blue values, each ranging from 0 through 1. This is for games that render only triangles without textures. These functions are suggested here because some games from the mid- to late 1990s often draw polygons without textures.

[17] This function and others in this section were inspired by APIs for drawing a block of 3-D primitives, such as the API introduced in DirectX 5's Direct3D (Immediate Mode). By contrast, the approach of *execute buffers*, found in DirectX versions 2 and 3, is not adopted in this section since many game developers reportedly found it hard to use. Likewise, Direct3D Retained Mode, a high-level API built on top of Direct3D Immediate Mode, was very rarely used in practice.

This is far from a complete list of useful 3-D drawing functions; there may be others, but the goal is to define a compact set of functions supporting only those 3-D capabilities actually used by video games in the 1990s and earlier.

4 Further Reading

- *Microsoft C/C++ Version 7.0 Run-Time Library Reference* (1991), section 2.6 ("Microsoft C runtime"). Describes the graphics routines the named library supports.
- The Windows graphics device interface (GDI):
  - The GDI supported in Windows 95, Windows 98, and Windows 3.x (the "16-bit GDI"), along with its limitations, is described in the following references:
    - "Microsoft Windows Software Development Kit: Programmer's Reference" (for Windows 3.1).
    - Charles Petzold, "Programming Windows 3.1: The Microsoft Guide to Writing Applications for Windows 3.1", Microsoft Press, 1992.
    - *Microsoft Win32 Programmer's Reference* (for Windows 95 and Windows 98).
    - "Microsoft Win32 API Overview", Microsoft Developer Network, February 1994.
    - "INFO: Developing Win32-Based GDI Apps for Windows 95 and Windows NT or Windows2000", Microsoft Knowledge Base article Q136989.
  - The Windows NT GDI is described in the following references. Its dashed-line feature did not materially change between the two references.
    - Chapter 3 of the *Graphics Driver Design Guide*, part of the Windows NT 4.0 Device Driver Kit.
    - The **archived documentation for the XDDM**[18], the driver model supported in Windows 2000, Windows XP, Windows Vista, and Windows 7.
  - Windows CE 2.0's version of the GDI and its 2-D graphics features are described in Jon Christiansen, "Microsoft Windows CE Graphics Features" (undated, about 1997). Windows CE was an operating system for embedded and handheld computers.

5 License

6 Notes

[18] https://learn.microsoft.com/en-us/previous-versions/windows/drivers/display/
[19] https://creativecommons.org/publicdomain/zero/1.0/