OpenGL

Open Graphics Library (OpenGL)[3][4] is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

Silicon Graphics Inc., (SGI) began developing OpenGL in 1991 and released it on June 30, 1992;[5][6] applications use it extensively in the fields of computer-aided design (CAD), virtual reality, scientific visualization, information visualization, flight simulation, and video games. Since 2006 OpenGL has been managed by the non-profit technology consortium Khronos Group.

Contents

1 Design

1.1 Development

2 Documentation

3 Associated libraries

3.1 Context and window toolkits

3.2 Extension loading libraries

3.3 Implementations

4 History

5 Version history

5.1 OpenGL 2.0

5.2 Longs Peak and OpenGL 3.0

5.3 OpenGL 3.0

5.4 OpenGL 3.1

5.5 OpenGL 3.2

5.6 OpenGL 4.0

5.7 OpenGL 4.1

5.8 OpenGL 4.2

5.9 OpenGL 4.3

5.10 OpenGL 4.4

5.11 OpenGL 4.5

5.12 OpenGL 4.6

6 Vulkan

7 See also

8 References

9 Further reading

10 External links

Design

An illustration of the graphics pipeline process

The OpenGL specification describes an abstract API for drawing 2D and 3D graphics. Although it is possible for the API to be implemented entirely in software, it is designed to be implemented mostly or entirely in hardware.

The API is defined as a set of functions which may be called by the client program, alongside a set of named integer constants (for example, the constant GL\_TEXTURE\_2D, which corresponds to the decimal number 3553). Although the function definitions are superficially similar to those of the programming language C, they are language-independent. As such, OpenGL has many language bindings, some of the most noteworthy being the JavaScript binding WebGL (API, based on OpenGL ES 2.0, for 3D rendering from within a web browser); the C bindings WGL, GLX and CGL; the C binding provided by iOS; and the Java and C bindings provided by Android.

In addition to being language-independent, OpenGL is also cross-platform. The specification says nothing on the subject of obtaining, and managing an OpenGL context, leaving this as a detail of the underlying windowing system. For the same reason, OpenGL is purely concerned with rendering, providing no APIs related to input, audio, or windowing.

Development

OpenGL is an evolving API. New versions of the OpenGL specifications are regularly released by the Khronos Group, each of which extends the API to support various new features. The details of each version are decided by consensus between the Group's members, including graphics card manufacturers, operating system designers, and general technology companies such as Mozilla and Google.[7]

In addition to the features required by the core API, graphics processing unit (GPU) vendors may provide additional functionality in the form of extensions. Extensions may introduce new functions and new constants, and may relax or remove restrictions on existing OpenGL functions. Vendors can use extensions to expose custom APIs without needing support from other vendors or the Khronos Group as a whole, which greatly increases the flexibility of OpenGL. All extensions are collected in, and defined by, the OpenGL Registry.[8]

Each extension is associated with a short identifier, based on the name of the company which developed it. For example, Nvidia's identifier is NV, which is part of the extension name GL\_NV\_half\_float, the constant GL\_HALF\_FLOAT\_NV, and the function glVertex2hNV().[9] If multiple vendors agree to implement the same functionality using the same API, a shared extension may be released, using the identifier EXT. In such cases, it could also happen that the Khronos Group's Architecture Review Board gives the extension their explicit approval, in which case the identifier ARB is used.[10]

The features introduced by each new version of OpenGL are typically formed from the combined features of several widely implemented extensions, especially extensions of type ARB or EXT.

Documentation

OpenGL's popularity is partially due to the quality of its official documentation.[citation needed] The OpenGL Architecture Review Board released a series of manuals along with the specification which have been updated to track changes in the API. These are commonly referred to by the colors of their covers:

The Red Book

OpenGL Programming Guide, 9th Edition. ISBN 978-0-134-49549-1

The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V

The Orange Book

OpenGL Shading Language, 3rd edition. ISBN 0-321-63763-1

A tutorial and reference book for GLSL.

Historic books (pre-OpenGL 2.0):

The Green Book

OpenGL Programming for the X Window System. ISBN 978-0-201-48359-8

A book about X11 interfacing and OpenGL Utility Toolkit (GLUT).

The Blue Book

OpenGL Reference manual, 4th edition. ISBN 0-321-17383-X

Essentially a hard-copy printout of the Unix manual (man) pages for OpenGL.

Includes a poster-sized fold-out diagram showing the structure of an idealised OpenGL implementation.

The Alpha Book (white cover)

OpenGL Programming for Windows 95 and Windows NT. ISBN 0-201-40709-4

A book about interfacing OpenGL with Microsoft Windows.

Associated libraries

The earliest versions of OpenGL were released with a companion library called the OpenGL Utility Library (GLU). It provided simple, useful features which were unlikely to be supported in contemporary hardware, such as tessellating, and generating mipmaps and primitive shapes. The GLU specification was last updated in 1998 and depends on OpenGL features which are now deprecated.

Context and window toolkits

Given that creating an OpenGL context is quite a complex process, and given that it varies between operating systems, automatic OpenGL context creation has become a common feature of several game-development and user-interface libraries, including SDL, Allegro, SFML, FLTK, and Qt. A few libraries have been designed solely to produce an OpenGL-capable window. The first such library was OpenGL Utility Toolkit (GLUT), later superseded by freeglut. GLFW is a newer alternative.[11]

These toolkits are designed to create and manage OpenGL windows, and manage input, but little beyond that.[12]

GLFW – A cross-platform windowing and keyboard-mouse-joystick handler; is more game-oriented

freeglut – A cross-platform windowing and keyboard-mouse handler; its API is a superset of the GLUT API, and it is more stable and up to date than GLUT

OpenGL Utility Toolkit (GLUT) – An old windowing handler, no longer maintained.

Several "multimedia libraries" can create OpenGL windows, in addition to input, sound and other tasks useful for game-like applications

Allegro 5 – A cross-platform multimedia library with a C API focused on game development

Simple DirectMedia Layer (SDL) – A cross-platform multimedia library with a C API

SFML – A cross-platform multimedia library with a C++ API and multiple other bindings to languages such as C#, Java, Haskell, and Go

Widget toolkits

FLTK – A small cross-platform C++ widget library

Qt – A cross-platform C++ widget toolkit. It provides many OpenGL helper objects, which even abstract away the difference between desktop GL and OpenGL ES

wxWidgets – A cross-platform C++ widget toolkit

Extension loading libraries

Given the high workload involved in identifying and loading OpenGL extensions, a few libraries have been designed which load all available extensions and functions automatically. Examples include GLEE, GLEW and glbinding. Extensions are also loaded automatically by most language bindings, such as JOGL and PyOpenGL.

Implementations

Mesa 3D is an open-source implementation of OpenGL. It can do pure software rendering, and it may also use hardware acceleration on BSD, Linux, and other platforms by taking advantage of the Direct Rendering Infrastructure. As of version 13.0, it implements version 4.5 of the OpenGL standard.

History

In the 1980s, developing software that could function with a wide range of graphics hardware was a real challenge. Software developers wrote custom interfaces and drivers for each piece of hardware. This was expensive and resulted in multiplication of effort.

By the early 1990s, Silicon Graphics (SGI) was a leader in 3D graphics for workstations. Their IRIS GL API[13] was considered state-of-the-art[citation needed] and became the de facto industry standard, overshadowing the open standards-based PHIGS. This was because IRIS GL was considered easier to use, and because it supported immediate mode rendering. By contrast, PHIGS was considered difficult to use and outdated in functionality.

SGI's competitors (including Sun Microsystems, Hewlett-Packard and IBM) were also able to bring to market 3D hardware supported by extensions made to the PHIGS standard, which pressured SGI to open source a version of IrisGL as a public standard called OpenGL.

However, SGI had many customers for whom the change from IrisGL to OpenGL would demand significant investment. Moreover, IrisGL had API functions that were irrelevant to 3D graphics. For example, it included a windowing, keyboard and mouse API, in part because it was developed before the X Window System and Sun's NeWS. And, IrisGL libraries were unsuitable for opening due to licensing and patent issues[further explanation needed]. These factors required SGI to continue to support the advanced and proprietary Iris Inventor and Iris Performer programming APIs while market support for OpenGL matured.

One of the restrictions of IrisGL was that it only provided access to features supported by the underlying hardware. If the graphics hardware did not support a feature natively, then the application could not use it. OpenGL overcame this problem by providing software implementations of features unsupported by hardware, allowing applications to use advanced graphics on relatively low-powered systems. OpenGL standardized access to hardware, pushed the development responsibility of hardware interface programs (device drivers) to hardware manufacturers, and delegated windowing functions to the underlying operating system. With so many different kinds of graphics hardware, getting them all to speak the same language in this way had a remarkable impact by giving software developers a higher level platform for 3D-software development.

In 1992,[14] SGI led the creation of the OpenGL Architecture Review Board (OpenGL ARB), the group of companies that would maintain and expand the OpenGL specification in the future.

In 1994, SGI played with the idea of releasing something called "OpenGL++" which included elements such as a scene-graph API (presumably based on their Performer technology). The specification was circulated among a few interested parties – but never turned into a product.[15]

Microsoft released Direct3D in 1995, which eventually became the main competitor of OpenGL. Over 50 game developers signed an open letter to Microsoft, released on June 12, 1997, calling on the company to actively support Open GL.[16] On December 17, 1997,[17] Microsoft and SGI initiated the Fahrenheit project, which was a joint effort with the goal of unifying the OpenGL and Direct3D interfaces (and adding a scene-graph API too). In 1998, Hewlett-Packard joined the project.[18] It initially showed some promise of bringing order to the world of interactive 3D computer graphics APIs, but on account of financial constraints at SGI, strategic reasons at Microsoft, and a general lack of industry support, it was abandoned in 1999.[19]

In July 2006, the OpenGL Architecture Review Board voted to transfer control of the OpenGL API standard to the Khronos Group.[20][21]

In June 2018, Apple deprecated OpenGL APIs on all of their platforms (iOS, macOS and tvOS), strongly encouraging developers to use their proprietary Metal API, which has been available for a few years.[22]

Version history

The first version of OpenGL, version 1.0, was released on June 30, 1992 by Mark Segal and Kurt Akeley. Since then, OpenGL has occasionally been extended by releasing a new version of the specification. Such releases define a baseline set of features which all conforming graphics cards must support, and against which new extensions can more easily be written. Each new version of OpenGL tends to incorporate several extensions which have widespread support among graphics-card vendors, although the details of those extensions may be changed.

OpenGL version history

Version Release Date Features

1.1 March 4, 1997 Texture objects

1.2 March 16, 1998 3D textures, BGRA and packed pixel formats,[23] introduction of the imaging subset useful to image-processing applications

1.2.1 October 14, 1998 A concept of ARB extensions

1.3 August 14, 2001 Multitexturing, multisampling, texture compression

1.4 July 24, 2002 Depth textures, GLSlang[24]

1.5 July 29, 2003 Vertex Buffer Object (VBO), Occlusion Queries[25]

2.0 September 7, 2004 GLSL 1.1, MRT, Non Power of Two textures, Point Sprites,[26] Two-sided stencil[25]

2.1 July 2, 2006 GLSL 1.2, Pixel Buffer Object (PBO), sRGB Textures[25]

3.0 August 11, 2008 GLSL 1.3, Texture Arrays, Conditional rendering, Frame Buffer Object (FBO)[27]

3.1 March 24, 2009 GLSL 1.4, Instancing, Texture Buffer Object, Uniform Buffer Object, Primitive restart[28]

3.2 August 3, 2009 GLSL 1.5, Geometry Shader, Multi-sampled textures[29]

3.3 March 11, 2010 GLSL 3.30, Backports as much function as possible from the OpenGL 4.0 specification

4.0 March 11, 2010 GLSL 4.00, Tessellation on GPU, shaders with 64-bit precision[30]

4.1 July 26, 2010 GLSL 4.10, Developer-friendly debug outputs, compatibility with OpenGL ES 2.0[31]

4.2 August 8, 2011[32] GLSL 4.20, Shaders with atomic counters, draw transform feedback instanced, shader packing, performance improvements

4.3 August 6, 2012[33] GLSL 4.30, Compute shaders leveraging GPU parallelism, shader storage buffer objects, high-quality ETC2/EAC texture compression, increased memory security, a multi-application robustness extension, compatibility with OpenGL ES 3.0[34]

4.4 July 22, 2013[35] GLSL 4.40, Buffer Placement Control, Efficient Asynchronous Queries, Shader Variable Layout, Efficient Multiple Object Binding, Streamlined Porting of Direct3D applications, Bindless Texture Extension, Sparse Texture Extension[35]

4.5 August 11, 2014[8][36] GLSL 4.50, Direct State Access (DSA), Flush Control, Robustness, OpenGL ES 3.1 API and shader compatibility, DX11 emulation features

4.6 July 31, 2017[37][38] GLSL 4.60, More efficient geometry processing and shader execution, more information, no error context, polygon offset clamp, SPIR-V, anisotropic filtering

OpenGL 2.0

Release date: September 7, 2004

OpenGL 2.0 was originally conceived by 3Dlabs to address concerns that OpenGL was stagnating and lacked a strong direction.[39] 3Dlabs proposed a number of major additions to the standard. Most of these were, at the time, rejected by the ARB or otherwise never came to fruition in the form that 3Dlabs proposed. However, their proposal for a C-style shading language was eventually completed, resulting in the current formulation of the OpenGL Shading Language (GLSL or GLslang). Like the assembly-like shading languages it was replacing, it allowed replacing the fixed-function vertex and fragment pipe with shaders, though this time written in a C-like high-level language.

The design of GLSL was notable for making relatively few concessions to the limits of the hardware then available. This hearkened back to the earlier tradition of OpenGL setting an ambitious, forward-looking target for 3D accelerators rather than merely tracking the state of currently available hardware. The final OpenGL 2.0 specification[40] includes support for GLSL.

Longs Peak and OpenGL 3.0

Before the release of OpenGL 3.0, the new revision had the codename Longs Peak. At the time of its original announcement, Longs Peak was presented as the first major API revision in OpenGL's lifetime. It consisted of an overhaul to the way that OpenGL works, calling for fundamental changes to the API.

The draft introduced a change to object management. The GL 2.1 object model was built upon the state-based design of OpenGL. That is, to modify an object or to use it, one needs to bind the object to the state system, then make modifications to the state or perform function calls that use the bound object.

Because of OpenGL's use of a state system, objects must be mutable. That is, the basic structure of an object can change at any time, even if the rendering pipeline is asynchronously using that object. A texture object can be redefined from 2D to 3D. This requires any OpenGL implementations to add a degree of complexity to internal object management.

Under the Longs Peak API, object creation would become atomic, using templates to define the properties of an object which would be created with one function call. The object could then be used immediately across multiple threads. Objects would also be immutable; however, they could have their contents changed and updated. For example, a texture could change its image, but its size and format could not be changed.

To support backwards compatibility, the old state based API would still be available, but no new functionality would be exposed via the old API in later versions of OpenGL. This would have allowed legacy code bases, such as the majority of CAD products, to continue to run while other software could be written against or ported to the new API.

Longs Peak was initially due to be finalized in September 2007 under the name OpenGL 3.0, but the Khronos Group announced on October 30 that it had run into several issues that it wished to address before releasing the specification.[41] As a result, the spec was delayed, and the Khronos Group went into a media blackout until the release of the final OpenGL 3.0 spec.

The final specification proved far less revolutionary than the Longs Peak proposal. Instead of removing all immediate mode and fixed functionality (non-shader mode), the spec included them as deprecated features. The proposed object model was not included, and no plans have been announced to include it in any future revisions. As a result, the API remained largely the same with a few existing extensions being promoted to core functionality.

Among some developer groups this decision caused something of an uproar,[42] with many developers professing that they would switch to DirectX in protest. Most complaints revolved around the lack of communication by Khronos to the development community and multiple features being discarded that were viewed favorably by many. Other frustrations included the requirement of DirectX 10 level hardware to use OpenGL 3.0 and the absence of geometry shaders and instanced rendering as core features.

Other sources reported that the community reaction was not quite as severe as originally presented,[43] with many vendors showing support for the update.[44][45]

OpenGL 3.0

Release date: August 11, 2008

OpenGL 3.0 introduced a deprecation mechanism to simplify future revisions of the API. Certain features, marked as deprecated, could be completely disabled by requesting a forward-compatible context from the windowing system. OpenGL 3.0 features could still be accessed alongside these deprecated features, however, by requesting a full context.

Deprecated features include:

All fixed-function vertex and fragment processing

Direct-mode rendering, using glBegin and glEnd

Display lists

Indexed-color rendering targets

OpenGL Shading Language versions 1.10 and 1.20

OpenGL 3.1

Release date: March 24, 2009

OpenGL 3.1 fully removed all of the features which were deprecated in version 3.0, with the exception of wide lines. From this version onwards, it's not possible to access new features using a full context, or to access deprecated features using a forward-compatible context. An exception to the former rule is made if the implementation supports the ARB\_compatibility extension, but this is not guaranteed.

OpenGL 3.2

Release date: August 3, 2009

OpenGL 3.2 further built on the deprecation mechanisms introduced by OpenGL 3.0, by dividing the specification into a core profile and compatibility profile. Compatibility contexts include the previously-removed fixed-function APIs, equivalent to the ARB\_compatibility extension released alongside OpenGL 3.1, while core contexts do not. OpenGL 3.2 also included an upgrade to GLSL version 1.50.

OpenGL 4.0

Release date: March 11, 2010

OpenGL 4.0 was released alongside version 3.3. It was designed for hardware able to support Direct3D 11.

As in OpenGL 3.0, this version of OpenGL contains a high number of fairly inconsequential extensions, designed to thoroughly expose the abilities of Direct3D 11-class hardware. Only the most influential extensions are listed below.

Hardware support: Nvidia GeForce 400 series and newer, AMD Radeon HD 5000 Series and newer (FP64 shaders implemented by emulation on some TeraScale GPUs), Intel HD Graphics in Intel Ivy Bridge processors and newer.[46]

OpenGL 4.1

Release date: July 26, 2010

Hardware support: Nvidia GeForce 400 series and newer, AMD Radeon HD 5000 Series and newer (FP64 shaders implemented by emulation on some TeraScale GPUs), Intel HD Graphics in Intel Ivy Bridge processors and newer.[46]

Minimum "maximum texture size" is 16,384 × 16,384 for GPU's implementing this specification.[47]

OpenGL 4.2

Release date: August 8, 2011[32]

Support for shaders with atomic counters and load-store-atomic read-modify-write operations to one level of a texture

Drawing multiple instances of data captured from GPU vertex processing (including tessellation), to enable complex objects to be efficiently repositioned and replicated

Support for modifying an arbitrary subset of a compressed texture, without having to re-download the whole texture to the GPU for significant performance improvements

Hardware support: Nvidia GeForce 400 series and newer, AMD Radeon HD 5000 Series and newer (FP64 shaders implemented by emulation on some TeraScale GPUs), and Intel HD Graphics in Intel Haswell processors and newer.[46] (Linux Mesa: Ivy Bridge and newer)

OpenGL 4.3

Release date: August 6, 2012[33]

Compute shaders leveraging GPU parallelism within the context of the graphics pipeline

Shader storage buffer objects, allowing shaders to read and write buffer objects like image load/store from 4.2, but through the language rather than function calls.

Image format parameter queries

ETC2/EAC texture compression as a standard feature

Full compatibility with OpenGL ES 3.0 APIs

Debug abilities to receive debugging messages during application development

Texture views to interpret textures in different ways without data replication

Increased memory security and multi-application robustness

Hardware support: Nvidia GeForce 400 series and newer, AMD Radeon HD 5000 Series and newer (FP64 shaders implemented by emulation on some TeraScale GPUs), Intel HD Graphics in Intel Haswell processors and newer.[46] (Linux Mesa: Ivy Bridge without stencil texturing, Haswell and newer)

OpenGL 4.4

Release date: July 22, 2013[35]

Enforced buffer object usage controls

Asynchronous queries into buffer objects

Expression of more layout controls of interface variables in shaders

Efficient binding of multiple objects simultaneously

Hardware support: Nvidia GeForce 400 series and newer,[48] AMD Radeon HD 5000 Series and newer (FP64 shaders implemented by emulation on some TeraScale GPUs), Intel HD Graphics in Intel Broadwell processors and newer (Linux Mesa: Haswell and newer),[49] Tegra K1.

OpenGL 4.5

Release date: August 11, 2014[8][36]

Direct State Access (DSA) – object accessors enable state to be queried and modified without binding objects to contexts, for increased application and middleware efficiency and flexibility.[50]

Flush Control – applications can control flushing of pending commands before context switching – enabling high-performance multithreaded applications;

Robustness – providing a secure platform for applications such as WebGL browsers, including preventing a GPU reset affecting any other running applications;

OpenGL ES 3.1 API and shader compatibility – to enable the easy development and execution of the latest OpenGL ES applications on desktop systems.

Hardware support: Nvidia GeForce 400 series and newer,[48] AMD Radeon HD 5000 Series and newer (FP64 shaders implemented by emulation on some TeraScale GPUs), Intel HD Graphics in Intel Broadwell processors and newer (Linux Mesa: Haswell and newer), Tegra K1, and Tegra X1.[51][52]

OpenGL 4.6

Release date: July 31, 2017[8][37][38]

more efficient, GPU-sided, geometry processing

more efficient shader execution (AZDO)

more information through statistics, overflow query and counters

higher performance through no error handling contexts

clamping of polygon offset function, solves a shadow rendering problem

SPIR-V shaders

Improved anisotropic filtering

Hardware support: Nvidia GeForce 400 series and newer,[48] Intel Haswell and newer, AMD Radeon HD 5000 Series and newer (FP64 shaders implemented by emulation on some TeraScale GPUs).

Driver support:

Mesa 19.2 on Linux supports OpenGL 4.6 for Intel Broadwell and newer.[53] Support for AMD Radeon GCN and Nvidia Kepler+ is in progress.

NVIDIA GeForce 397.31 Graphics Driver on Windows 7, 8, 10 x86-64 bit only, no 32-bit support. Released April 2018[54]

AMD Adrenalin 18.4.1 Graphics Driver on Windows 7 SP1, 10 version 1803 (April 2018 update) for AMD Radeon™ HD 7700+, HD 8500+ and newer. Released April 2018.[55][56]

Intel 26.20.100.6861 graphics driver on Windows 10. Released May 2019.[57][58]

Vulkan

Main article: Vulkan (API)

Vulkan, formerly named the "Next Generation OpenGL Initiative" (glNext),[59][60] is a grounds-up redesign effort to unify OpenGL and OpenGL ES into one common API that will not be backwards compatible with existing OpenGL versions.[61][62][63]

The initial version of Vulkan API was released on February 16, 2016.