Geometric Tools Engine Version 8.2 Installation Manual and Release Notes

David Eberly
https://www.geometrictools.com
https://github.com/davideberly/GeometricTools
Document Version 8.2.1
March 14, 2026

Contents

1 Introduction **2**

1.1 License 2

1.2 Copying the Distribution to Your Machine 3

1.3 Important Preprocessor Symbols Required by Projects 3

2 Development on Microsoft Windows **4**

2.1 Environment Variables 4

2.2 Compiling the Source Code 4

2.3 Automatic Generation of Project and Solution Files 5

2.4 Running the Samples 6

2.5 Microsoft Visual Studio Custom Visualizers 6

2.6 Falling Back to Direct3D 10 6

2.7 Falling Back to Direct3D 9 7

3 Development on Linux **7**

3.1 Environment Variables 8

3.2 Dependencies on Other Packages 8

3.3 Compiling the Source Code 8

3.3.1 Compiling and Running Using CMake from a Terminal Window 8

3.3.2 Compiling and Running Using Visual Studio Code 9

3.4 Support for OpenGL via Proprietary Drivers 10

4 Accessing the OpenGL Driver Information **10**

1 Introduction

You are about to install the Geometric Tools Engine 8.2. The source code consists of

- a header-only mathematics library,
- a graphics library for DirectX 11 or OpenGL 4.6 on Microsoft Windows,
- a graphics library for OpenGL 4.6 on Linux,
- a GPU-based mathematics library (not fully featured yet),
- an application library for DirectX 11 or OpenGL 4.6 on Microsoft Windows,
- an application library for OpenGL 4.6 on Linux,

where the application libraries are simple and used for the sample applications of the distribution.

1.1 License

The Geometric Tools Engine uses the Boost License, listed next.

> Boost Software License - Version 1.0 - August 17th, 2003
>
> Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the Software) to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:
>
> The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.2 Copying the Distribution to Your Machine

Unzip the distribution to a folder of your choice. The top-level folder of the distribution is GeometricTools and the subfolder for the distribution is named GTE. Some of the folder hierarchy is shown next.

```
GeometricTools
  GTE                      // Root folder for Geometric Tools Engine, set GTE_PATH to here.
    Applications           // Platform-independent interfaces for the samples.
        GLX                // Platform-dependent code for Linux GLX applications.
        MSW                // Platform-dependent code for Microsoft Windows applications.
    Graphics               // Platform-independent graphics files.
        DX11               // DX11-specific graphics files.
        GL46               // Platform-independent OpenGL-specific graphics files.
            GL             // The standard OpenGL header files supported by the engine.
            GLX            // Linux GLX graphics files.
            WGL            // Microsoft Windows WGL graphics files.
    Mathematics            // The bulk of the engine consists of mathematics support.
    MathematicsGPU         // GPU-based implementation for some mathematics algorithms.
    Samples                // Sample applications to illustrate parts of the code.
        Data               // A small number of data files for the samples.
        Distance           // Samples for distance algorithms.
        Geometrics         // Samples for computational geometry.
        Graphics           // Samples for graphics.
        Imagics            // Samples for 2D and 3D image processing.
        Intersection       // Samples for intersection algorithms.
        Mathematics        // Samples for mathematical algorithms and numerical methods.
        Physics            // Samples for 2D and 3D physics.
        SceneGraphs        // Samples for scene-graph-based 3D graphics.
    Tools                  // Several convenient tools.
        BitmapFontCreator  // Generate .h/.cpp file to represent a graphics font.
        GenerateProject    // Generate MSVS 2022/2026 projects for applications.
```

The Samples subfolders are many. Listing them here would make the displayed hierarchy difficult to read. The projects all use paths relative to GTE and they do not rely on the top-level directory being located at the root of a hard drive. An environment variable GTE_PATH is used to locate data files required by the application. How you set an environment variable depends on the operating system and/or shell you are using.

1.3 Important Preprocessor Symbols Required by Projects

On Microsoft Windows, the preprocessor symbol GTE_USE_MSWINDOWS must be added to the Visual Studio project configurations. If a project uses DirectX 11 (or later), the symbol GTE_USE_DIRECTX must be added to the Visual Studio projects. If a project uses OpenGL, the symbol GTE_USE_OPENGL must be added to the Visual Studio projects. Only one of GTE_USE_DIRECTX and GTE_USE_OPENGL is allowed. There is no support in GTE for having both graphics systems active simultaneously.

On Linux, the make system must have the defines GTE_USE_LINUX and GTE_USE_OPENGL. You should not define GTE_USE_DIRECTX in the Linux settings because GTE does not have an emulation layer that converts the DirectX code to OpenGL.

The following preprocessors apply whether you are using Microsoft Windows or Linux.

By default, GTE uses row-major order for storing matrices. Previously you had to add to the project settings the preprocessor symbol GTE_USE_ROW_MAJOR. The source code has been modified so that you no longer need to define GTE_USE_ROW_MAJOR. If you want column-major order instead, add to the project settings the preprocessor symbol GTE_USE_COL_MAJOR.

By default, GTE multiplies a matrix A and a vector V using $A*V$, which is the *vector-on-the-right convention*. With this convention, A is an $n \times m$ matrix, V is an m-tuple considered to be an $m \times 1$ (column) vector, and AV is an $n \times 1$ matrix but considered to be an n-tuple. Previously you had to add to the project settings the preprocessor symbol GTE_USE_MAT_VEC. The source code has been modified so that you no longer need to define GTE_USE_MAT_VEC. If you want the multiplication to represent $V * A$ instead, which is the vector-on-the-left convention, add to the project settings the preprocessor symbol GTE_USE_VEC_MAT. With this convention, V is an n-tuple considered to be a $1 \times n$ matrix and the product $V * A$ is a $1 \times m$ matrix considered to be an m-tuple.

The forthcoming Geometric Tools Library (GTL) eliminates the conditional compilation symbols for matrix storage order and for matrix-vector products. In GTL, matrices are stored in row-major order, but code that takes raw pointers to matrix data stored in 1-dimensional memory also has arguments for you to specify the storage order. GTL has adapter classes to wrap the raw pointers and access the data correctly depending on the storage order.

2 Development on Microsoft Windows

The code is maintained currently on an Intel-based computer with Microsoft Windows 11 Professional, Version 24H2 using Microsoft Visual Studio 2022. Previous versions of Microsoft Visual Studio are no longer supported because they are past their product life cycles.

LLVM clang-cl, Intel C++ Compiler 2024, and Intel C++ Compiler 2025 are supported. You can select the compiler in the Microsoft Visual Studio 2022 by launching the project properties dialog and modifying the Platform Toolset using the drop-down list. Use the tool GeometricTools/GTE/Tools/ChangePlatformToolset for modification of the platform toolset for all .vcxproj files in the GTE folder hierarchy.

2.1 Environment Variables

Create an environment variable named GTE_PATH that stores the absolute directory path to the folder GeometricTools/GTE. For example, if you unzipped the distribution to the root of the C drive, you would set GTE_PATH to C:/GeometricTools/GTE.

2.2 Compiling the Source Code

Microsoft Visual Studio 2022 is Version 17 (Platform Toolset v143). The solution, project and filter names have embedded in them. The solution, project and filter files are in the root folder GeometricTools/GTE and are named

- GTMathematics.v17.{sln,vcxproj,vcxproj.filters}

- GTGraphics.v17.{sln,vcxproj,vcxproj.filters}
- GTMathematicsGPU.v17.{sln,vcxproj,vcxproj.filters}
- GTGraphicsDX11.v17.{sln,vcxproj,vcxproj.filters}
- GTGraphicsGL46.v17.{sln,vcxproj,vcxproj.filters}
- GTApplicationsDX11.v17.{sln,vcxproj,vcxproj.filters}
- GTApplicationsGL46.v17.{sln,vcxproj,vcxproj.filters}
- GTBuildAllDX11.v17.sln
- GTBuildAllGL46.v17.sln
- GTBuildAll.v17.sln

The GTMathematics library is header-only, so no output is produced by building these projects. The GT-Graphics library contains the graphics-API-independent graphics classes and depends on GTMathematics. The GTMathematicsGPU library contains GPU-based implementations and depends on GTMathematics and GT-Graphics; it does not have much in it yet, but as CPU-based algorithms are ported to the GPU, the library will be populated with these implementations. The GTGraphicsDX11 library adds DirectX 11 support (for Microsoft Windows) and the GTGraphicsGL44 library adds OpenGL 4.6 support (for Microsoft Windows via WGL and for Linux via GLX). The GTApplicationsDX11 library provides common files for all samples plus DX11-specific code. The GTApplicationsGL46 library has the same common files but also had GL46-specific code. The build-all solutions allow you to build everything with one press of the build button. One solution is for DX11 builds, one solution is for GL46 builds, and the last solution builds everything. WARNING: If you use build-all, the disk storage requirements are large.

2.3 Automatic Generation of Project and Solution Files

Creating a new Microsoft Visual Studio project and manually setting its properties to match those of the current sample applications is tedious. A tool is provided to generate a skeleton project, solution and source files, namely, GeometricTools/GTE/Tools/GenerateProject. You must specify whether the project is for a console application (c), a 2D windowed application (w2) or a 3D windowed application (w3). You must also specify a nesting level relative to the GeometricTools/GTE folder. For example, suppose you want to create a new 3D windowed project in the folder, GeometricTools/GTE/Samples/Graphics/MySample for a sample application. Copy GenerateProject.exe to that folder, and in a command window opened in that folder, execute

```
GenerateProject w3 3 MySample
```

The application type is specified by the command line parameter w3, which leads to generation of skeleton source code files for a 3D windowed application. The command line parameter 3 indicates the nesting of the MySample folder relative to the GTE folder. The tool creates solution files, project files, and filter files for all three supported compilers. It also creates three source files: MySampleWindow3.h, MySampleWindow3.cpp and MySampleMain.cpp. You can open a solution, compile the project, and run the application. By default, a window is displayed with white client region. The application does nothing until you add your own code.

If you want the generated files to live in a folder outside the GTE hierarchy, you will need to modify the include path in the projects to $(GTE_PATH). You will also need to delete the GTE projects from the Required folder of the solution and re-add them so that the correct path occurs. After re-adding, you need to set the project references to include GTGraphics, GTMathematicsGPU (if you are using this library in your application), and the pair GTGraphicsDX11 and GTApplicationsDX11 for DX11-based applications or to the pair GTGraphicsGL46 and GTApplicationsGL46 for WGL-based applications. The references cause the build to link in these libraries.

Also, it is not necessary to copy GenerateProject.exe to the project folder. If the executable can be found via the PATH statement, just execute it in any folder of your choosing and then copy the generated files to your project folder.

2.4 Running the Samples

You can run the samples from within the Microsoft Visual Studio development environment. Samples that access data files use the GTE_PATH environment variable to locate those files; code is in place to assert when the environment variable is not set. If you run from Microsoft Windows, presumably double-clicking an executable via Windows Explorer, the environment variable is necessary for the application file to locate data files.

Many of the samples compile HLSL shaders at run time. This requires having D3Dcompiler_*.dll in your path, where * is the version number of the shader compiler. You might have to modify your PATH environment variable to include the path to the DLL. Typically, the DLL is in a Windows Kit folder.

2.5 Microsoft Visual Studio Custom Visualizers

A file has been added, GeometricTools/GTE/gtengine.natvis, that provides a native visualizer for the Vector and Matrix classes. Copy this to

C:/Users/YOURLOGIN/Documents/Visual Studio <VERSION>/Visualizers

where <VERSION> is one of 2019 or 2022. More visualizers will be added over time. Feel free to suggest GTE classes for which you want specialized visualization during debugging.

2.6 Falling Back to Direct3D 10

For Microsoft Windows machines, the default settings for GTE are to use Direct3D 11.0 or later for rendering and to compile the shaders for the built-in effects (such as Texture2Effect and VertexColorEffect) using Shader Model 5. These settings are also used when compiling shaders that are part of the sample application or those you write yourself. If you do not have graphics hardware recent enough to support the default configuration, it is possible to modify the start-up code in the sample applications to fall back to Direct3D 10.0 (Shader Model 4.0) or Direct3D 10.1 (Shader Model 4.1).

Open the graphics sample named VertexColoring. The main function has the block of code

```
Window::Parameters parameters(L"VertexColoringWindow", 0, 0, 512, 512);
auto window = TheWindowSystem.Create<VertexColoringWindow>(parameters);
TheWindowSystem.MessagePump(window, TheWindowSystem.DEFAULT_ACTION);
TheWindowSystem.Destroy(window);
```

All the 2D and 3D windowed applications have similar blocks of code. The Window::Parameters structure has a member named featureLevel that defaults to D3D_FEATURE_LEVEL_11_0. The general list of values from which you can choose is

```
enum D3D_FEATURE_LEVEL
{
    D3D_FEATURE_LEVEL_9_1   = 0x9100,   // 4_0_level_9_1
    D3D_FEATURE_LEVEL_9_2   = 0x9200,   // 4_0_level_9_1
    D3D_FEATURE_LEVEL_9_3   = 0x9300,   // 4_0_level_9_3
    D3D_FEATURE_LEVEL_10_0  = 0xa000,   // 4_0
    D3D_FEATURE_LEVEL_10_1  = 0xa100,   // 4_1
    D3D_FEATURE_LEVEL_11_0  = 0xb000,   // 5_0
    D3D_FEATURE_LEVEL_11_1  = 0xb100    // 5_1
}
D3D_FEATURE_LEVEL;
```

The enumeration is found in d3dcommon.h. If you have a graphics card that supports at most Direct3D 10.0, then modify the main code to

```
    Window::Parameters parameters(L"VertexColoringWindow", 0, 0, 512, 512);
#if defined(GTE_USE_DIRECTX)
    parameters.featureLevel = D3D_FEATURE_LEVEL_10_0;
#endif
    auto window = TheWindowSystem.Create<VertexColoringWindow>(parameters);
    TheWindowSystem.MessagePump(window, TheWindowSystem.DEFAULT_ACTION);
    TheWindowSystem.Destroy(window);
```

The class-static variable HLSLProgramFactory::defaultVersion is set in DX11Engine::CreateBestMatchingDevice() according to the feature level used to create the DX11 device.

For non-windowed applications, the DX11Engine constructors allow you to specify directly the feature level.

2.7 Falling Back to Direct3D 9

This is not really possible, because GTE uses constant buffers and other concepts without equivalent DX9 representations. The best you can do is specify one of the feature levels mentioned in the previous section for which LEVEL_9 is part of the name. Note that there is no shader profile with name 4_0_level_9_2. If you set the version string to "3_0", the D3DReflect call will fail with HRESULT 0x8876086C, which is not listed in winerror.h. This is the code for the obsolete D3DERR_INVALIDCALL. The HLSL assembly instructions for Shader Model 3 do not contain constant buffer register assignments (because they did not exist then).

3 Development on Linux

The GTE source code and sample applications have been tested on

- Ubuntu 24.04.1 LTS with CMake 3.28.1, gcc 13.2.0, and Visual Studio Code 1.85.1.
- Fedora 41 with CMake 3.28.2, gcc 14.2.1, and Visual Studio Code 1.92.1.
- openSUSE Leap 15.5 with CMake 3.20.4, gcc 7.5.0, and Visual Studio Code 1.85.1.

As mentioned previously, your graphics driver must be capable of OpenGL 4.6.

3.1 Environment Variables

Create an environment variable named GTE_PATH that stores the absolute directory path to the folder GeometricTools/GTE. For example, if you use a bash shell, you would define the environment variable in the file .bashrc by adding the line

```
GTE_PATH=/home/YOURLOGIN/GeometricTools/GTE ; export GTE_PATH
```

The actual path depends on YOURLOGIN and where you copied the GTE distribution. The .bashrc file is processed when you login; however, if you modify it after logging in, you may process it by executing

```
source .bashrc
```

from a terminal window whose active directory is your home folder. For other versions of Linux or other shells, consult your user's guide on how to create an environment variable.

3.2 Dependencies on Other Packages

GTE depends on development packages for X11, OpenGL, GLX, EGL and libpng. The latter package is used for a simple reader/writer of PNG files for the sample applications. Use the package manager for your Linux distribution to install the aforementioned dependencies.

3.3 Compiling the Source Code

3.3.1 Compiling and Running Using CMake from a Terminal Window

The libraries must be built first using the shell script

```
GeometricTools/GTE/CMakeLibraries.sh
```

The file attributes for the script need to be set before running the script the first time. Change directory to the aforementioned folder and run

```
chmod a+x CMakeLibraries.sh
```

so that the script is executable. There are 4 library configurations. The build type is Debug or Release. The library flavor is Static (archives with extension .a) or Shared (shared libraries with extension .so.*). The build commands are

```
./CMakeLibraries.sh Debug Static
./CMakeLibraries.sh Debug Shared
./CMakeLibraries.sh Release Static
./CMakeLibraries.sh Release Shared
```

The build type and flavor names are case sensitive. If you misspell a parameter or omit one, the shell script will terminate with a message to that effect.

You can also build the libraries and all samples using the shell script CMakeLibrariesAndSamples.sh. The instructions for CMakeLibaries.sh apply as well for the library-sample script.

If you want to build samples only in a subfolder, say in the Graphics subfolder, you can use a shell script in that subfolder,

```
GeometricTools/GTE/Samples/Graphics/CMakeSamples.sh
```

The file attributes for the script need to be set using chmod before running the script the first time. Change directory to the aforementioned folder and run

```
chmod a+x CMakeSamples.sh
```

so that the script is executable. The build commands are

```
./CMakeSamples.sh Debug Static
./CMakeSamples.sh Debug Shared
./CMakeSamples.sh Release Static
./CMakeSamples.sh Release Shared
```

If you want to build a specific sample application, say Graphics/AreaLights, you can use a shell script in that subfolder,

```
GeometricTools/GTE/Samples/Graphics/AreaLights/CMakeSample.sh
```

The file attributes for the script need to be set before running the script the first time. Change directory to the aforementioned folder and run

```
chmod a+x CMakeSample.sh
```

so that the script is executable. The build commands are

```
./CMakeSample.sh Debug Static
./CMakeSample.sh Debug Shared
./CMakeSample.sh Release Static
./CMakeSample.sh Release Shared
```

The executables are stored in the following folders

```
GeometricTools/GTE/Samples/Graphics/AreaLights/build/DebugStatic/AreaLights
GeometricTools/GTE/Samples/Graphics/AreaLights/build/DebugShared/AreaLights
GeometricTools/GTE/Samples/Graphics/AreaLights/build/ReleaseStatic/AreaLights
GeometricTools/GTE/Samples/Graphics/AreaLights/build/ReleaseShared/AreaLights
```

You can change directory to a folder containing the executable and run it from the terminal window,

```
./AreaLights
```

Depending on Linux distribution, it might also be possible to double-click on the executable from a File Manager.

3.3.2 Compiling and Running Using Visual Studio Code

The top-level workspace folder for building the libraries and samples is

```
GeometricTools/GTE/GTE.code-workspace
```

From the file manager, you can right-click and open this by selecting the Visual Studio Code application. Or you can launch Visual Studio Code and use its file explorer to navigate to the workspace file. I have provided cmake-variants.json files so that the 4 build configurations mentioned previously are available to you.

The libraries must be built before you can build sample applications individually. I have provided .code-workspace files for the sample applications. To build, run, and debug the AreaLights sample, change directory to

```
GeometricTools/GTE/Samples/Graphics/AreaLights
```

and open AreaLights.code-workspace using Visual Studio Code. You can build any of the 4 configurations mentioned in the shell script discussion.

3.4 Support for OpenGL via Proprietary Drivers

I use proprietary graphics drivers; specifically, my Linux boxes have NVIDIA graphics cards and I use NVIDIA drivers. The drivers must support OpenGL 4.6 or later.

GTE uses a minimum of GLX functions in order to create windows that allow OpenGL accelerated rendering. All functions are included in the GLX packages for Linux, so there is no need for GLX extensions.

4 Accessing the OpenGL Driver Information

This section is applicable both to Microsoft Windows and to Linux.

The GL46Engine code is designed to allow you to write to disk information about the OpenGL driver. Extending the example for VertexColoring described in the previous sections, modify the main code

```
    Window::Parameters parameters(L"VertexColoringWindow", 0, 0, 512, 512);
#if defined(GTE_USE_OPENGL)
    parameters.deviceCreationFlags = 1;
#endif
    auto window = TheWindowSystem.Create<VertexColoringWindow>(parameters);
    TheWindowSystem.MessagePump(window, TheWindowSystem.DEFAULT_ACTION);
    TheWindowSystem.Destroy(window);
```

For now the only device creation flags for OpenGL are the default 0 or 1, the latter causing the OpenGL driver information to be written to a file named OpenGLDriverInfo.txt. The first several lines of the file show the vendor, the renderer (graphics card model and related) and the OpenGL version supported by the driver. The remaining lines list supported OpenGL extensions.