Geometric Tools Engine Version 6.3 Installation Manual and Release Notes

David Eberly, Geometric Tools
Document Version 6.3.0
April 3, 2022

Contents

1 Introduction

You are about to install the Geometric Tools Engine 6.3. The source code consists of

- a header-only mathematics library,
- a graphics library for DirectX 11 or OpenGL 4.5 on Microsoft Windows,
- a graphics library for OpenGL 4.5 on Linux,
- a GPU-based mathematics library (not fully featured yet),
- an application library for DirectX 11 or OpenGL 4.5 on Microsoft Windows,
- an application library for OpenGL 4.5 on Linux,
- an application library that was written for the sample applications.

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

The Linux distribution will typically require you to install the graphics card manufacturer's proprietary driver in order to use the graphics engine, because Linux tends to ship with the Nouveau Open Source graphics drivers that are not yet running OpenGL 4.5 and, typically, the performance is substandard. Visit the Geometric Tools website for updates, bug fixes, known problems, new features and other materials.

1.1 License

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

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.
      GL45                      // 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.
      ChangePlatformToolset     // Change the compiler used by the MSVS 2019 IDE.
      FiniteDifferences         // Generate coefficients for derivative approximations.
      GenerateApproximations    // Generate minimax approximations to standard functions.
      GenerateOpenGLWrapper     // Create OpenGL 4.5 support from ARB header filers.
      GenerateProject           // Generate MSVS 2015/2017/2019 vcxproj, sln, h, cpp for applications.
      PrecisionCalculator       // A simple testbed for computing bits needed for rational arithmetic.
      RotationApproximation     // Generate minimax approximations for rotations and their derivatives.
```

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

If you have a Visual Studio project that is compiled for Microsoft Windows, you must set the following preprocessor symbols:

```
GTE_USE_MSWINDOWS
GTE_USE_ROW_MAJOR
GTE_USE_MAT_VEC
```

The first symbol in the list exposes application code that is specific to Microsoft Windows and the Windows 10 SDK (and even Windows 8.1 if you choose that as the target SDK). The second symbol indicates that the matrices used in your code are stored in row-major order. The third symbol indicates that the product of an $n \times m$ matrix A and a vector V is AV. Abstractly, this means V has size $m \times 1$, but the implementation of vectors in GTE treats V as an m-tuple, not a matrix with 1 column.

It is possible to force column-major order by instead defining GTE_USE_COL_MAJOR, but I do not recommend

this. It is also possible to force the product of an $n \times m$ matrix A and a vector V to be VA. Abstractly, this means V has size $1 \times n$, but the implementation of vectors in GTE treats V as an n-tuple, not a matrix with 1 row. 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.

On Microsoft Windows, you have a choice of using DirectX or OpenGL for graphics. You must also select which one to use by using a preprocessor symbol.

```
GTE_USE_DIRECTX  // define this if you want DirectX
GTE_USE_OPENGL   // define this if you want OpenGL
```

You must define exactly one of these, no more and no less.

For Linux builds, the CMakeLists.txt files have add_definition commmands to define matrix storage order and matrix-vector products. If you roll your own make files, you must define the symbols in those files. Also, you need preprocessor symbols

```
GTE_USE_LINUX
GTE_USE_OPENGL
```

You should not define GTE_USE_DIRECTX in the Linux settings because I do not have an emulation layer that converts the DirectX code to OpenGL.

2 Development on Microsoft Windows

The code is maintained currently on an Intel-based computer with Microsoft Windows 10 Professional, Version 2004 using Microsoft Visual Studio 2015, 2017 and 2019. Microsoft Visual Studio 2013 and previous versions are no longer supported because they are past their Microsoft-deemed product life cycles.

Intel C++ compilers with version 17, 18 and 19 are supported. These are shipped in Intel Parallel Studio XE 2017 (for compiler version 17), 2018 (for compiler version 18), 2019 (for compiler version 19.0) and 2020 (for compiler version 19.1). Version 19.0 does not correctly compile code for std::shared_ptr reference counting; you should use Version 19.1 or later. Once installed, you can select the Intel compiler within Microsoft Visual Studio by right-clicking on the solution folder. The pop-up menu has an item *Intel Compiler*. Select that item to see a submenu that allows you to select Visual C++, Intel C++ or gcc (if you installed the latter).

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 2015 is Version 14 (Platform Toolset v140) of the compiler, Microsoft Visual Studio 2017 is Version 15 (Platform Toolset v141) of the compiler and Microsoft Visual Studio 2019 is Version 16 (Platform Toolset v142) of the compiler. The solution, project and filter names have embedded in them v14, v15 or v16; that is, all three versions of the compiler are supported. The solution, project and filter files are in the root folder GeometricTools/GTE and are named

- GTMathematics.{v14,v15,v16}.{sln,vcxproj,vcxproj.filters}
- GTGraphics.{v14,v15,v16}.{sln,vcxproj,vcxproj.filters}
- GTMathematicsGPU.{v14,v15,v16}.{sln,vcxproj,vcxproj.filters}
- GTGraphicsDX11.{v14,v15,v16}.{sln,vcxproj,vcxproj.filters}
- GTGraphicsGL45.{v14,v15,v16}.{sln,vcxproj,vcxproj.filters}
- GTApplicationsDX11.{v14,v15,v16}.{sln,vcxproj,vcxproj.filters}
- GTApplicationsGL45.{v14,v15,v16}.{sln,vcxproj,vcxproj.filters}
- GTBuildAllDX11.{v14,v15,v16}.sln
- GTBuildAllGL45.{v14,v15,v16}.sln
- GTBuildAll.{v14,v15,v16}.sln

The GTMathematics library is header-only, so no output is produced by building these projects. The GTGraphics library contains the graphics-API-independent graphics classes and depends on GTMathematics. The GTMathematicsGPU library contains GPU-based implementations and depends on GTMathematics and GTGraphics; 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 GTGraphicsGL45 library adds OpenGL 4.5 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 GTApplicationsGL45 library has the same common files but also had GL45-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 GL45 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 w3, which leads to generation of skeleton source code files for a 3D windowed application. The number 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 (although it 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. This is necessary because the Microsoft Visual Studio reference system is used to link in the GTE 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 still necessary.

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. With latest Windows, the DLL should be in a Windows Kit bin 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 2015, 2017 or 2019. 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 20.04.1 LTS with CMake 3.15.2 and gcc 9.3.0 and on Fedora 33 with CMake 3.18.3 and gcc 10.2.1. As mentioned previously, your graphics driver must be capable of OpenGL 4.5. You must have CMake installed and Visual Studio Code installed. I installed my copies using snap.

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

Versions of GTE prior to 5.1 used Unix makefiles for building the libraries and sample applications. These are no longer supported, replaced by the ability to use CMake from the command line or Visual Studio 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.

Once the libraries are build, the samples can be built using a shell script

```
GeometricTools/GTE/Samples/CMakeSamples.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 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 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 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
```

3.3.2 Compiling and Running Using Visual Studio Code

The top-level workspace folder for building the libraries is

```
GeometricTools/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.

I have provided .code-workspace files only for the sample applications. There is no parent workspace that encapsulates all the samples. 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. For some reason, the launch control in the status bar or side bar does not appear for me to select the build type to execute. I do so by selecting the debugger in the activity bar, at which time a drop-down list is available to select the build configuration to run.

3.4 Support for OpenGL via Proprietary Drivers

Many of the Linux distributions ship with Nouveau as the default graphics driver. If the driver does not support OpenGL 4.5 or later, the sample applications will terminate with a message *OpenGL 4.5 is required*. To execute the samples, You should install the proprietary drivers for your graphics hardware. How you install these drivers depends on the Linux distribution.

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 GL45Engine 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.