

# Geometric Tools Engine Version 3.24 Installation Manual and Release Notes

David Eberly, [Geometric Tools](#)  
Document Version 3.24.0  
April 12, 2019

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	License . . . . .	4
1.2	Copying the Distribution to Your Machine . . . . .	5
<b>2</b>	<b>Development on Microsoft Windows</b>	<b>6</b>
2.1	Environment Variables . . . . .	6
2.2	Compiling the Source Code . . . . .	6
2.3	Support for OpenGL . . . . .	7
2.4	Automatic Generation of Project and Solution Files . . . . .	7
2.5	Running the Samples . . . . .	7
2.6	Microsoft Visual Studio Custom Visualizers . . . . .	8
2.7	Falling Back to Direct3D 10 . . . . .	8
2.8	Falling Back to Direct3D 9 . . . . .	9
2.9	No Support Yet for Dynamic Libraries for GTEngine . . . . .	9
2.10	Managed Wrapper to Access GTEngine Code from C# . . . . .	9
2.11	Two-Phase Name Lookups in MSVS 2017/2019 and its Consequences . . . . .	9
<b>3</b>	<b>Development on Linux</b>	<b>10</b>
3.1	Environment Variables . . . . .	10
3.2	Dependencies on Other Packages . . . . .	11
3.3	Compiling the Source Code . . . . .	11
3.4	Support for OpenGL via Proprietary Drivers . . . . .	11
3.5	Running the Samples . . . . .	11

<b>4 Support for Macintosh OS X is Discontinued</b>	<b>12</b>
<b>5 Accessing the OpenGL Driver Information</b>	<b>12</b>
<b>6 Automatic Generation of a Wrapper for OpenGL</b>	<b>12</b>

# 1 Introduction

You are about to install Geometric Tools Engine 3.24. Version 1.0 source code was the companion to the book [GPGPU Programming for Games and Science](#) and was developed on Microsoft Windows 8.1 using Microsoft Visual Studio 2013, C++ 11 and Direct3D 11.1. Version 3.x is close enough to what is in the book that you should have no problem navigating the code as you read the book. The source code has been reorganized using subfolders of `Source` and `Include` and the header includes are now slightly different.

Version 3.x has a graphics engine that requires minimally OpenGL 4.3 (or later) and GLSL 4.3 (or later) in order to support compute shaders and GLSL introspection, and it runs on both Microsoft Windows (WGL) and Linux (GLX). **Please observe that the engine and sample applications require OpenGL 4.3 (or later) and GLSL 4.3 (or later). If your graphics driver does not support this, the applications will gracefully terminate with a message to the console window: *OpenGL 4.3 is required*. In particular, the Nouveau Open Source graphics drivers that ship with the various flavors of Linux might not support OpenGL 4.3 (or later) or GLSL 4.3 (or later). You must install the graphics card manufacturer's proprietary driver in order to use the graphics engine.**

A Direct3D 12 engine is in development, but given that the programming model is significantly different from Direct3D 11, the graphics subsystem of GTEngine needs to be redesigned. The plan is to factor GTEngine into smaller projects, each graphics API having its own project and corresponding application support. The mathematics library will also be factored into a stand-alone project to allow developers to use the code without having to depend on all other parts of the library.

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.

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

You may 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 GTEngine. Some of the folder hierarchy is shown next. The Include and Source folders contain all the code for the engine.

```
GeometricTools
GTEngine          // Root folder for Geometric Tools Engine, set GTE_PATH to here.
  Include         // Location for *.h files.
    Applications  // Platform-independent classes.
      GLX         // Support for Linux GLX applications.
      MSW        // Support for Microsoft Windows applications.
      DX11       // Support for Direct3D 11 applications.
      WGL        // Support for WGL applications.
    Graphics     // Platform-independent graphics files.
      DX11       // DX11-specific graphics files.
      GL4        // Platform-independent OpenGL-specific graphics files.
      GLX        // Linux GLX graphics files.
      WGL        // WGL graphics files.
    Imagsics     // Image processing files.
    LowLevel    // Several low-level utility files.
      MSW        // Microsoft Windows-specific files.
    Mathematics // The bulk of the engine consists of mathematics support.
      MSW        // Microsoft Windows-specific files.
    Physics     // Some physics support, not all WM5 physics code has been ported.
  Source        // Location for *.cpp files.
    Applications // Platform-independent classes.
      GLX         // Support for Linux GLX applications.
      MSW        // Support for Microsoft Windows applications.
      DX11       // Support for Direct3D 11 applications.
      WGL        // Support for WGL applications.
    Graphics     // Platform-independent graphics files.
      DX11       // DX11-specific graphics files.
      GL4        // Platform-independent OpenGL-specific graphics files.
      GLX        // Linux GLX graphics files.
      WGL        // WGL graphics files.
    Imagsics     // Image processing files.
    LowLevel    // Several low-level utility files.
      MSW        // Microsoft Windows-specific files.
    Mathematics // The bulk of the engine consists of mathematics support.
      MSW        // Microsoft Windows-specific files.
    Physics     // Some physics support, not all WM5 physics code has been ported.
  Samples       // Sample applications, many discussed in the GPGPU book.
    Data        // A small number of data files for the samples.
    Basics     // Basic tutorials for several HLSL concepts.
    CSharpCppManaged // Sample application with libraries to call C++ code from C#.
    Geometrics // Samples for computational geometry.
    Graphics    // Samples for graphics and video streams (parallel copy).
    Imagsics   // Samples for 2D and 3D image processing.
    Mathematics // Samples for mathematical algorithms and numerical methods.
    Physics    // Samples for 2D and 3D physics.
    DX11       // Samples specifically for Direct3D 11.
  Shaders      // HLSL/GLSL shader files (embedded versions are in the engine source).
  Tools        // Several convenient tools.
    BitmapFontCreator // Generate .h/.cpp file to represent a graphics font.
    GenerateApproximations // Used to generate the minimax approximations for common functions.
    GenerateOpenGLWrapper // Source-code generator for gl* wrappers driven by glcorearb.h.
    GenerateProject // Generate MSVS 2013/2015 vcxproj, sln, h and cpp for applications.
    PrecisionCalculator // Determine bits of precision needed exact-arithmetic algorithms.
```

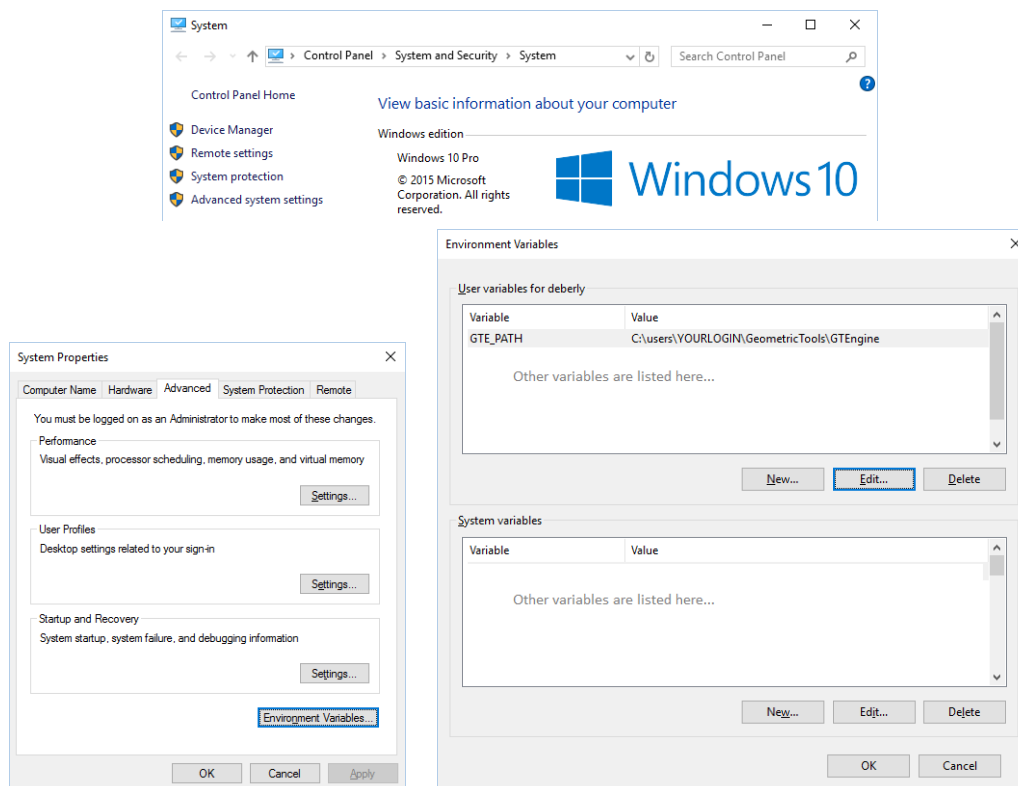
The Samples subfolders are many. Listing them here would make the displayed hierarchy difficult to read. The projects all use paths relative to GTEngine and they do not rely on the top-level directory being located at the root of a hard drive. An environment variable GTE\_PATH is used to locate data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

## 2 Development on Microsoft Windows

The code is maintained currently on an Intel-based computer with Microsoft Windows 10, Version 1809 (OS Build 17763.437). If you develop on Microsoft Windows 7 or 8.x and encounter problems with the code, contact us via the email address listed at the [Geometric Tools](#) website.

### 2.1 Environment Variables

Create an environment variable named `GTE_PATH` that stores the absolute directory path to the folder `GeometricTools/GTEngine`. For example, if you unzipped the distribution to the root of the C drive, you would set `GTE_PATH` to `C:/GeometricTools/GTEngine`. You can set environment variables via `System | Advanced system settings`, which launches the System Properties dialog with a button for `Environment Variables`.



### 2.2 Compiling the Source Code

Microsoft Visual Studio 2013 is Version 12 (Platform Toolset v120) of the compiler, 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 project and solution names have embedded in them v12, v14, v15 or v16; that

is, all four versions of the compiler are supported. The engine solutions are in the root folder `GeometricTools/GTEngine` and are named `GTEngine.v12.sln`, `GTEngine.v14.sln`, `GTEngine.v15.sln` and `GTEngine.v16.sln`. Each sample application or tool has its own solution with all dependencies listed, so it is possible to open a sample application and compile and run it without explicitly building the engine solution first. The root folder also contains the solutions `GTBuildAll.v12.sln`, `GTBuildAll.v14.sln`, `GTBuildAll.v15.sln` and `GTBuildAll.v16.sln` if you want to build the engine, samples and tools at the same time rather than building the projects separately.

## 2.3 Support for OpenGL

All the Microsoft Visual Studio projects have configurations `Debug` and `Release` that include compiling the `Direct3D 11` source code. The configurations `DebugGL4` and `ReleaseGL4` include compiling the `OpenGL` source code. Although it is possible to have an application that creates both a `Direct3D 11` engine and an `OpenGL` engine, the current project design does not allow this. If you have such a need, you must create your own `GTEngine` project that enables both sets of graphics code. For example, you might have an application that uses `OpenGL` for rendering but uses `Direct3D 11` for compute shaders (for `GPGPU`).

## 2.4 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/GTEngine/Tools/GenerateProject`. As an example of how to use the tool, suppose you want to create a new project in the folder, `GeometricTools/GTEngine/Samples/Graphics/MySample` for a sample application. Copy `GenerateProject.exe` to that folder, and in a command window opened in that folder, execute

```
GenerateProject 3 MySample
```

The number 3 indicates the nesting of the `MySample` folder relative to the `GTEngine` folder. The tool creates the files `MySample.v12.sln`, `MySample.v12.vcxproj`, `MySample.v12.vcxproj.filters`, `MySample.v14.sln`, `MySample.v14.vcxproj`, `MySample.v14.vcxproj.filters`, `MySample.v15.sln`, `MySample.v15.vcxproj`, `MySample.v15.vcxproj.filters`, `MySample.v16.sln`, `MySample.v16.vcxproj`, `MySample.v16.vcxproj.filters`, `MySampleWindow.h` and `MySampleWindow.cpp`. You can open the solution, build and run to see a blank window of size  $512 \times 512$ .

If you want the generated files to live in a folder outside the `GTEngine` hierarchy, you will need to modify the include path in the projects to `$(GTE_PATH)/Include`. You will also need to delete the `GTEngine` project from the `Required` folder of the solution and re-add it so that the correct path occurs. This is necessary because the Microsoft Visual Studio reference system is used to link in the `GTEngine` library.

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.

## 2.5 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.

## 2.6 Microsoft Visual Studio Custom Visualizers

A new file has been added, `GeometricTools/GTEngine/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 2013, 2015, 2017 or 2019. More visualizers will be added over time. Feel free to suggest GTEngine classes for which you want specialized visualization during debugging.

## 2.7 Falling Back to Direct3D 10

For Microsoft Windows machines, the default settings for GTEngine 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<VertexColoringWindow>(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_DEV_OPENGL)
parameters.featureLevel = D3D_FEATURE_LEVEL_10_0;
#endif
auto window = TheWindowSystem.Create<VertexColoringWindow>(parameters);
TheWindowSystem.MessagePump(window, TheWindowSystem.DEFAULT_ACTION);
TheWindowSystem.Destroy<VertexColoringWindow>(window);

```

Comments were added after the enumerates to indicate what to assign to `HLSLProgramFactory::defaultVersion`.

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

## 2.8 Falling Back to Direct3D 9

This is not really possible, because `GTengine` 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).

## 2.9 No Support Yet for Dynamic Libraries for GTengine

Currently, the engine solution generates static libraries. The hooks are in place for dynamic libraries, but the build configurations have not yet been added to the projects.

## 2.10 Managed Wrapper to Access GTengine Code from C#

There are projects

```

GeometricTools/GTengine/Samples/CSharpCppManaged_MSVS2013
GeometricTools/GTengine/Samples/CSharpCppManaged_MSVS2015
GeometricTools/GTengine/Samples/CSharpCppManaged_MSVS2017
GeometricTools/GTengine/Samples/CSharpCppManaged_MSVS2019

```

that shows how to wrap `GTengine` code using a managed wrapper (C++/CLI) that can be called from C# code.

## 2.11 Two-Phase Name Lookups in MSVS 2017/2019 and its Consequences

Two-phase name lookups is in reference to how names are resolved in a template declaration. See the blog [Two-phase name lookup support comes to MSVC](#) for a detailed discussion of what this means. As the authors indicate, the default is *not* to use the two-phase lookups. If you do want this, say, to have the compiler behave like the g++ compilers on Linux platforms, you must add a command-line option `/permissive-` as described in the blog. This command-line option has been added to `GTengine.v15.vcxproj`.

It is also important to read the paragraph before *What's coming next*. The Windows Kits for Microsoft Windows 8.1 has a header file that is indirectly included when using DirectX 11 graphics. Compilation fails when `/permissive-` is specified. If you have selected the target platform for Microsoft Windows 8.1, you either have to remove the `/permissive-` option or add the option `/Zc:twoPhase`. Alternatively, you can skip support for Microsoft Windows 8.1 and change the target platform to Microsoft Windows 10. Currently, the GTEngine projects (engine and applications) set the target to 10.0 (latest installed version). You can always choose a previous version of the Windows SDK from a drop-down list.

One of the consequences of two-phase name lookups is that you might have to reorder headers. The lookup algorithm is based (by definition) on the order in which template functions occur during parsing and compiling. For example, the torus-fitting code in `GteAppTorus3.h` had a set of headers included in the order

```
#include <Mathematics/GteRootsPolynomial.h>
#include <Mathematics/GteBSRational.h>
#include <Mathematics/GteUIntegerAP32.h>
```

The compilation failed with a complaint that a call to `std::sqrt` in `GteRootsPolynomial.h` was ambiguous. The compiler suggested choices involving inputs of `float`, `double` or `long double`. Those are the only options available at the time because `GteBSRational.h` has not yet been included. The reordering of the headers to

```
#include <Mathematics/GteUIntegerAP32.h>
#include <Mathematics/GteBSRational.h>
#include <Mathematics/GteRootsPolynomial.h>
```

fixed the problem because now `GteUIntegerAP32.h` is processed first, so that type is known to the compiler. The file `GteBSRational.h` is processed next, and it has the definitions for `std::sqrt` (and other math functions) whose inputs are of type `BSRational`. Finally, the file `GteRootsPolynomial.h` is processed. When the torus-fitting code using the polynomial root finder is encountered, the compiler now has access to the math functions defined for `BSRational`.

The same reordering was required using `g++` on Linux when two-phase name lookups are enabled.

## 3 Development on Linux

The GTEngine source code and sample applications have been tested on [Ubuntu 16.04](#) and [Ubuntu 18.04](#). As mentioned previously, your graphics driver must be capable of OpenGL 4.3 (or later) and GLSL 4.3 (or later).

### 3.1 Environment Variables

Create an environment variable named `GTE_PATH` that stores the absolute directory path to the folder `GeometricTools/GTEngine`. 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/GTEngine ; export GTE_PATH
```

The actual path depends on `YOURLOGIN` and where you copied the GTEngine 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

Each of the five supported flavors of Linux was installed from Live distributions. GTEngine depends on development packages for X11, OpenGL, GLX 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

The makefile to build the GTEngine library is `GeometricTools/GTEngine/makeengine.gte` where both static and shared library builds are supported. From a terminal window execute

```
make CFG=configuration -f makeengine.gte
```

where `configuration` is `Debug` or `Release` for static libraries and `DebugDynamic` or `ReleaseDynamic` for shared libraries.

You can build all samples by changing directory to `GeometricTools/GTEngine/Samples` and executing

```
make CFG=configuration -f makeallsamples.gte
```

If you want to build a single sample application, change directory to the sample folder. For example, change directory to `GeometricTools/GTEngine/Samples/Graphics/VertexColoring` and execute

```
make CFG=configuration APP=VertexColoring -f ../../makesample.gte
```

## 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 4.3 or later, the sample applications will terminate with a message *OpenGL 4.3 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.

GTEngine 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.

## 3.5 Running the Samples

For the static library builds, you can simply open a terminal window and change directory to the project directory. For example, if you built the static release library and the `Graphics/BlendedTerrain` sample application, the application can be launched by executing `./BlendedTerrain.Release`

For shared library builds, the libraries are stored in `GeometricTools/GTEngine/lib`. A simple way to launch the application is the following. Suppose you have a terminal window open and you have changed directory

to `Samples/Graphics/BlendedTerrain` and that you have built the shared release versions of the engine and application. Execute the following

```
LD_LIBRARY_PATH=$GTE_PATH/lib/ReleaseDynamic ./BlendedTerrain.Release
```

## 4 Support for Macintosh OS X is Discontinued

Development on Macintosh OS X has evolved significantly, veering away from standard windowing and graphics support. I do not program in Objective C or Swift, and Metal and OpenCL are preferred over third-party OpenGL. My test machine for Macintosh had been a Mac Mini, and the latest OS X cannot be installed on it. Apple and Macintosh products seem to be designed mainly for consumers, not for scientific computing or graphics. I no longer have the time (or desire) to support the Macintosh.

That said, if you want to use the non-graphics code on a Macintosh, feel free to grab that code and create your own Xcode projects. If you get compiler warnings or errors for the code, send me an email and I will do my best to modify the code to fix those problems. The effort on your part will be to test my changes and let me know whether they worked, in which case I will update the source code accordingly in the distribution.

## 5 Accessing the OpenGL Driver Information

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

The `GL4Engine` 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);
#ifdef GTE_DEV_OPENGL
    parameters.deviceCreationFlags = 1;
#endif
auto window = TheWindowSystem.Create<VertexColoringWindow>(parameters);
TheWindowSystem.MessagePump(window, TheWindowSystem.DEFAULT_ACTION);
TheWindowSystem.Destroy<VertexColoringWindow>(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.

## 6 Automatic Generation of a Wrapper for OpenGL

Several packages may be found online that provide wrappers that query for OpenGL function pointers and that encapsulate many of the gory details in using the OpenGL API. As an alternative, `GTEngine` ships with a tool to build a wrapper, `GeometricTools/GTEngine/Tools/GenerateOpenGLWrapper`. The file `GteOpenGL.h` was written manually—do not modify it. The file `GteOpenGL.cpp` is generated by parsing `glcorearb.h`, a file that is available from the [OpenGL Registry](#). This file is introduced in the OpenGL 4.3 Specification.

If the file is updated or if you want to use a previous version, you can copy your file to the tool's project folder and then use the wrapper tool to regenerate `GteOpenGL.cpp` and copy it to

GeometricTools/GTEngine/Source/Graphics/GL4/GteOpenGL.cpp

You must also copy your version of `glcorearb.h` to

GeometricTools/GTEngine/Include/Graphics/GL4/GL/glcorearb.h

The extension files from the OpenGL Registry were also copied to the same folder as `glcorearb.h`. These include `glxext.h`, `glxext.h` and `wglxext.h`. The first two are not used by the engine, but the last one is used by the WGL OpenGL engine.

The `GenerateOpenGLWrapper` tool has projects for Microsoft Visual Studio. To build this on Linux,

```
g++ -std=c++14 -c GenerateOpenGLWrapper.c -o GenerateOpenGLWrapper.o
g++ GenerateOpenGLWrapper.o -o GenerateOpenGLWrapper
```

and then execute the program in the project folder.