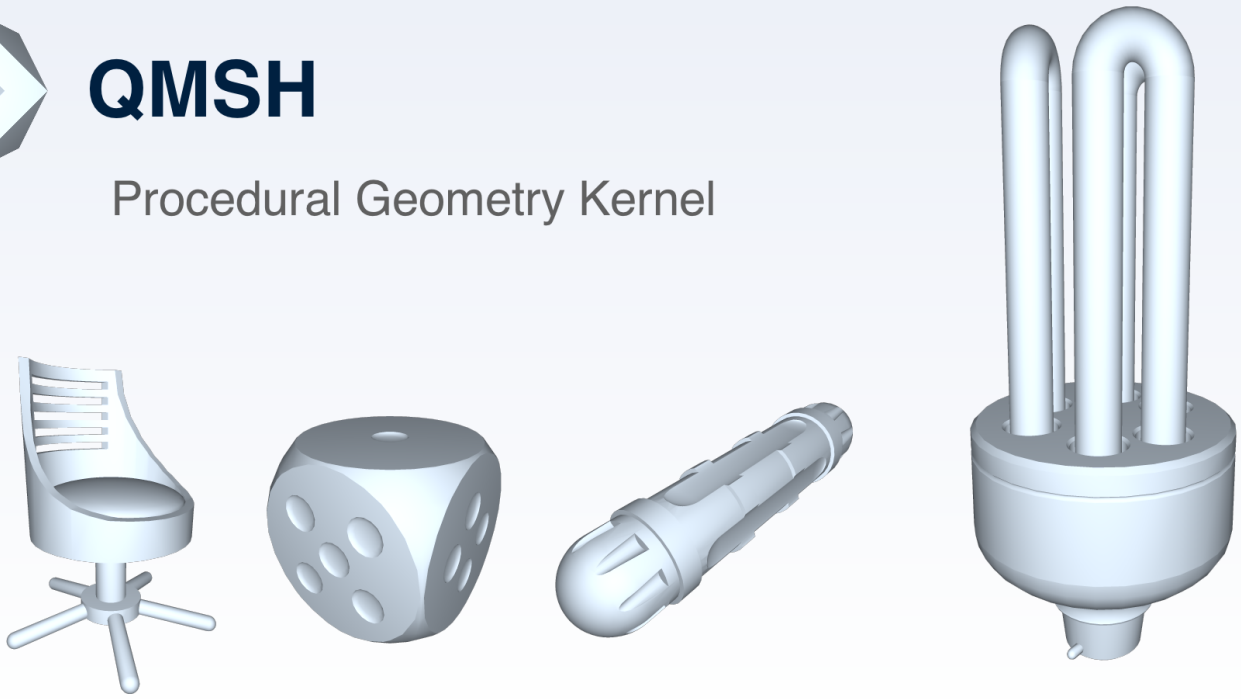
QMSH

Procedural Geometry Kernel



QMSH

Engine-Plugin for Unity

Quick-Start User-Guide: V0

K. Edum-Fotwe
Codemine-Industrial-Software

Keywords and Categories

Geometric-Modelling, Procedural-Modelling, Polyhedral-Mesh, Constructive-Modelling, CSG, Generalised-Cylinders, Parametric-Modelling, Boolean-Logic, Shape-Grammars, Language-Theory, Programming-Language-Constructs, Generative-Modelling, Digital-Content-Creation, Polygonal-Modelling

Abstract

The Quick-Mesh Engine-Plugin is an extension module designed to simplify the process of invoking the Quick-Mesh Kernel within game-engines and digital-content-creation suites. This document provides a concise guide - targeted at advanced users - which covers the use of the Quick-Mesh plugin within the Unity game-engine for the purpose of assembling 3D mesh procedurally in-editor and in-player.

Table of Contents

1 Introduction 4
1.1 Pre-Requisites 4

2 Inputs & Outputs 5

3 Plugin Architecture 6
3.1 In-Player Plugin 8
3.2 In-Editor Plugin 10

4 Example Usage 14
4.1 In-Editor Invocation (Ahead-of-Time) 15
4.2 In-Player Invocation (Runtime) 16

5 Additional Notes 19
5.1 Performance Considerations 20

1 Introduction

This short guide provides practical instructions on the use of the Quick-Mesh Engine-Plugin for Unity on Mac-OS and Windows operating systems. The primary aim of this guide is to help and direct advanced users by explaining the process of utilising the extension-module within the game-engine both in-editor (ahead-of-time) and in-player (at runtime) in order to generate procedurally defined 3D content.

1.1 Pre-Requisites

*** Items Required to Follow this Guide and Model 3D Objects using the Engine-Plugin ***

- Hardware : Desktop or Laptop Computer (64-bit, i.e. x86_64 architecture)
- Operating-System : Mac-OS or Windows
- Software : Unity Game-Engine (minimum supported version 5)
- Software : Quick-Mesh Kernel (a command-line executable program)
- Software : Quick-Mesh Engine-Plugin (an extension module that provides interop facilities)

Additional (Optional) Physical Items That May Be of Use

- Pencil/Pen (i.e. a drawing or writing implement) and Paper
- Ruler/Tape-Measure (to measure dimensions if modelling physically-based entities)

Skills and Technical Experience Necessary to Wield the Engine-Plugin Effectively

- Elementary Understanding of Geometric Modelling Operations and Techniques
  e.g. using Euclidean transformations, types of primitive and polygon mesh attributes.
- Familiarity with Unity's Editor and Development Techniques
  e.g. experience using GameObjects, Transforms, Behaviours, Components, Renderers and the ability to navigate the various Menus and Inspectors (i.e. Scene-Hierarchy, Object and Settings)
- Familiarity with Invoking Programs from the Terminal or Command-Line
  e.g. being able to comfortably control a computer with commands rather than interactively.
- Familiarity with an Imperative Programming or Scripting Language
  e.g. knowing what variables and functions are and where, how and why to use them.
- Imagination + Spatial Reasoning Skills
  ...and a love of geometry.

Note: that this is a technical guide targeted at advanced QMSH users - and does not cover the basics of the scripting language. Readers are expected to already be familiar with the kernel and grammar.

Note: if you are new to procedural modelling, Unity (or programming in general) you may find it easier to begin by trying the Quick-Mesh Mobile-Editor before progressing to using the Engine-Plugin.

2 Inputs & Outputs

The input to the engine-plugin is a quick-mesh script - specified as an object of type String. The output of the engine-plugin is a triangle-mesh representing the entity defined by the input script - which can be rendered by Unity as GameObjects. In essence the plugin provides basic interop facilities that aim to simplify communication with the QMSH kernel (an external executable) within Unity.

Figure 2.0 provides an outline of the relationship between the inputs and outputs produced and consumed by the Unity game-engine, the QMSH plugin and the QMSH kernel - when used in concert.

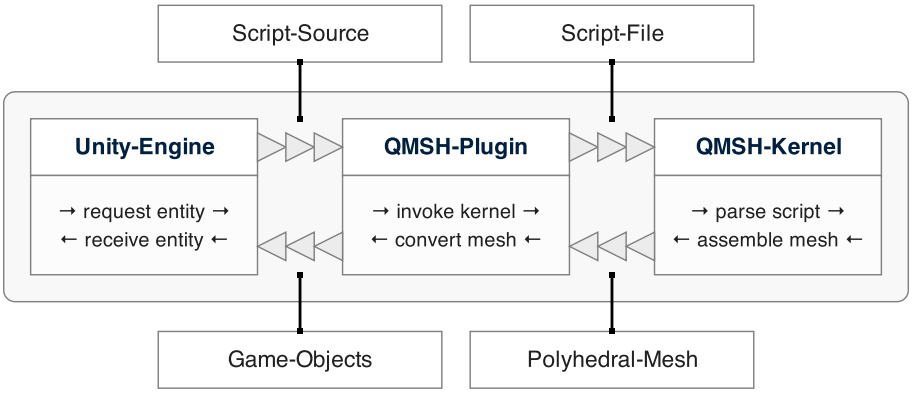


Figure 2.0: schematic outline of the inputs to and outputs of the QMSH engine-plugin.

Observe (in figure 2.0) that the QMSH plugin performs the role of a broker or intermediary - in the sense that it sits between Unity and the QMSH kernel and manages (arranges) the execution of assembly tasks on behalf of an application. Vitally this highlights the fact that the plugin is a convenience and not a mandate. In plain terms it is surprisingly simple to invoke the QMSH kernel directly from within Unity - without the use of the plugin if one so desires. The plugin aims only to make this process more intuitive by providing a coherent working example of how it can be coordinated.

Note: that there are a number of constraints imposed by the use of the plugin - relative to invoking the kernel directly or externally. Foremost: the level of control an application possesses over the assembly process is reduced. For example the plugin only supports the injection of triangulated mesh - meaning that the polygonal elements that the kernel is capable of producing are not exposed by the plugin.

Additionally it only supports generating Unity GameObjects from Mesh containing up to 65336 vertices.

Further - whilst the plugin supports the construction of GameObjects containing Mesh with per-vertex RGBA colours - it does not support the application of UV-texture-coordinates to generated entities.

Although such a task could be addressed with an auxiliary automatic-parameterisation or manual-texture-mapping step within a separate program or DCC system - the plugin does not afford the grain of control that would be required in such cases. This is a key consideration that should be factored into the determination of the utility of the plugin towards meeting a given objective.

To review: the engine-plugin is responsible for feeding input scripts into the QMSH kernel and converting the generated output mesh into GameObjects that can be rendered and simulated by Unity.

3 Plugin Architecture

This section covers the plugin's architecture in terms of the types and functions exposed. It first outlines the benefits and limitations of the two principal usage-patterns (ahead-of-time and at runtime). Then (for each) it enumerates the key behavioural aspects and components.

In-Editor Plugin: Advantages and Disadvantages

- Reduced Runtime CPU Load - Assembly Occurs In-Editor: i.e. only on script-changes (when explicitly requested by the user) and results are cached as Game-Objects with the Scene's data.
- Increased Project Storage Size - 3D-Entities are Stored as concrete Mesh objects rather than as concise String objects - meaning they are as expensive to store as traditional assets.
- Reduced Load Time (on Run/Play) - Scenes Load Faster: because Assets are already assembled.
- Suited to Mixed-Content - Entities can be Manipulated Interactively: using non-procedural tools, techniques and widgets in the editor prior to build - enabling more fluid workflow integration.
- Compatible with WebGL Builds - Entities can be Included in WebGL Scenes: ahead-of-time use means there is not a runtime dependency on the QMSH Kernel (i.e. once constructed entities can be compressed and bundled just as any other asset for browser-compatible builds).
- Applicable to Statics Only - Entities can not be Re-Assembled In-Player: which means the loss of the ability to simulate dynamic entity behaviours by re-creating mesh with altered parameters.
- Relatively Easy Learning Curve - Simpler to Get Working: compared to runtime use.

In-Player Plugin: Advantages and Disadvantages

- Increased Runtime CPU Load - Assembly Occurs In-Player: typically during initialisation and potentially at junctures throughout application use. To prevent blocking the process is threaded.
- Reduced Project Storage Size - 3D-Entities are Stored as concise String objects rather than as concrete Mesh objects: enabling seemingly complex scenes to be stored in KB rather than MB.
- Increased Load Time (on Run/Play) - Scenes Take Longer to Load: because Assets must be assembled directly from their source prior to being injected into GameObjects and rendered.
- Suited-To-Pure-Content - Entities can be Manipulated Procedurally: using programmatic tools and algorithmic constructs - but not directly with interactive techniques provided by the Editor.
- Incompatible with WebGL Builds - Entities are not Suitable for WebGL Scenes: given that in-browser runtime invocation of the QMSH Kernel relies on access to the local file-system or a Javascript/Web-Assembly implementation - both of which are currently not supported.
- Applicable to Statics and Dynamics - Entities can be Re-Assembled In-Player *On-The-Fly*: which enables powerful post-compilation functionality such as parametric-geometric-mutations, runtime-extension-support (i.e. generative plugins for applications), and open-world content-streaming.
- Steeper Learning Curve - More Involved to Coordinate: relative to ahead-of-time use.

Note: both the in-editor and in-player variants of the plugin require an application to explicitly state the absolute file path to the QMSH kernel executable and a staging (interop) directory on the device or machine in question - before they can be used. This is typically coordinated on a once-per-project basis for in-editor use and during root-scene initialisation (upon each launch) when used in-player.

Before progressing to detailing the in-editor and in-player plugin components - an outline of the structure of the distribution (the Unity package containing the QMSH plugin) is provided below.

QMSH-Plugin-Unity/
> **Scripts/**
> > QMSH.cs
> > QMSHEntity.cs
> **Editor/**
> > **Scripts/**
> > > QMSHEntityEditor.cs
> > > QMSHPluginMenu.cs
> > > QMSHSettings.cs
> > > QMSHSettingsWindow.cs
> **Scenes/**
> > QMSH_ExampleSceneA_InEditorInvocation.unity
> > QMSH_ExampleSceneB_InPlayerRuntimeInvocation_Blocking.unity
> > QMSH_ExampleSceneC_InPlayerRuntimeInvocation_Blocking.unity
> > QMSH_ExampleSceneD_InPlayerRuntimeInvocation_Threaded.unity
> > **Controllers/**
> > > QMSH_ExampleSceneA_Controller.cs
> > > QMSH_ExampleSceneB_Controller_Blocking.cs
> > > QMSH_ExampleSceneC_Controller_Blocking.cs
> > > QMSH_ExampleSceneD_Controller_Threaded.cs
> **Materials/**
> > QMSH_RGBDiffuse.mat
> > QMSH_RGBLambert.mat
> > QMSH_RGBStandard.mat
> > QMSH_RGBUnlit.mat
> > **Shaders/**
> > > QMSH_RGBDiffuseShader.shader
> > > QMSH_RGBLambertShader.shader
> > > QMSH_RGBStandardShader.shader
> > > QMSH_RGBUnlitShader.shader
> **Resources/**
> > cube_octahedron.png
> > QMSHPluginConfigurationSettings.asset
> **Documentation/**
> > README.txt
> > qmsh_plugin_guide_unity.pdf

Figure 3.0: the contents of the Unity package containing the QMSH-Plugin listed for reference.

3.1 In-Player Plugin

This section details the structure of the in-player plugin's source-code.

The only script required to use the plugin at runtime (i.e. in-player) is the C-Sharp source-file QMSH.cs - which defines the most important class type in the plugin - QMSH. This section covers the public components of the QMSH class type which you shall use to coordinate runtime assembly tasks.

Specifically - this section enumerates the signatures of the main functions, properties, enumerations and delegates that you can use within your C-Sharp scripts to interact with the QMSH plugin. This is to enable you to quickly gain familiarity with the principal features provided.

The source-code for this type also includes detailed comments which you should refer to for greater insight to the operation of each component. As always the best documentation is the source - and you are encouraged to examine it directly to gain a deeper understanding of the behaviour outlined here.

---

class QMSH
→ k.qmsh.unity.plugin.QMSH *(defined in the source-file QMSH.cs)*

QMSH : Public Nested Enumerations and Delegates

```
→ enum KernelExecutionMode {
                PURE_UNION_MESH
                BASIC_CSG_MESH
                OPTIMAL_CSG_MESH
                };
→ delegate void KernelAssemblyListener ( QMSH assembly_data );
→ delegate void KernelErrorListener ( string error_data );
→ delegate void KernelParseListener ( string parse_data );
```

QMSH : Public Static Global Properties

```
→ string KERNEL_EXECUTABLE_PATH { get, set };
→ string INTEROP_TEMPORARY_PATH { get, set };
```

QMSH : Public Static Global Functions

```
→ QMSH ASSEMBLE ( string script_source_code );
→ QMSH ASSEMBLE ( string script_source_code, KernelExecutionMode kernel_execution_mode );
→ QMSH ASSEMBLE (
                string script_source_code,
                KernelExecutionMode kernel_execution_mode,
                string kernel_executable_path,
                string interop_temporary_path
                );
→ bool ASSEMBLE_IN_BACKGROUND_THREAD (
                string script_source_code,
                KernelExecutionMode kernel_execution_mode,
                string kernel_executable_path,
                string interop_temporary_path,
                KernelAssemblyListener on_completion_callback,
                KernelErrorListener on_error_callback
                );
→ void INIT ( string kernel_executable_path, string interop_temporary_path );
```

```
→ string PARSE ( string srcipt_source_code );
→ string PARSE ( string script_source_code, string kernel_executable_path, string interop_temporary_path );
→ bool PARSE_IN_BACKGROUND_THREAD (
                string script_source_code,
                string kernel_executable_path,
                string interop_temporary_path,
                KernelAssemblyListener on_completion_callback,
                KernelErrorListener on_error_callback
                );
```

QMSH : Public Instance Member Properties

```
→ string trace { get };
→ bool is_empty { get };
```

QMSH : Public Instance Member Functions

```
→ GameObject to_UGO (
                Transform parent,
                string name,
                Material material,
                bool colliders,
                bool shadows
                );
→ Mesh to_UMSH ( );
```

In order to create an instance of a QMSH object you typically invoke one of the type's blocking public static ASSEMBLE(...) functions on application start with an appropriate quick-mesh script, kernel-executable-path and interop-temporary-path set. You can repeat this process multiple times to assemble multiple QMSH objects sequentially in a blocking fashion. The QMSH type also provides a non-blocking variant ASSEMBLE_IN_BACKGROUND_THREAD(...) which you can use post-application-initiation to assemble entities without halting Unity's main thread. Once you have a valid (non-null and non-empty) QMSH object reference - you can call the type's instance member functions to_UMSH() to create a Unity Mesh object or to_UGO(...) to create a Unity GameObject - depending on how you intend to use the mesh. For example if your application makes heavy use of GameObjects to coordinate rendering you would typically call to_UGO(...), whereas if your application by-passes GameObjects in favour of using Unity's GL classes to directly draw mesh you would call to_UMSH() instead.

Note: that whilst you can safely thread invocation of the type's static ASSEMBLE(...) functions yourself - you must invoke the type's instance functions to_UMSH() and to_UGO(...) from Unity's main thread.

Extension Notes and Guidelines for Revision:-

Once you are familiar with the QMSH type and the operation of the plugin at runtime you might choose to extend or refactor it to better suit your particular application's requirements. In such cases the following exercises represent good starting points for experimentation and further development.

- Adding Support for Faster Binary Mesh Interchange Formats.
- Partitioning Mesh with More than 65336 Vertices.
- Auto-Generating UV Texture-Coordinates for Mesh.
- Experimenting with Procedural 3D-Material Shaders.

3.2 In-Editor Plugin

This section details the structure of the in-editor plugin's source-code - which extends the runtime plugin's source-code (QMSH.cs) with types that simplify mesh assembly during scene construction.

Note: that for in-editor use you do not necessarily need to touch the source-code directly - because the functionality provided is exposed through component inspectors (see figures 3.2a, 3.2b). However you may find that an awareness of the structure of the source helps you to make better use of the plugin.

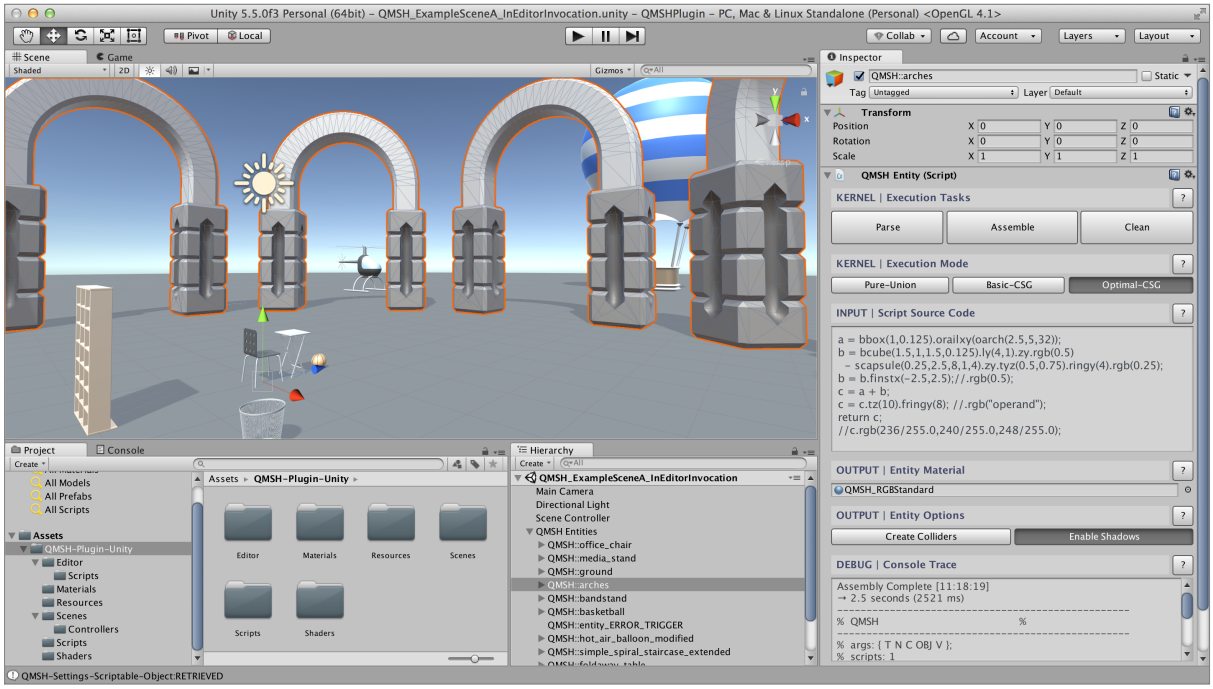


Figure 3.2a: screenshot of the QMSH Plugin running in the Unity Editor with an example scene open and the procedurally defined entity (arches) selected - the right-hand inspector illustrates the appearance of the QMSH-Entity component's Editor-GUI which is the main interface to interaction with the plugin for ahead-of-time use.

The plugin's QMSHEntity and QMSHEntityEditor class types are responsible for managing (respectively) the addition of quick-mesh entity generator components to GameObjects and the display of corresponding custom controls in the Editor's inspector whenever an entity is selected.

---

class QMSHEntity : UnityEngine.MonoBehaviour
→ k.qmsh.unity.plugin.QMSHEntity *(defined in the source-file QMSHEntity.cs)*

QMSHEntity : Public Nested Delegates
→ **delegate void** UnityInjectionTask ( );

QMSHEntity : Public Instance Member Functions
→ **void** ASSEMBLE ( );
→ **UnityInjectionTask** ASSEMBLE(**bool[]** done_flag, **bool[]** cancel_flag);
→ **void** CLEAN ( );

→ **void** PARSE ( );
→ **void** COLLIDERS (**bool** colliders);
→ **void** MATERIAL (**Material** material);
→ **void** SHADOWS (**bool** shadows);

QMSHEntity : Public Instance Member Variables
→ **string** QMSHSourceCode;
→ **KernelExecutionMode** ExecutionMode;
→ **bool** CreateColliders;
→ **bool** EnableShadows;
→ **Material** RGBAMaterial;

QMSHEntity : Public Instance Member Properties
→ **bool** is_locked { **get** };
→ **string** trace { **get** };

Note: although you can also safely use the QMSHEntity component type in standalone desktop builds - the QMSHEntityEditor type can only be used within the Unity-Editor. Refer to §4.2 for further details.

---

class QMSHEntityEditor : UnityEditor.Editor
→ k.qmsh.unity.plugin.QMSHEntityEditor
(defined in the source-file QMSHEntityEditor.cs)

QMSHEntityEditor : Public Instance Member Functions
→ **void** OnInspectorGUI();

The operation of the QMSHEntity inspector should be relatively easy to intuit - however if you get stuck or are unsure of anything - you can use the righthand tooltip buttons (?) to display help dialogs explaining the behaviour of each feature (see figure 3.2b).

Note: that the functionality exposed by the QMSHEntity inspector largely mirrors the functionality exposed by the QMSH type - with one key exception - which is the added ability to easily cancel assembly tasks.

Additionally observe the presence of both member variables and functions (in the QMSHEntity type) to address seemingly the same tasks of configuring entity state. The distinction is that alterations to the member variables will only take effect after the next assembly task - whilst invocation of the member functions causes an immediate synchronisation of state with any previously instantiated GameObjects.

With the per-entity types covered - the in-editor types that manage global plugin settings are covered next.

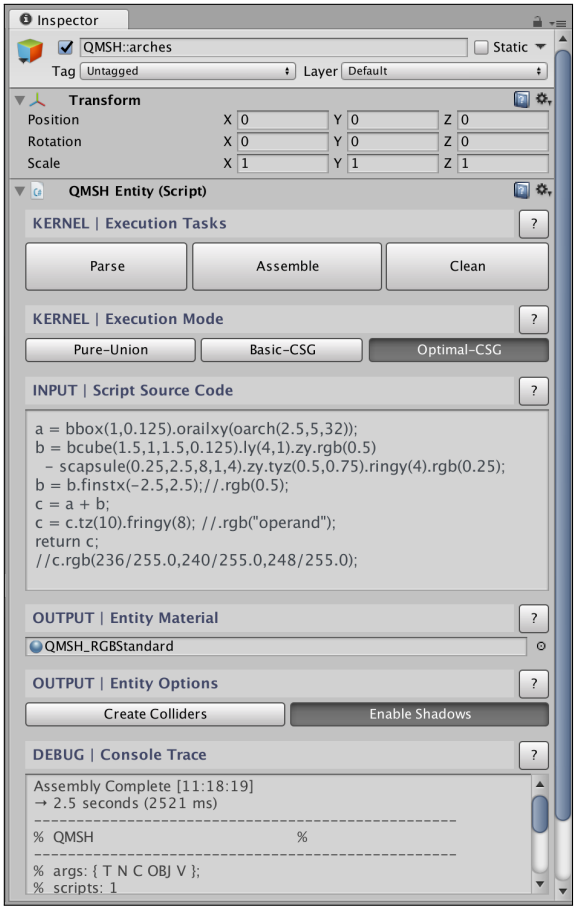


Figure 3.2b: close-up screenshot of the QMSHEntity component's custom inspector - which is rendered by the accompanying QMSHEntityEditor type.

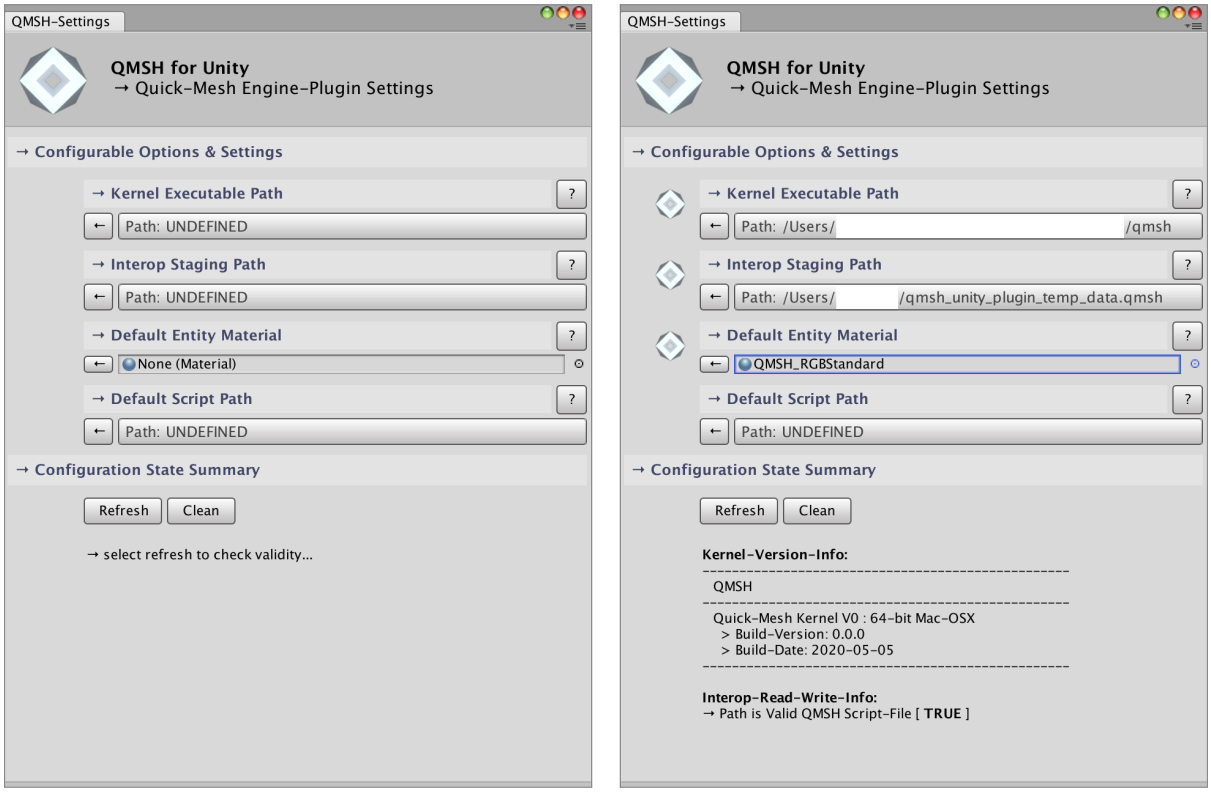


Figure 3.2c: screenshots of the plugin's settings window which is used to specify the configuration of the plugin - including slots for specifying the mandatory settings (kernel-executable-path and interop-staging-path) which are required for plugin use - and the optional settings (default-entity-material and default-script-path).

The QMSHSettingsWindow type is an auxiliary class which manages global plugin settings and provides a simple dialog for configuring these settings (see figure 3.2c). Note: that as a derivative of the UnityEditor.EditorWindow type it can only be used within the Editor and not in the Player.

In order to support persistent storage of the plugin's settings (across unloads and reloads) - the additional QMSHSettings type implements a singleton scriptable-object which enables storing the mandatory and optional settings as a project asset. Whenever you change one of the settings (through the QMSHSettingsWindow) the QMSHSettings instance is automatically updated to reflect the change.

---

class QMSHSettingsWindow : UnityEditor.EditorWindow
→ k.qmsh.unity.plugin.QMSHSettingsWindow *(defined in the source-file QMSHSettingsWindow.cs)*

QMSHSettingsWindow : Public Static Global Functions
→ **void** DISPLAY ( );

Note: there is a one-to-one mapping between the options exposed interactively by the QMSHSettingsWindow type and the public members of the QMSHSettings scriptable-object type.

class QMSHSettings : UnityEngine.ScriptableObject
→ k.qmsh.unity.plugin.QMSHSettings *(defined in the source-file QMSHSettings.cs)*

QMSHSettings : Public Instance Member Variables
→ **string** KERNEL_EXECUTABLE_PATH;
→ **string** INTEROP_TEMPORARY_PATH;
→ **Material** DEFAULT_ENTITY_MATERIAL;
→ **string** DEFAULT_SCRIPT_PATH;

QMSHSettings : Public Static Global Properties
→ **QMSHSettings** REF { **get** };

Finally the QMSHPluginMenu type adds a drop-down menu to the Unity Editor which provides shortcuts for performing plugin actions - such as creating new entities, loading example entities or script files and displaying the plugin's settings window. Note: that this type does not expose any public members.

QMSHPluginMenu : UnityEngine.MonoBehaviour
→ k.qmsh.unity.plugin.QMSHPluginMenu *(defined in the source-file QMSHPluginMenu.cs)*

No Publics Defined

To review: the in-editor plugin's source extends the in-player plugin's source (QMSH.cs) with five supplementary class types that simplify interactive use of the plugin through Unity's interface.

These additional in-editor types are: the QMSHEntity class, the QMSHEntityEditor class, the QMSHSettingsWindow class, the QMSHSettings class and the QMSHPluginMenu class.

***

Now that you have a deeper appreciation of the plugin's underlying architecture and some of the important trade-offs inherent to both in-editor and in-player plugin use-cases, the next section provides a number of practical (hands-on) examples demonstrating how to invoke the QMSH-Kernel within Unity using the components detailed in this section - both ahead-of-time and at runtime.

4 Example Usage

This section's examples demonstrate how to use the QMSH-Plugin in the Unity-Editor and Unity-Player.

- **In-Editor Invocation: Work-Flow Break-Down (Step-By-Step Instructions)**
  This example demonstrates the construction of a simple multi-object scene within the Unity-Editor - for which each GameObject is the result of assembling a Quick-Mesh script. This example represents the archetype for in-editor use of the plugin. The work-flow steps covered are applicable to any project that takes the ahead-of-time approach to use of the QMSH engine-plugin.
  Key Files: [QMSH_ExampleSceneA_InEditorInvocation.unity, QMSH.cs, QMSHEntity.cs, QMSHEntityEditor.cs, QMSHPluginMenu.cs, QMSHSettings.cs, QMSHSettingsWindow.cs, cube_octahedron.png, QMSHPluginConfigurationSettings.asset, QMSH_ExampleSceneA_Controller.cs, *.mat, *.shader]

- **In-Player Invocation: Blocking-Assembly with QMSH Type**
  This example demonstrates the construction of a simple multi-object scene within the Unity-Player at runtime. This example covers the minimum requirements for runtime invocation in a blocking fashion (i.e. assembly on program launch) - using the principal plugin class type QMSH.
  Key Files: [QMSH_ExampleSceneB_InPlayerRuntimeInvocation_Blocking.unity, QMSH_ExampleSceneB_Controller_Blocking.cs, QMSH.cs, *.mat, *.shader]

- **In-Player Invocation: Blocking-Assembly with QMSH Type and QMSHEntity Type**
  This example demonstrates the construction of a simple multi-object scene within the Unity-Player at runtime - with post-instantiation re-assembly coordinated with the help of the QMSHEntity class type. This example covers the steps required to manage QMSHEntity instances dynamically.
  Key Files: [QMSH_ExampleSceneC_InPlayerRuntimeInvocation_Blocking.unity, QMSH_ExampleSceneC_Controller_Blocking.cs, QMSH.cs, QMSHEntity.cs, *.mat, *.shader]

- **In-Player Invocation: Non-Blocking-Assembly with QMSH Type**
  This example demonstrates the construction of a simple mono-object scene within the Unity-Player at runtime - with the assembly process coordinated from a background thread. This example uses an animated oscillating slider-bar to provide visual feedback to a user that assembly is occurring. This example's use of the QMSH class represents the archetype for non-blocking runtime mesh assembly which can be particularly helpful for long-running assembly tasks.
  Key Files: [QMSH_ExampleSceneB_InPlayerRuntimeInvocation_Threaded.unity, QMSH_ExampleSceneB_Controller_Threaded.cs, QMSH.cs, *.mat, *.shader]

4.1 In-Editor Invocation (Ahead-of-Time)

QMSH_ExampleSceneA_InEditorInvocation.unity
QMSH_ExampleSceneA_Controller.cs

The steps involved in creating entities with the plugin within the Unity Editor are enumerated in figure 4.1 for reference. Remember that these steps are applicable to any project which takes the ahead-of-time approach to use of the plugin. As such - once you have followed these steps - you can then repeat them as many times as you require to add as many entities as your project necessitates.

---

0 | **Initialise QMSH Configuration Settings (Once-Per-Project)**

→ Menu: [QMSH > Settings]

---

1 | **Create New Entity, Open Script-File Entity or Load Example Entity**

→ Menu: [QMSH > New > Entity]
→ Menu: [QMSH > Open > Script]
→ Menu: [QMSH > Examples > ...]

---

2 | **Configure Entity with Editor-Inspector**

→ Select Kernel-Execution-Mode
→ Edit Input-Script-Source-Code
→ Select Output-Entity-Material
→ Toggle Output-Entity-Options

---

3 | **Execute Entity Kernel-Task**

→ Parse Entity Script
→ Assemble Entity Mesh
→ Clean Entity Resources

Figure 4.1: instructions covering the steps involved in creating QMSH entities in the Unity-Editor (ahead-of-time).

The instructions in figure 4.1 can be applied directly to the accompanying example scene QMSH_ExampleSceneA_InEditorInvocation.unity in order to add new entities and experiment with the in-editor plugin. You can also skip step 1 and jump directly to configuring and re-assembling the scene's built-in demo entities. Once you are familiar with these steps you can then apply them to creating a new scene from scratch or to adding procedurally defined entities to scenes from other projects.

Note: if you are unsure of what an option provided by the plugin does - use the option's adjacent help button (?) to display a help dialog that explains the behaviour of the option. This applies to both the QMSHEntity component's inspector and the QMSHSettingsWindow configuration dialog.

4.2 In-Player Invocation (Runtime)

This section covers the example scenes that demonstrate the various ways in which you can coordinate runtime invocation of the QMSH-Kernel with the assistance of the plugin.

Note: for each of the runtime invocation example scenes that follow - the kernel-execution-path (EXE) and the interop-stating-path (TEMP) must be set prior to running each scene. This can be handled through the Editor or by editing the source-code for each scene's controller component directly.

QMSH_ExampleSceneB_InPlayerRuntimeInvocation_Blocking.unity
QMSH_ExampleSceneB_Controller_Blocking.cs

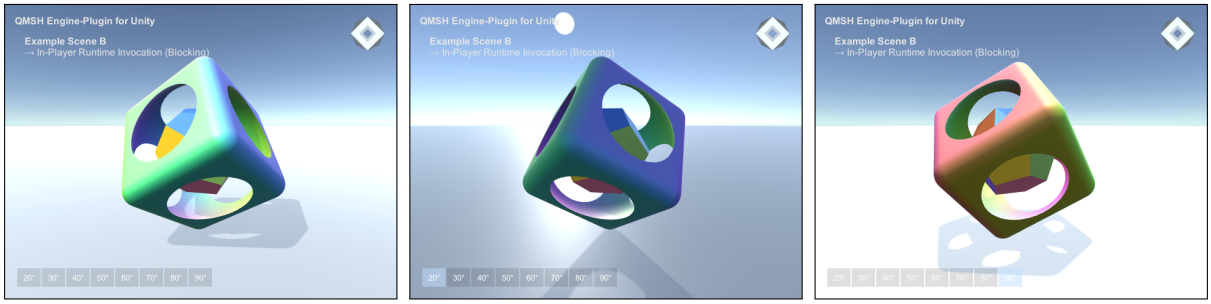


Figure 4.2a: screenshots of the result of running the first runtime-invocation example-scene - depicting the two simple abstract entities and the scene's directional-light being rotated independently post assembly.

The first runtime example scene acts as a minimalistic template of the steps required to invoke the quick-mesh kernel programmatically. Figure 4.2a illustrates the result of running the scene in the editor. Note: that this example generates two static entities using the principal plugin type - QMSH. The update function for this example simply rotates the entities and the scene's light. Note: the OnGUI callback (used by Unity's legacy 2D GUI-system) is used in this example to display overlain title-text, the plugin-icon and a simple action-bar whose buttons alter the angle of the directional-light.

QMSH_ExampleSceneC_InPlayerRuntimeInvocation_Blocking.unity
QMSH_ExampleSceneC_Controller_Blocking.cs

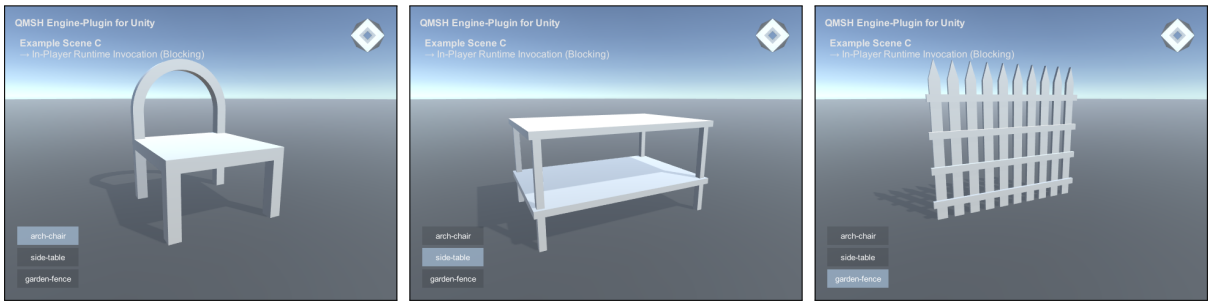


Figure 4.2b: screenshots of the result of running the second runtime-invocation example-scene - depicting a set of simple static entities which are assembled (post-instantiation) in response to user-actions.

The second runtime example scene demonstrates the use of the QMSHEntity type as a complement to the QMSH type. In particular the QMSHEntity type is used in order to simplify the coordination of post-instantiation re-assembly tasks in response to user-triggered events. Figure 4.2b illustrates the result of running the scene in the editor. The scene's setup is relatively easy to follow. The action-bar buttons (drawn in the OnGUI function) are used to trigger runtime re-assembly - which is coordinated in the update function with the assistance of the QMSHEntity type. Note: although this scene blocks the

main-thread during re-assembly - the omission of animation means this undesirable behaviour is almost imperceptible to an end-user. Generally: you should aim to avoid blocking the main-thread (see §5.1 for more) - however in certain situations (especially when dealing with entities that have been tested ahead-of-time and are empirically known/shown to be fast to assemble - i.e $\lesssim\approx$ 200 ms) - this rule-of-thumb can sometimes be circumvented - so to avoid the overhead of added threading latency.

QMSH_ExampleSceneD_InPlayerRuntimeInvocation_Threaded.unity
QMSH_ExampleSceneD_Controller_Threaded.cs

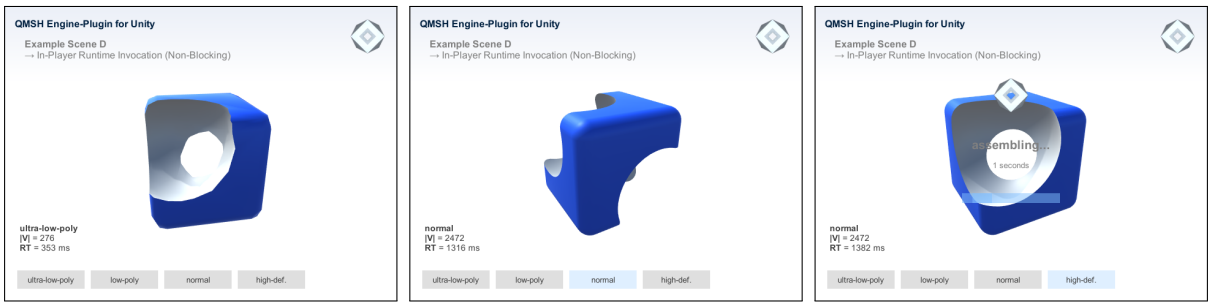


Figure 4.2c: screenshots of the result of running the third runtime-invocation example-scene - depicting a simple abstract entity (assembled at varying levels-of-detail) being rotated with an animated task-progression-indicator overlain to convey the non-blocking nature of the assembly process when invoked from a background thread.

The third runtime example scene provides a template of how to coordinate runtime mesh assembly from a background-thread so as not to block Unity's main-thread - using the QMSH type. Figure 4.2c illustrates the result of running the scene in the editor. Although this example's source is slightly more involved it should still be relatively easy to follow. The key difference is that rather than invoking the QMSH type's static ASSEMBLE(...) functions - the type's ASSEMBLE_IN_BACKGROUND_THREAD(...) function is invoked instead with delegates supplied to respond to the AssemblyComplete and AssemblyError callbacks. Note: that in invoking QMSH from a background-thread there is a slight added latency relative to invocation from Unity's main-thread. This latency can range from $\approx$ 100-500 ms depending on factors such as the operating-system and device/machine characteristics. An interesting aspect of this scene (from a behavioural perspective) is the re-assembly in response to user-interaction. The thing to observe is that the total assembly runtime for this scene varies depending upon the density or level-of-detail selected for the abstract entity. In this manner it is plain to see that: A) the animation of the scene continues uninterrupted as re-assembly occurs (irrespective of the complexity/size/workload-/overhead) of the background task, and that B) the difference in runtime between the low-poly entities and the higher-density variants is substantial - and crucially often non-linear in its growth. One additional note relates to the manner in which this scene marshals the assembled polyhedral entities between the background-thread and Unity's main-thread. Specifically the example's source does not touch (mutate or access) any Unity-Engine types in the delegate callbacks - it simply binds (i.e. assigns to a member variable) the resulting QMSH instance such that it can be accessed organically from Unity's main thread in the update function - where the actual creation of GameObjects from QMSH instances occurs.

As a supplement to the example-scenes - this section closes with a summative source-snippet which can used as a reference of the core steps in using the plugin at runtime to assemble 3D mesh.

```
using UnityEngine;
using QMSH = k.qmsh.unity.plugin.QMSH;

public class MinimalisticRuntimeTest :  MonoBehaviour {

    public string EXE;
    public string TEMP;
    public Material RGBAMTL;

    void Start() {

        if (!QMSH.INIT(EXE,TEMP)) { print("INVALID-CONFIG"); return; }

        string src = "cube - sphere(1.25).rgb(0,0.5,1)";
        QMSH msh = QMSH.ASSEMBLE(src);
        GameObject ugo = msh.to_UGO(null,"xmpl",RGBAMTL,false,false);

        string src2 = "dodecahedron(0.5).hsb(0.625)";
        QMSH msh2 = QMSH.ASSEMBLE(src2);
        GameObject ugo2 = msh2.to_UGO(null,"xmpl2",RGBAMTL,false,false);
    }
}
```

Figure 4.2d: stripped-down (bare-bones) source-code demonstrating an example component that can be added to a GameObject to coordinate runtime-invocation with the plugin - as a reference of the main steps involved.

5 Additional Notes

Finally this section closes this guide with additional notes and performance considerations.

Adding Additional Unity Components to QMSH Entities

The manner in which the plugin generates GameObjects differs slightly depending upon whether or not the QMSH type is used directly or the QMSHEntity type is used as a proxy to direct invocation of the QMSH type's static functions. In-particular: the QMSH type's to_UGO(...) function enables one to minimise the number of GameObjects that are instantiated - as it does not automatically wrap the actual Mesh component holding GameObject in a parent GameObject. The QMSHEntity type on the other hand involves the use of a (technically superfluous) wrapper GameObject - as this slight redundancy trivialises management of entities within the Unity-Editor - and acts as a suitable basis for extension to managing mesh containing greater than 65336 vertices. As such bear in mind the following considerations and caveats if you need to add further behavioural components (after mesh assembly) to GameObjects generated by the plugin. For GameObjects generated by the QMSH type it is safe to add MonoBehaviours directly to their instances - however if you destroy these GameObjects you will have to re-add any components the next time you assemble. For GameObjects generated via the QMSHEntity type it is safe to add MonoBehaviours directly to their parent wrapper instances - or to their child instances (which hold the actual MeshFilter and MeshRenderer components). If you add components to the child instances - the same destroy rule applies as for the QMSH type. If on the other hand you add components to the parent wrapper instances - the components (including their state) will persist across parse, assembly and clean tasks. Note: though however that you may also need to augment the components to operate on the parent instance's Transform's children - depending upon whether the auxiliary component(s) require(s) access to the actual mesh data (e.g. to add UV-data or tangents) or simply to a root reference (e.g. for applying a transformation or animated state transition).

Kernel-Execution Meshing-Modes

The kernel-execution meshing-mode dictates the manner in which the quick-mesh kernel handles polyhedral mesh assembly. At a high-level one can think of the meshing-mode as loosely analogous to the algorithm the kernel uses to arrange mesh elements. Each meshing mode has its own advantages and limitations and provides certain behavioural features which make each suited to serving distinct use-cases. For example the Pure-Union-Mesh mode is fast to assemble but only supports CSG union operations (i.e. no difference or intersection operations). The Basic-CSG-Mesh mode is handy for debugging errors in the kernel's automatic minimal-vertex-optimisation - but generally yields entities that are unsuitable for direct use. The Optimal-CSG-Mesh mode offers the most versatile option in terms of supporting all CSG operations and generating low-poly (minimal-vertex), render-ready entities - however it is also the most expensive of the supported modes. Note: that the mode naming convention applied by the QMSH-Plugin mirrors that employed by the QMSH-Editor. You can therefore refer directly to the kernel-execution meshing-mode portion of the Run Script Menu-Dialog section (§6.2) of the QMSH-Editor's PDF Quick-Start-Guide for further information regarding each meshing-mode.

External Kernel Architectural Considerations

The QMSH-Kernel executes as an external process - which means that its operation is independent of the Unity-Engine. This is in contrast to an internal kernel architecture - wherein the QMSH-Kernel would alternatively be compiled as a native-library and bundled with the plugin. The decision taken for the QMSH-Plugin (to employ an external kernel integration strategy) comes with its own set of strengths and limitations. The primary benefits of the external kernel approach are: stability, resource-management, portability and updatability - whilst the key constraints are: efficiency and the external-dependency. Specifically: in terms of stability: the external kernel approach enables task-isolation (i.e. the separation of potentially heavy-weight 3D modelling tasks undertaken by QMSH, from the realtime interactive functionality provided by Unity). By isolating these processes the plugin supports fail-safe behaviour -

in that an error or issue in the execution of the QMSH-Kernel will not (directly) cause the Unity-Engine to crash. Although this behaviour is not unique to the external approach it is (anecdotally) far easier to coordinate through process-level task-isolation than via the internal native-library approach. In plain terms: having the plugin use an external kernel (that is independent of Unity) decouples the potential problems that may naturally occur during use and in doing so ensures a stable workflow. In terms of resource-management: the external kernel approach enables the immediate recovery of computational resources (used during mesh assembly) at the point of process termination. This essentially ensures that any memory that has to be allocated to generate a 3D entity is reclaimed by the operating system independently of Unity - which removes the possibility of memory-leaks (resulting from mesh assembly) manifesting in Unity. In plain terms: the ability to forcibly kill the external kernel process (if need be) supports low-overhead, leak-free operation. In terms of portability: the external kernel approach removes the requirement to bundle multiple versions of a native library for each supported architecture - which has the secondary-benefit of reducing the overall size of the plugin. The key aspect is that it shifts the responsibility of ensuring the QMSH-Kernel is present (on the device or machine in question) from the application developer to the application user. In terms of updatability: the external kernel approach means that applications that make use of the plugin can benefit from updates to the QMSH-Kernel without having to recompile or rebuild their corresponding Unity projects. In contrast to the advantages of the external kernel approach - the main constraint is a reduction in efficiency relative to the internal kernel approach - which is equivalent to the variance in overhead between a function call and a system-call. In plain terms: calling an external executable is substantially slower than invoking a function. However given that the proportion of work that this constitutes for each mesh assembly task is generally negligible (relative to the actual geometric processing) - this constraint is far out-weighed by the external approach's benefits. The behaviour that could conceivably be vastly improved by the use of the internal kernel approach is the removal of the file-IO overhead associated with marshalling mesh data between applications (i.e. QMSH writing interchange files and Unity reading/loading those files).

5.1 Performance Considerations

The key performance consideration (applicable specifically to the plugin) relates to the overhead associated with mesh-assembly at runtime. Specifically mesh-assembly can be a time consuming process. As such it is generally preferable to thread the process to avoid blocking Unity's main-thread. The one key exception to this is first-time assembly (i.e. on application startup) - which can generally be effected in a blocking-fashion without detriment to the end-user's experience subject to an appropriate loading message being displayed. In all other cases - of post-instantiation mesh-assembly - one should aim to coordinate the process exclusively from background threads. Note: that in some instances (particularly for mesh approaching the 65336 vertex-count limit) rendering hiccups can occur during injection of the assembled polyhedral elements into Unity Mesh and GameObjects. These momentary hiccups are most perceptible when MeshColliders are also generated. If such behaviour is inappropriate or insufficient for a target application then consider the use of co-routines to break the injection task down over several frames - such that its impact on Unity's main-thread is reduced.

***

So ends this guide. Remember to check the project homepage for updates - and have fun meshing!