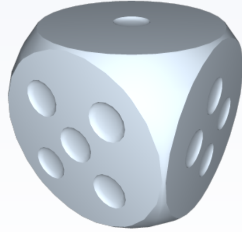
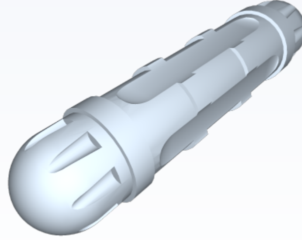
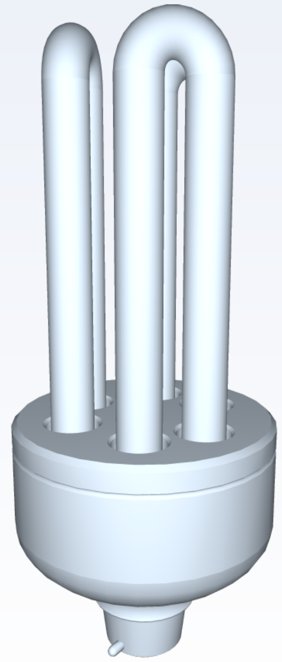
QMSH

Procedural Geometry Kernel







QMSH

Command-Line Kernel

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 Kernel is a general-purpose, platform-agnostic, polyhedral procedural-modelling kernel which exposes a high-level, imperative geometric scripting language. This document provides a short, informal guide to the use of the Quick-Mesh Kernel at the command-line and within user-written programs on Linux, Mac-OSX and Windows - for the purpose of assembling 3D mesh algorithmically.

Notice of Rights

All rights reserved. No part of this publication may be reproduced or transmitted in any form by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the publisher - or in accordance with the provisions of the Copyright, Designs and Patents Act 1988. Any person who does any unauthorised act in relation to this publication may be liable to criminal prosecution and civil claims for damages.

Notice of Liability

The information in this publication is distributed on an *AS IS* basis, without warranty. While every precaution has been taken in the preparation of this publication, neither the author(s) nor publisher(s), shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained in this publication or by the computer software and hardware products described in it.

Trademarks

Quick-Mesh, *qmsh*, *QMSH*, *.qmsh* and the *qmsh-logo* are trademarks of K. Edum-Fotwe and Codemine-Industries.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this publication, they are used in referential fashion only and for the benefit of such companies with no intention of infringement of the trademark. No such use, or the the use of any trade name, is intended to convey endorsement or other affiliation with this publication, its author(s) or its publisher(s).

Table of Contents

1 Introduction 4

1.1 Pre-Requisites 4

2 Inputs & Outputs 5

3 Executable File 5

4 Command Format 5

5 Command-Line Options 6

5.1 Option-Flags to Control General-Operation of the Kernel 6

5.2 Option-Flags to Control Output 3D Model File Formats 6

5.3 Option-Flags to Control Output Geometric Attributes 7

5.4 Option-Flags to Control Constructive-Modelling Operations 7

5.5 Option-Flags to Control Mesh Post-Processing Operations 7

6 Example Usage 8

6.1 Command-Line Invocation 9

6.2 Runtime Invocation via System Calls 10

6.2.1 Runtime Invocation : C 11

6.2.2 Runtime Invocation : C++ 13

6.2.3 Runtime Invocation : C# 15

6.2.4 Runtime Invocation : Groovy 17

6.2.5 Runtime Invocation : Java 18

6.2.6 Runtime Invocation : Objective-C 19

6.2.7 Runtime Invocation : Python 21

7 Additional Notes 22

7.1 Performance Considerations 23

1 Introduction

This short guide provides practical instructions on the use of the Quick-Mesh Command-Line Kernel for Linux, Mac-OSX and Windows operating systems. The primary aim of this guide is to help and direct advanced users by explaining the process of invoking the command-line tool and the meaning and behaviour of the option flags that control the assembly (the creation/construction) of 3D model files.

1.1 Pre-Requisites

*** Items Required to Follow this Guide and Model 3D Objects using the Command-Line Kernel ***

- Hardware : Desktop or Laptop Computer (32-bit or 64-bit, i.e. x86 or x86_64 architecture)
- Operating-System : Linux, Mac-OSX or Windows
- Software : Quick-Mesh Kernel (a command-line executable program)
- Software : 3D-Mesh Viewer-Application (to visualise the output mesh - such as meshlab)
- Software : Text-Editor (to edit input scripts - i.e. vim, emacs, gedit, atom, notepad++, ...)

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 Command-Line Kernel Effectively

- Elementary Understanding of Geometric Modelling Operations and Techniques
  e.g. using Euclidean transformations, types of primitive and polygon mesh attributes.
- 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 (or programming in general) you may find it easier to begin by trying the Quick-Mesh Mobile-Editor before progressing to using the Command-Line Kernel.

2 Inputs & Outputs

The input to the command-line kernel is a set of (one or more) Quick-Mesh scripts - which are plain-text files written in the QMSH grammar (the kernel's scripting language) - and stored with file extension *.qmsh*. The output of the command-line kernel is a set of (one or more) 3D model-files.

The command-line kernel outputs mesh data in the following 3D model file-formats:

- **OBJ** - Alias-Wavefront's 3D Object-Format
- **OFF** - Object-File-Format
- **PLY** - Stanford Polygon Format
- **STL** - Stereolithagraphy Format (or Standard Triangle/Tessellation Language)

The attributes and features supported for each output 3D model file-format are specified in figure 1.

Format	vertices	triangles	polygons	normals	colours	smooth-groups	ascii	binary
OBJ	✓	✓	✓	✓	✓	✓	✓	-
OFF	✓	✓	✓	✓	✓	✓	✓	-
PLY	✓	✓	✓	✓	✓	✓	✓	✓
STL	✓	✓	-	✓	-	-	✓	✓

Figure 1: table of the attributes supported for the command-line kernel's output 3D model file formats.

3 Executable File

Under Linux, Mac-OSX (and similar Unix-based) operating systems the command-line kernel's executable file is named **qmsh**. However under Windows operating systems it is named **qmsh.exe**.

4 Command Format

The format of the command for invoking the command-line kernel is specified below:

qmsh [OPTION-FLAGS...] [INPUT-SCRIPT-PATHS...]

where:

- **qmsh** - is the executable file to invoke: i.e. qmsh for Linux and OSX or qmsh.exe for Windows.
- **[OPTION-FLAGS...]** - is a sequence of command-line control-arguments - each of which begins with a dash/hyphen, and with each seperated (delimited) by a space character. Refer to the command line options section for more detail on the supported control flags.
- **[INPUT-SCRIPT-PATHS...]** - is a sequence of one or more input script file-paths - each of which ends with the file extension .qmsh - and each of which shall be evaluated in the order in which they are specified. The script files paths can either be absolute or relative.

5 Command-Line Options

5.1 Option-Flags to Control General-Operation of the Kernel

-help | -man | -manual | -options | -use | -usage → *print program use and option summary*

Instructs the kernel to print program usage summary information to the terminal.

-about | -version | -info → *print program version information*

Instructs the kernel to print information (a summary-message) about the build version.

-v | -verbose → *execute in verbose logging mode*

Instructs the kernel to execute with a higher-level of logging messages than by default.

-d | -dir | -directory → *write output mesh to a (succeeding) user-specified directory*

Instructs the kernel to export generated mesh to a specific output directory. Note: that by default the kernel saves each mesh in the same directory as its defining script. This option-flag enables one to override this behaviour globally for every script in the command. Note: that the specified directory MUST already exist. The kernel (purposefully) does not invoke the mkdir command on one's behalf, and as such it is the invoker's responsibility to ensure the specified directory exists, prior to invocation.

5.2 Option-Flags to Control Output 3D Model File Formats

-obj → *export 3D mesh in alias-wavefront's model-format*

Instructs the kernel to write ASCII .obj files for each of the input-scripts in the command.

-off → *export 3D mesh in the object-file-format*

Instructs the kernel to write ASCII .off files for each of the input-scripts in the command.

-ply → *export 3D mesh in the stanford-polygon-format*

Instructs the kernel to write .ply files for each of the input-scripts in the command.

-stl → *export 3D mesh in the stereolithagraphy file-format*

Instructs the kernel to write .stl files for each of the input-scripts in the command.

-b | -bin | -binary → *export binary encoded 3D mesh files for applicable formats*

Instructs the kernel to generate binary data instead of ASCII data for PLY and STL outputs.

5.3 Option-Flags to Control Output Geometric Attributes

-p | -ngon | -ngons | -poly | -polygons | -loops → *export polygonal (n-gonal) topology*

Instructs the kernel to preserve the topology of mesh and write simple and complex faces.

-t | -tess | -tesselate | -tri | -triangles → *export tessellated (triangle) topology*

Instructs the kernel to explicitly triangulate all polygonal faces and write triangle mesh.

-n | -nxyz | -normals → *export per-vertex unit-length surface-normals*

Instructs the kernel to compute derivatives for mesh that may be used for rendering and shading.

-c | -rgb | -rgba | -color | -colour | -colours → *export per-vertex red-green-blue-alpha colours*

Instructs the kernel to include colour values for mesh that may be used for rendering and shading.

5.4 Option-Flags to Control Constructive-Modelling Operations

-noop | -no_optimisation → *disable minimal-vertex topological optimisations*

Instructs the kernel to forgo (omit/skip) its mesh modifications that minimise vertex-count.

-nobo | -no_boolean_ops → *disable boolean-logic (set-theoretic/constructive) operations*

Instructs the kernel to forgo (omit/skip) the application of CSG operations during mesh assembly.

5.5 Option-Flags to Control Mesh Post-Processing Operations

-ssv | -strip_shared_vertex | -faceted → *disassociate shared-vertex to counteract gouraud-shading*

Instructs the kernel to *unweld* vertices that are shared between surface-elements to forcibly induce a faceted mesh. Note: applying this option typically bears a heavier mesh storage cost.

-mmv | -merge_manifold_vertex | -mm | -make_manifold | -manifold → *weld duplicate vertices*

Instructs the kernel to merge duplicate vertices between faces in a mesh in pursuit of a watertight 2-manifold (suitable for 3D-printing). This option is usually coupled with the resolve-T-junctions option.

-rtj | -resolve_t_junctions → *insert vertices to remove T-junctions between surface smoothing-groups*

Instructs the kernel to add edge-vertices to combat non-manifold T-junctions that can appear between the smoothing-groups of a mesh. Note: this option does not retopologise mesh-elements post insert.

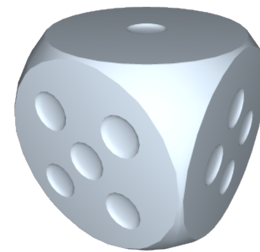
6 Example Usage

This section provides a number of practical examples that demonstrate how to invoke the kernel directly from the command-line and at runtime from within compiled programs via system-calls.

Figure 2 specifies the source-code for the scripts referred to in the command-line invocation examples.

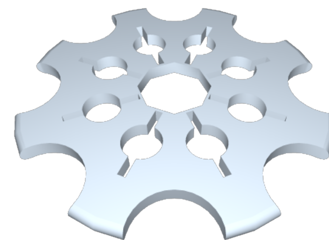
six_sided_dice.qmsh

```
1: c = cube & sphere(32,16).s(1.4);
2: d = sphere(16,8).s(0.2,0.05,0.2).rgb(0.25);
3: z1 = d.duo.ty(0.5);
4: z2 = d.lx(2,0.4).cx.ry(45).ty(0.5).rz(90);
5: z3 = d.duo.tz(0.2).ringy(3).ty(0.5).rx(90);
6: z4 = d.duo.tz(0.2).ringy(4).ty(0.5).rz(-90);
7: z5 = d.duo + d.gridxz(2,2,0.4,0.4).cxz;
8: z5.ty(0.5).rx(-90);
9: z6 = d.duo.tz(0.25).ringy(6).ty(-0.5);
10: return c - z1 - z2 - z3 - z4 - z5 - z6;
```



mechanical_gear_A.qmsh

```
1: c = cylinder(1,0.1,32);
2: t = torus(1,0.05,32,16);
3: m = c+t;
4: i = cylinder.s(0.5).tz(1.025).ringy(8);
5: o = cylinder.s(0.25).tz(0.5)
6:     .ry(180.0/8).ringy(8);
7: b = cube(0.5,1,0.05).tx(-0.5)
8:     .ry(180.0/8).ringy(8);
9: h = cylinder(0.25,1,8);
10: return m - i - o - b - h;
```



swivel_chair.qmsh

```
1: c = cylinder(0.5,1,32)
2:   - cylinder(0.45,1,32)
3:   - cylinder(0.8,1,32).rz(90).t(0,0.5,0.5)
4:   + sphere.s(0.9,0.25,0.9).ty(-0.3)
5:   + cylinder(0.45,0.05,32).ty(-0.5)
6:   - cube(0.6,0.05,1).ly(5,0.1)
7:   + cylinder(0.1,0.5,32).ty(-0.75)
8:   + cylinder(0.2,0.1,32).ty(-1)
9:   + capsule(0.05,0.5,16,8).rx(90).zz
10:     .rx(-10).tz(0.1).ty(-1).ry(36).ringy(5);
11: return c.cy;
```

Figure 2: the source-code (with assembled mesh illustrated to the right) for the scripts referred to in the command-line invocation examples - these can be entered directly into a text-editor and saved with the file-extension .qmsh - alternatively refer to the example_scripts directory in the command-line kernel distribution for pre-written versions.

6.1 Command-Line Invocation

This section provides examples of command-line invocation of the kernel. The heading of each example invocation describes the effect of the combination of option-flags employed.

Note: that these examples assume that the scripts indicated in figure 2 are located in the same directory as the kernel's executable binary file. Additionally - if the executable or script files are not in the current working directory then the path tokens in each invocation must be augmented to reflect each file's relative location to the current working directory or each file's absolute location.

Assembling a Tessellated OBJ File
→ qmsh -obj -tess six_sided_dice.qmsh

Assembling N-Gon and Triangle OBJ Files Simultaneously with Normal and RGB Data
→ qmsh -obj -ngons -tess -n -rgb six_sided_dice.qmsh

Assembling Tessellated OBJ Files for Multiple Input Scripts
→ qmsh -obj -tess six_sided_dice.qmsh mechanical_gear_A.qmsh swivel_chair.qmsh

Assembling an ASCII STL File
→ qmsh -stl six_sided_dice.qmsh

Assembling a Binary STL File
→ qmsh -stl -b six_sided_dice.qmsh

Assembling a Manifold Binary STL File Suitable for 3D-Printing
→ qmsh -stl -binary -manifold -rtj six_sided_dice.qmsh

Assembling a Tessellated Binary PLY File with Normal and RGB Data
→ qmsh -ply -tess -n -rgb six_sided_dice.qmsh

Assembling an N-Gon ASCII OFF File
→ qmsh -off -poly six_sided_dice.qmsh

Assembling N-Gon and Triangle, ASCII, OBJ, PLY and OFF Files with Normal and RGB Data
→ qmsh -obj -ply -off -ngons -tess mechanical_gear_A.qmsh

Printing Kernel Version Information to the Terminal
→ qmsh -version

Printing Kernel Command-Line Option-Flag Summary to the Terminal
→ qmsh -help

6.2 Runtime Invocation via System Calls

This section provides annotated source-code for a set of simple runtime invocation examples that demonstrate the four mains steps involved in invoking the kernel from within user-written programs via standard system-calls in the C family of programming languages (C, C++, C#, Java and Python).

At a high-level the main steps in runtime-invocation are: (firstly) to define the script statements, (secondly) to write the script statements to a file, (thirdly) to issue a system call that invokes the kernel and (fourthly) to process the generated mesh file as required by the custom application.

Note: that although the languages in which the following minimum working example programs are written differ - the underlying structure and functionality implemented by each is fundamentally the same - and is outlined in the break-down below as a pre-cursor to the source-code.

Structure of Runtime Invocation Example Source-Code (§6.2.1 - §6.2.7)

HEAD : PRE-AMBLE

- INCLUDES/USINGS/IMPORTS
- FORWARD-DECLARATIONS

BODY : MAIN-FUNCTION

- STEP-1 : DEFINE SCRIPT STATEMENTS
- STEP-2 : WRITE SCRIPT FILE
- STEP-3 : INVOKE KERNEL
- STEP-4 : PROCESS GENERATED MESH

FOOT : UTILITY-ROUTINES

- WRITE-TEXT-FILE
- EXECUTE-COMMAND

6.2.1 Runtime Invocation : C

```
// QMSH CL-KERNEL : RUNTIME INVOCATION EXAMPLE - C
//    - COMPILE : gcc qmsh_runtime_example.c
//    - EXECUTE : ./a.out

// INCLUDES
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/timeb.h>
#include <sys/types.h>
#include <sys/stat.h>

// UTILITY-TYPES
typedef char* File;
typedef char* string;
typedef int bool;
#define true 1
#define false 0

// UTILITY-FUNCTIONS : FORWARD-DECLARATION SIGNATURES
long now();
bool write_text_file(string text, File filepath);
bool execute_command(string path, int argc, char** argv);
bool file_exists(File f);
long file_length(File f);

// QMSH_PATH : THE LOCATION OF THE QMSH-KERNEL BINARY EXECUTABLE
const string QMSH_PATH = "./qmsh"; // LINUX + MAC-OSX
//const string QMSH_PATH = "./qmsh.exe"; // WINDOWS

// PROGRAM-ENTRY-POINT
int main() {

    // 1 : DEFINE SCRIPT STATEMENT(S) AS STRING
    string script = "return cube - sphere(1.25) & sphere(1.4);";

    // 2 : WRITE SCRIPT STATEMENT(S) TO FILE WITH .QMSH EXTENSION
    write_text_file(script, "rt_xmpl_c.qmsh");

    // 3 : INVOKE KERNEL TO ASSEMBLE A .OBJ TRIANGLE MESH FROM SCRIPT-FILE
    long start = now();
    char* args[] = {"-obj", "-tess", "rt_xmpl_c.qmsh"};
    execute_command(QMSH_PATH, 3, args);
    long rt = now() - start;

    // 4 : PROCESS GENERATED .OBJ MESH FILE AS REQUIRED
    File obj = "rt_xmpl_c_TESS.obj";
    if (file_exists(obj)) {
        // IN THIS BASIC EXAMPLE WE SIMPLY PRINT THE LENGTH OF THE FILE AND SOME
        // KEY STATISTICS ABOUT THE ASSEMBLY (I.E. RUNTIME, IO-RATIO) - HOWEVER
        // THERE ARE ANY NUMBER OF WAYS ONE COULD USE THE GENERATED MESH FILE
        long insz = strlen(script);
        long outsz = file_length(obj);
        printf("Input-Script-Size : %i bytes\n", (int)insz);
        printf("Output-Mesh-Size  : %i bytes\n", (int)outsz);
        printf("Execution-Runtime : %i ms\n", (int)rt);
        printf("IO-Ratio (in/out) : %f %s\n", (((int)((insz/(double)outsz)*10000))/100.0), "%");
        return 0;
    }
    else { printf("ERROR!\n"); return -1; }
}

// UTILITY-FUNCTIONS : IMPLEMENTATIONS
long now() {
    struct timeb t;
    ftime(&t);
    return t.time*1000 + t.millitm;
}
bool write_text_file(string text, File filepath) {
    FILE* f = fopen(filepath,"w");
    if (!f) return false;
    fputs(text,f);
    fclose(f);
```

```
75      return true;
76  }
77  bool execute_command(string path, int argc, char** argv) {
78      int pid = fork();
79      if (pid == -1) { return false; }
80      else if (pid == 0) {
81          int i, alen = argc;
82          char* a[1+alen+1];
83          a[0] = (char*)path;
84          for (i = 0; i < alen; i++)
85              a[i+1] = (char*)argv[i];
86          a[alen+1] = (char*)NULL;
87          execvp(a[0],a);
88      } else { wait(NULL); }
89      return true;
90  }
91  bool file_exists(File filepath) {
92      FILE* f = fopen(filepath,"r");
93      if (!f) return false;
94      fclose(f);
95      return true;
96  }
97  long file_length(File filepath) {
98      struct stat buffer;
99      if (stat(filepath, &buffer) == 0)
100         return buffer.st_size;
101     return 0;
102 }
```

Directory Contents Pre	Compile, Execute and Terminal Output	Directory Contents Post
qmsh qmsh_runtime_example.c	~$ gcc qmsh_runtime_example.c ~$ ./a.out Input-Script-Size : 41 bytes Output-Mesh-Size : 42143 bytes Execution-Runtime : 162 ms IO-Ratio (in/out) : 0.090000 %	a.out qmsh qmsh_runtime_example.c rt_xmpl_c.qmsh rt_xmpl_c_TESS.obj

Figure 3: a simple C program that demonstrates the four main steps in invoking the QMSH kernel at runtime.

6.2.2 Runtime Invocation : C++

```
// QMSH CL-KERNEL : RUNTIME INVOCATION EXAMPLE - C++
//   - COMPILE : g++ qmsh_runtime_example.cpp
//   - EXECUTE : ./a.out

// INCLUDES
#include <string>   // FOR STRING TYPE
#include <vector>   // FOR VECTOR TYPE
#include <iostream> // FOR CONSOLE-OUT PRINT STATEMENTS
#include <fstream>  // FOR INPUT AND OUTPUT FILE STREAMS
#include <chrono>   // FOR SYSTEM CLOCK
#include <unistd.h> // FOR PROCESS MANAGEMENT (FORK, EXEC..)

// USINGS
using namespace std;
using namespace std::chrono;

// UTILITY-TYPE
class File {
public:
    string path;
    File() { }
    File(const string &p) : path(p) { }
    ~File() { }
    long size() { return ifstream(path, ios::binary | ios::ate).tellg(); }
    bool exists() { return ifstream(path).good(); }
};

// UTILITY-FUNCTIONS : FORWARD-DECLARATION SIGNATURES
long now();
bool write_text_file(const string &text, const File &file);
bool execute_command(const string &path, const vector<string> &args);
bool execute_command(const vector<string> &path_args);

// QMSH_PATH : THE LOCATION OF THE QMSH-KERNEL BINARY EXECUTABLE
const string QMSH_PATH = "./qmsh"; // LINUX + MAC-OSX
//  const string QMSH_PATH = "./qmsh.exe"; // WINDOWS

// PROGRAM-ENTRY-POINT
int main() {

    // 1 : DEFINE SCRIPT STATEMENT(S) AS STRING
    string script = "return cube - sphere(1.25) & sphere(1.4);";

    // 2 : WRITE SCRIPT STATEMENT(S) TO FILE WITH .QMSH EXTENSION
    write_text_file(script, File("rt_xmpl_cpp.qmsh"));

    // 3 : INVOKE KERNEL TO ASSEMBLE A .OBJ TRIANGLE MESH FROM SCRIPT-FILE
    long start = now();
    execute_command(QMSH_PATH, { "-obj", "-tess", "rt_xmpl_cpp.qmsh" });
    //execute_command({ QMSH_PATH, "-obj", "-tess", "rt_xmpl_cpp.qmsh" });
    long rt = now() - start;

    // 4 : PROCESS GENERATED .OBJ MESH FILE AS REQUIRED
    File obj("rt_xmpl_cpp_TESS.obj");
    if (obj.exists()) {
        // IN THIS BASIC EXAMPLE WE SIMPLY PRINT THE LENGTH OF THE FILE AND SOME
        // KEY STATISTICS ABOUT THE ASSEMBLY (I.E. RUNTIME, IO-RATIO) - HOWEVER
        // THERE ARE ANY NUMBER OF WAYS ONE COULD USE THE GENERATED MESH FILE
        long insz = script.size();
        long outsz = obj.size();
        cout << "Input-Script-Size  : " << insz << " bytes" << endl;
        cout << "Output-Mesh-Size   : " << outsz << " bytes" << endl;
        cout << "Execution-Runtime  : " << rt << " ms" << endl;
        cout << "IO-Ratio (in/out)  : " << (((int)((insz/(double)outsz)*10000))/100.0) << " %" << endl;
        return 0;
    }
    else { cout << "ERROR!" << endl; return -1; }
}

// UTILITY-FUNCTIONS : IMPLEMENTATIONS
long now() {
    milliseconds t = duration_cast<milliseconds>(system_clock::now().time_since_epoch());
    return t.count();
}
```

```
75  bool write_text_file(const string &text, const File &file) {
76      ofstream out;
77      out.open(file.path);
78      if (out.is_open()) {
79          out << text;
80          out.close();
81          return true;
82      } return false;
83  }
84  bool execute_command(const string &path, const vector<string> &args) {
85      int pid = fork();
86      if (pid == -1) { return false; }
87      else if (pid == 0) {
88          int i, alen = args.size();
89          char *a[1+alen+1];
90          a[0] = (char*)path.c_str();
91          for (i = 0; i < alen; i++)
92              a[i+1] = (char*)args[i].c_str();
93          a[alen+1] = (char*)NULL;
94          execvp(a[0],a);
95      } else { wait(NULL); }
96      return true;
97  }
98  // ALTERNATIVE INTERFACE
99  bool execute_command(const vector<string> &path_args) {
100     if (path_args.size() < 1) return false;
101     int pid = fork();
102     if (pid == -1) { return false; }
103     else if (pid == 0) {
104         int i, alen = path_args.size();
105         char *a[alen+1];
106         for (i = 0; i < alen; i++)
107             a[i] = (char*)path_args[i].c_str();
108         a[alen] = (char*)NULL;
109         execvp(a[0],a);
110     } else { wait(NULL); }
111     return true;
112 }
```

Directory Contents Pre	Compile, Execute and Terminal Output	Directory Contents Post

`| qmsh`
`| qmsh_runtime_example.cpp`
	`| ~$ g++ qmsh_runtime_example.cpp`
	`| ~$ ./a.out`
	`| Input-Script-Size : 41 bytes`
	`| Output-Mesh-Size : 42145 bytes`
	`| Execution-Runtime : 161 ms`
	`| IO-Ratio (in/out) : 0.09 %`
		`| a.out`
		`| qmsh`
		`| qmsh_runtime_example.cpp`
		`| rt_xmpl_cpp.qmsh`
		`| rt_xmpl_cpp_TESS.obj`

Figure 4: a simple C++ program that demonstrates the four main steps in invoking the QMSH kernel at runtime.

6.2.3 Runtime Invocation : C#

```
// QMSH CL-KERNEL : RUNTIME INVOCATION EXAMPLE - C#
//    - COMPILE : mcs qmsh_runtime_example.cs
//    - EXECUTE : mono qmsh_runtime_example.exe // LINUX + OSX
//    - EXECUTE : qmsh_runtime_example.exe      // WINDOWS

// USINGS
using System;
using System.IO;
using System.Text;
using System.Diagnostics;

public class qmsh_runtime_example {

    // QMSH_PATH : THE LOCATION OF THE QMSH-KERNEL BINARY EXECUTABLE
    private static readonly string QMSH_PATH = "./qmsh"; // LINUX + MAC-OSX
    //  private static readonly String QMSH_PATH = "./qmsh.exe"; // WINDOWS

    // PROGRAM-ENTRY-POINT
    public static void Main() {

        // 1 : DEFINE SCRIPT STATEMENT(S) AS STRING
        string script = "return cube - sphere(1.25) & sphere(1.4);";

        // 2 : WRITE SCRIPT STATEMENT(S) TO FILE WITH .QMSH EXTENSION
        write_text_file(script, new File("rt_xmpl_cs.qmsh"));

        // 3 : INVOKE KERNEL TO ASSEMBLE A .OBJ TRIANGLE MESH FROM SCRIPT-FILE
        long start = now();
        execute_command(QMSH_PATH, "-obj", "-tess", "rt_xmpl_cs.qmsh");
        //execute_command(QMSH_PATH, "-obj -tess rt_xmpl_cs.qmsh");
        long rt = now() - start;

        // 4 : PROCESS GENERATED .OBJ MESH FILE AS REQUIRED
        File obj = new File("rt_xmpl_cs_TESS.obj");
        if (obj.exists()) {
            // IN THIS BASIC EXAMPLE WE SIMPLY PRINT THE LENGTH OF THE FILE AND SOME
            // KEY STATISTICS ABOUT THE ASSEMBLY (I.E. RUNTIME, IO-RATIO) - HOWEVER
            // THERE ARE ANY NUMBER OF WAYS ONE COULD USE THE GENERATED MESH FILE
            long insz = script.Length;
            long outsz = obj.Length;
            print("Input-Script-Size  : " + insz + " bytes");
            print("Output-Mesh-Size   : " + outsz + " bytes");
            print("Execution-Runtime  : " + rt + " ms");
            print("IO-Ratio (in/out)  : " + ((int)((insz/(double)outsz)*10000))/100.0 + " %");
        } else { print("ERROR!"); }
        return;
    }

    // UTILITY-FUNCTIONS
    private static void print(string s) { Console.WriteLine(s); }
    private static long now() {
        return (long)(DateTime.Now - DateTime.MinValue).TotalMilliseconds;
    }
    private static bool write_text_file(string text, File file) {
        try {
            using (StreamWriter writer = new StreamWriter(new FileStream(file.path, FileMode.Create))) {
                writer.Write(text);
            } return true;
        } catch (Exception ex) { print(ex.StackTrace); return false; }
    }
    private static bool execute_command(string path, params string[] args) {
        try {
            int i, alen = args.Length;
            StringBuilder sb = new StringBuilder();
            for (i = 0; i < alen; i++) {
                sb.Append(args[i]);
                if (i < alen-1) sb.Append(" ");
            }
            String argstr = sb.ToString();
            Process proc = Process.Start(path, argstr);
            proc.WaitForExit();
            return true;
        } catch (Exception ex) { print(ex.StackTrace); return false; }
    }
```

```
75      // ALTERNATIVE INTERFACE
76      private static bool execute_command(string path, string args) {
77          try {
78              Process proc = Process.Start(path, args);
79              proc.WaitForExit();
80              return true;
81          } catch (Exception ex) { print(ex.StackTrace); return false; }
82      }
83  }
84
85  // UTILITY-TYPE
86  class File {
87      public string path;
88      public File() { }
89      public File(string p) { path = p; }
90      public bool exists() { return System.IO.File.Exists(path); }
91      public long Length { get { return new FileInfo(path).Length; } }
92  }
```

Directory Contents Pre	Compile, Execute and Terminal Output	Directory Contents Post

`| qmsh`
`| qmsh_runtime_example.cs`
	`| ~$ mcs qmsh_runtime_example.cs`
	`| ~$ mono qmsh_runtime_example.exe`
	`| Input-Script-Size : 41 bytes`
	`| Output-Mesh-Size  : 42144 bytes`
	`| Execution-Runtime : 223 ms`
	`| IO-Ratio (in/out) : 0.09 %`
		`| qmsh`
		`| qmsh_runtime_example.cs`
		`| qmsh_runtime_example.exe`
		`| rt_xmpl_cs.qmsh`
		`| rt_xmpl_cs_TESS.obj`

Figure 5: a simple C# program that demonstrates the four main steps in invoking the QMSH kernel at runtime.

6.2.4 Runtime Invocation : Groovy

```
// QMSH CL-KERNEL : RUNTIME INVOCATION EXAMPLE - GROOVY
//    - EXECUTE : groovy qmsh_runtime_example.groovy

class qmsh_runtime_example {

    // QMSH_PATH : THE LOCATION OF THE QMSH-KERNEL BINARY EXECUTABLE
    static final String QMSH_PATH = "../../qmsh"; // LINUX + MAC-OSX
    // private static final String QMSH_PATH = "../../qmsh.exe"; // WINDOWS

    // PROGRAM-ENTRY-POINT
    public static void main(String[] args) {
        // 1 : DEFINE SCRIPT STATEMENT(S) AS STRING
        String script = "return cube - sphere(1.25) & sphere(1.4);";
        // 2 : WRITE SCRIPT STATEMENT(S) TO FILE WITH .QMSH EXTENSION
        write_text_file(script, new File("rt_xmpl_groovy.qmsh"));
        // 3 : INVOKE KERNEL TO ASSEMBLE A .OBJ TRIANGLE MESH FROM SCRIPT-FILE
        long start = now();
        execute_command(QMSH_PATH, "-obj", "-tess", "rt_xmpl_groovy.qmsh");
        long rt = now() - start;
        // 4 : PROCESS GENERATED .OBJ MESH FILE AS REQUIRED
        File obj = new File("rt_xmpl_groovy_TESS.obj");
        if (obj.exists()) {
            // IN THIS BASIC EXAMPLE WE SIMPLY PRINT THE LENGTH OF THE FILE AND SOME
            // KEY STATISTICS ABOUT THE ASSEMBLY (I.E. RUNTIME, IO-RATIO) - HOWEVER
            // THERE ARE ANY NUMBER OF WAYS ONE COULD USE THE GENERATED MESH FILE
            long insz = script.length();
            long outsz = obj.length();
            println("Input-Script-Size : " + insz + " bytes");
            println("Output-Mesh-Size  : " + outsz + " bytes");
            println("Execution-Runtime : " + rt + " ms");
            println("IO-Ratio (in/out) : " + ((int)((insz/(double)outsz)*10000))/100.0 + " %");
        } else { println("ERROR!"); }
        return;
    }
    // UTILITY-FUNCTIONS
    static long now() { return System.currentTimeMillis(); }
    static boolean write_text_file(String text, File file) {
        if (text == null || file == null) return false;
        try {
            FileOutputStream out = new FileOutputStream(file, false);
            out.write(text.getBytes());
            out.close();
            return true;
        } catch (Exception e) { e.printStackTrace(); return false; }
    }
    static boolean execute_command(String... path_args) {
        if (path_args == null || path_args.length < 1) return false;
        try {
            Runtime rt = Runtime.getRuntime();
            Process p = rt.exec(path_args);
            p.waitFor();
            return true;
        } catch (Exception ex) { ex.printStackTrace(); return false; }
    }
}
```

Directory Contents Pre	Compile, Execute and Terminal Output	Directory Contents Post

`| qmsh`
`| qmsh_runtime_example.groovy`
	`| ~$ groovy qmsh_runtime_example.groovy`
	`| Input-Script-Size : 41 bytes`
	`| Output-Mesh-Size  : 42148 bytes`
	`| Execution-Runtime : --- ms`
	`| IO-Ratio (in/out) : 0.09 %`
		`| qmsh`
		`| qmsh_runtime_example.groovy`
		`| rt_xmpl_groovy.qmsh`
		`| rt_xmpl_groovy_TESS.obj`

Figure 6: a simple Groovy program that demonstrates the four main steps in invoking the QMSH kernel at runtime.

6.2.5 Runtime Invocation : Java

```java
// QMSH CL-KERNEL : RUNTIME INVOCATION EXAMPLE - JAVA
//    - COMPILE : javac qmsh_runtime_example.java
//    - EXECUTE : java qmsh_runtime_example

// IMPORTS
import java.io.File;
import java.io.FileOutputStream;

public class qmsh_runtime_example {

    // QMSH_PATH : THE LOCATION OF THE QMSH-KERNEL BINARY EXECUTABLE
    private static final String QMSH_PATH = "./qmsh"; // LINUX + MAC-OSX
    //  private static final String QMSH_PATH = "./qmsh.exe"; // WINDOWS

    // PROGRAM-ENTRY-POINT
    public static void main(String[] args) {
        // 1 : DEFINE SCRIPT STATEMENT(S) AS STRING
        String script = "return cube - sphere(1.25) & sphere(1.4);";
        // 2 : WRITE SCRIPT STATEMENT(S) TO FILE WITH .QMSH EXTENSION
        write_text_file(script, new File("rt_xmpl_java.qmsh"));
        // 3 : INVOKE KERNEL TO ASSEMBLE A .OBJ TRIANGLE MESH FROM SCRIPT-FILE
        long start = now();
        execute_command(QMSH_PATH, "-obj", "-tess", "rt_xmpl_java.qmsh");
        long rt = now() - start;
        // 4 : PROCESS GENERATED .OBJ MESH FILE AS REQUIRED
        File obj = new File("rt_xmpl_java_TESS.obj");
        if (obj.exists()) {
            // IN THIS BASIC EXAMPLE WE SIMPLY PRINT THE LENGTH OF THE FILE AND SOME
            // KEY STATISTICS ABOUT THE ASSEMBLY (I.E. RUNTIME, IO-RATIO) - HOWEVER
            // THERE ARE ANY NUMBER OF WAYS ONE COULD USE THE GENERATED MESH FILE
            long insz = script.length();
            long outsz = obj.length();
            print("Input-Script-Size : " + insz + " bytes");
            print("Output-Mesh-Size  : " + outsz + " bytes");
            print("Execution-Runtime : " + rt + " ms");
            print("IO-Ratio (in/out) : " + ((int)((insz/(double)outsz)*10000))/100.0 + " %");
        } else { print("ERROR!"); }
        return;
    }
    // UTILITY-FUNCTIONS
    private static long now() { return System.currentTimeMillis(); }
    private static void print(String s) { System.out.println(s); }
    private static boolean write_text_file(String text, File file) {
        if (text == null || file == null) return false;
        try {
            FileOutputStream out = new FileOutputStream(file, false);
            out.write(text.getBytes());
            out.close();
            return true;
        } catch (Exception e) { e.printStackTrace(); return false; }
    }
    private static boolean execute_command(String... path_args) {
        if (path_args == null || path_args.length < 1) return false;
        try {
            Runtime rt = Runtime.getRuntime();
            Process p = rt.exec(path_args);
            p.waitFor();
            return true;
        } catch (Exception ex) { ex.printStackTrace(); return false; }
    }
}
```

Directory Contents Pre

```
| qmsh
| qmsh_runtime_example.java
```

Compile, Execute and Terminal Output

```
| ~$ javac qmsh_runtime_example.java
| ~$ java qmsh_runtime_example
| Input-Script-Size : 41 bytes
| Output-Mesh-Size  : 42146 bytes
| Execution-Runtime : 182 ms
| IO-Ratio (in/out) : 0.09 %
```

Directory Contents Post

```
| qmsh
| qmsh_runtime_example.class
| qmsh_runtime_example.java
| rt_xmpl_java.qmsh
| rt_xmpl_java_TESS.obj
```

Figure 7: a simple Java program that demonstrates the four main steps in invoking the QMSH kernel at runtime.

6.2.6 Runtime Invocation : Objective-C

```
// QMSH CL-KERNEL : RUNTIME INVOCATION EXAMPLE - OBJECTIVE-C
//    - COMPILE : clang -x objective-c qmsh_runtime_example.m
//    - EXECUTE : ./a.out

// INCLUDES
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/timeb.h>
#include <sys/types.h>
#include <sys/stat.h>

// UTILITY-TYPES
typedef char* File;
typedef char* string;
typedef int bool;
#define true 1
#define false 0

// UTILITY-FUNCTIONS : FORWARD-DECLARATION SIGNATURES
long now();
bool write_text_file(string text, File filepath);
bool execute_command(string path, int argc, char** argv);
bool file_exists(File f);
long file_length(File f);

// QMSH_PATH : THE LOCATION OF THE QMSH-KERNEL BINARY EXECUTABLE
const string QMSH_PATH = "./qmsh"; // LINUX + MAC-OSX
//const string QMSH_PATH = "./qmsh.exe"; // WINDOWS

// PROGRAM-ENTRY-POINT
int main() {

    // 1 : DEFINE SCRIPT STATEMENT(S) AS STRING
    string script = "return cube - sphere(1.25) & sphere(1.4);";

    // 2 : WRITE SCRIPT STATEMENT(S) TO FILE WITH .QMSH EXTENSION
    write_text_file(script, "rt_xmpl_objc.qmsh");

    // 3 : INVOKE KERNEL TO ASSEMBLE A .OBJ TRIANGLE MESH FROM SCRIPT-FILE
    long start = now();
    char* args[] = {"-obj", "-tess", "rt_xmpl_objc.qmsh"};
    execute_command(QMSH_PATH, 3, args);
    long rt = now() - start;

    // 4 : PROCESS GENERATED .OBJ MESH FILE AS REQUIRED
    File obj = "rt_xmpl_objc_TESS.obj";
    if (file_exists(obj)) {
        // IN THIS BASIC EXAMPLE WE SIMPLY PRINT THE LENGTH OF THE FILE AND SOME
        // KEY STATISTICS ABOUT THE ASSEMBLY (I.E. RUNTIME, IO-RATIO) - HOWEVER
        // THERE ARE ANY NUMBER OF WAYS ONE COULD USE THE GENERATED MESH FILE
        long insz = strlen(script);
        long outsz = file_length(obj);
        printf("Input-Script-Size  : %i bytes\n", (int)insz);
        printf("Output-Mesh-Size   : %i bytes\n", (int)outsz);
        printf("Execution-Runtime  : %i ms\n", (int)rt);
        printf("IO-Ratio (in/out)  : %f %s\n", (((int)((insz/(double)outsz)*10000))/100.0), "%");
        return 0;
    }
    else { printf("ERROR!\n"); return -1; }
}

// UTILITY-FUNCTIONS : IMPLEMENTATIONS
long now() {
    struct timeb t;
    ftime(&t);
    return t.time*1000 + t.millitm;
}
bool write_text_file(string text, File filepath) {
    FILE* f = fopen(filepath,"w");
    if (!f) return false;
    fputs(text,f);
    fclose(f);
```

```
75      return true;
76  }
77  bool execute_command(string path, int argc, char** argv) {
78      int pid = fork();
79      if (pid == -1) { return false; }
80      else if (pid == 0) {
81          int i, alen = argc;
82          char* a[1+alen+1];
83          a[0] = (char*)path;
84          for (i = 0; i < alen; i++)
85              a[i+1] = (char*)argv[i];
86          a[alen+1] = (char*)NULL;
87          execvp(a[0],a);
88      } else { wait(NULL); }
89      return true;
90  }
91  bool file_exists(File filepath) {
92      FILE* f = fopen(filepath,"r");
93      if (!f) return false;
94      fclose(f);
95      return true;
96  }
97  long file_length(File filepath) {
98      struct stat buffer;
99      if (stat(filepath, &buffer) == 0)
100         return buffer.st_size;
101     return 0;
102 }
```

Directory Contents Pre	Compile, Execute and Terminal Output	Directory Contents Post
| qmsh | qmsh_runtime_example.m	| ~$ clang -x objective-c qmsh_runtime_example.m | ~$ ./a.out | Input-Script-Size : 41 bytes | Output-Mesh-Size : 42146 bytes | Execution-Runtime : --- ms | IO-Ratio (in/out) : 0.090000 %	| a.out | qmsh | qmsh_runtime_example.m | rt_xmpl_objc.qmsh | rt_xmpl_objc_TESS.obj

Figure 8: a simple Objective-C program demonstrating the four main steps in invoking the QMSH kernel at runtime.

6.2.7 Runtime Invocation : Python

```
# QMSH CL-KERNEL : RUNTIME INVOCATION EXAMPLE - PYTHON
#    - EXECUTE : python qmsh_runtime_example.py

# IMPORTS
import time
import os.path
import subprocess

# QMSH_PATH : THE LOCATION OF THE QMSH-KERNEL BINARY EXECUTABLE
QMSH_PATH = "./qmsh"; # LINUX + MAC-OSX
# QMSH_PATH = "./qmsh.exe"; # WINDOWS

# UTILITY-ROUTINES
def now(): return int(round(time.time() * 1000));
def write_text_file(text,filepath):
    f = open(filepath,"w");
    f.write(text);
    f.close();
def execute_command(*path_args): subprocess.call(list(path_args));
def file_exists(filepath): return os.path.isfile(filepath);
def file_length(filepath): return os.stat(filepath).st_size;

# PROGRAM-ENTRY-POINT
def main():
    # 1 : DEFINE SCRIPT STATEMENT(S) AS STRING
    script = "return cube - sphere(1.25) & sphere(1.4);";
    # 2 : WRITE SCRIPT STATEMENT(S) TO FILE WITH .QMSH EXTENSION
    write_text_file(script, "rt_xmpl_python.qmsh");
    # 3 : INVOKE KERNEL TO ASSEMBLE A .OBJ TRIANGLE MESH FROM SCRIPT-FILE
    start = now();
    execute_command(QMSH_PATH, "-obj", "-tess", "rt_xmpl_python.qmsh");
    rt = now() - start;
    # 4 : PROCESS GENERATED .OBJ MESH FILE AS REQUIRED
    obj = "rt_xmpl_python_TESS.obj";
    if (file_exists(obj)):
        # IN THIS BASIC EXAMPLE WE SIMPLY PRINT THE LENGTH OF THE FILE AND SOME
        # KEY STATISTICS ABOUT THE ASSEMBLY (I.E. RUNTIME, IO-RATIO) - HOWEVER
        # THERE ARE ANY NUMBER OF WAYS ONE COULD USE THE GENERATED MESH FILE
        insz = len(script);
        outsz = file_length(obj);
        print("Input-Script-Size : " + str(insz) + " bytes");
        print("Output-Mesh-Size  : " + str(outsz) + " bytes");
        print("Execution-Runtime : " + str(rt) + " ms");
        print("IO-Ratio (in/out) : " + str(int(((insz/float(outsz))*10000))/100.0) + " %");
    else:
        print("ERROR!");
    return;
# RUN
main();
```

Directory Contents Pre	Compile, Execute and Terminal Output	Directory Contents Post
| qmsh | qmsh_runtime_example.py	| ~$ python qmsh_runtime_example.py | Input-Script-Size : 41 bytes | Output-Mesh-Size : 42148 bytes | Execution-Runtime : 161 ms | IO-Ratio (in/out) : 0.09 %	| qmsh | qmsh_runtime_example.py | rt_xmpl_py.qmsh | rt_xmpl_py_TESS.obj

Figure 9: a simple Python program that demonstrates the four main steps in invoking the QMSH kernel at runtime.

7 Additional Notes

Output File Naming Convention

The mesh interchange files generated by the kernel are named according to the following convention.

The body of each output filename is drawn directly from the filename of its defining input script - with the file extension changed to reflect the target format. For interchange formats that support both triangle and polygonal representations (i.e. OBJ, OFF and PLY) an additional topological indicator token is appended to each output file's name to denote the type (or class) of reader necessary to open each file.

The topological post-fix take the form *_TESS* (to indicate tessellated triangle elements) or *_NGON* (to indicate polygonal elements). To help clarify: invocation of the kernel command:

```
./qmsh -obj -tess -ngon -c -n test/a.qmsh
```

would yield a pair of OBJ mesh files named (respectively): *test/a_TESS.obj* and *test/a_NGON.obj*. Whereas invocation of the kernel command:

```
./qmsh -stl test/a.qmsh
```

would yield a single STL mesh file named *test/a.stl*.

This convention aims to remove ambiguity and ensure that it is trivial (at a glance) to determine the type of mesh stored by a generated interchange file - in a format agnostic manner.

Trouble-Shooting: Resolving Common Sources of Script Error

Missing Return Statement: resulting from scripting oversight. Resolution: add a suitable return statement. Remember: quick-mesh scripts require a return statement to be considered syntactically valid.

Failure to Terminate Statement: resulting from omitted (typically forgotten) semi-colons at the end of a line. Resolution: make sure that each script statement is terminated by a semi-colon.

Invalid Use of Commas Instead of Dots: (and vice-versa) resulting from input typing errors. Resolution: ensure that these punctuation symbols are used appropriately in a script. Remember: that in the quick-mesh grammar dots (periods) are used for decimal numbers and post-fix function notation - whilst commas are used to separate elements in argument lists and arrays.

Invalid Input Argument(s) to Built-In Function: resulting from unfamiliarity with function signatures. Resolution: check and then revise the input arguments in the offending function invocation.

Unbalanced Operative Scope: resulting from either missing brackets or braces. Resolution: add (or remove) brackets and braces to ensure that each operative scope that is opened is closed appropriately.

7.1 Performance Considerations

Finally this section outlines some performance considerations to bear in mind when wielding the kernel.

Computational Efficiency: Execution Runtime
A number of factors govern the total execution time for a quick-mesh script. The dominating factor is the complexity of the geometric arrangement encoded by a script. In particular this means that the simpler the expression of geometric form encapsulated by a script the less time taken to execute the script by the kernel. In practical terms the greatest source of complexity in mesh assembly (in the quick-mesh grammar) is the application of CSG boolean-logic operations. For although (conceptually) the notion of geometric-arithmetic is delightfully simple to intuit - in computational terms - the implementation of minimal-vertex polyhedral solid-modelling operations bears a far heavier polynomial cost - relative to many of the kernel's O(n) (linear-complexity) built-in functions. In this regard - the key consideration to bear in mind is that the runtime of a script will largely be governed by the number of and complexity inherent to its CSG operations. The unfortunate aspect is that the complexity of a polyhedral CSG operation is non-trivial to determine exactly (other than by execution). However, despite this - a number of heuristics (rules-of-thumb) are applicable to the growth in execution time exhibited by the kernel's boolean-operations. Firstly: the greater the number of vertices in each operand - the greater the runtime. Secondly: the greater the number of successive boolean-ops applied to an operand, the greater the runtime. Thirdly: the greater the number of element interactions (between the operands in a boolean-op) that induce non-simplex polygons the greater the runtime.

Computational Efficiency: Execution Memory Requirements
As the total execution time - the runtime memory requirements for assembly of a mesh are largely determined by the memory requirements of the boolean-operations in its defining script. Hence memory-use-wise the same considerations apply to the growth in memory-use exhibited by the kernel.

Computational Efficiency: Simultaneous-Processing via Multi-Core Parallelism
The kernel executes as a single-core process - meaning that it is amenable to simple multi-core parallelism - by running each instance as a separate process. For example this can be handled trivially by issuing system-calls from multiple threads in a high-level general-purpose scripting or systems-programming language. Examples of languages suitable for coordinating this form of parallelism include: Java, Python and C-Sharp. Note: that when that when running the kernel in this manner - it is the responsibility of the invoker to manage any load-balancing deemed necessary. In particular one should aim to distribute the mesh-assembly work-load evenly across the kernel instances. Additionally strive to keep the number of active kernel instances running at any one time below the number of available cores on the executing device or machine. For the fewer active kernel instances the less scheduling and context-switching the underlying operating system has to do to accommodate them simultaneously. If too many kernel instances are running - undesirable thrashing-styled effects can result in which the operating-system must cane CPU-time simply to coordinate switching between the glut of processes.

Geometric Stability: Limitations of Fixed-Precision Floating-Point Arithmetic
The kernel uses fixed-precision arithmetic as opposed to arbitrary-precision arithmetic to coordinate its procedural geometric operations that involve floating-decimal-point calculations. Given that a finite number of bits are used to store each decimal number the kernel's floating-point calculations are subject to loss of information which may cause subtle numerical errors (overflow, underflow, rounding error). Often this limitation is imperceptible in the mesh generated by the kernel, however under certain conditions - and for certain arrangements - this can result in mesh which fail to preserve geometric features beyond the grain of accuracy afforded (attainable via) fixed-precision floating-point arithmetic. In extreme circumstances this can lead to cumulative errors that result in topologically invalid geometric entities being propagated through the assembly process - which has the potential to interfere with minimal-vertex

optimisation and even prevent convergence. Whilst the kernel employs a myriad of complementary numerically robust predicates to guard against such occurrences (and uses integer-arithmetic whenever applicable) - its underlying low runtime memory-use architecture means it is beyond the kernel's scope to guarantee the topological correctness of the polyhedra it generates. This undesirable aspect of the kernel represents a practical trade-off of absolute-correctness in favour of computational-efficiency.

Geometric Stability: Shading Artefacts Resulting from Mesh Attribute Interpolation
The kernel's minimal-vertex optimisations generate mesh that prioritise brevity and compactness over triangle quality. Practically - this means that during the tessellation of optimised N-gonal elements - skinny sliver triangles may be produced. Such elements have the potential to cause perceptible shading artefacts in object-order rendering algorithms and visualisation systems - and manifest most notably as discontinuities in shading at the clipped edges of smooth-group regions in a surface mesh. Note: these artefacts are less prevalent (yet still present) in image-order rendering techniques - as the product of per-pixel or sub-pixel over per-vertex attribute interpolation. In order to remedy such occurrences - retopologise the offending portions of the entity in question. This can be achieved by directly modifying the input script - for example by altering the discretisation-steps argument(s) used to generate native geometry. Alternatively this may also be resolved externally - by feeding a kernel generated output interchange file into a third-party high-quality mesh-generator program. In such instances useful search terms include: FEM-mesh-generator, high-quality-mesh-generator and Delaunay-mesh-generator.

Input-Output: Mesh Storage Requirements
The considerations and rules-of-thumb applicable to the storage requirements for the mesh interchange files generated by the kernel are as follows. Binary interchange files almost always require less space to store relative to their analogous ASCII interchange files in the same format. Interchange files containing mixed-topology (N-gonal) mesh elements typically require slightly less space to store than their tessellated counterparts - due to there being fewer indices to store. However - be aware that some mesh readers, loaders and importers may not directly support concave or complex face handling. Generally speaking - the greater the number of vertices in a mesh - the greater the space required to store the mesh. For binary interchange this growth is near-linear (subject to the exception of format specific header blocks) - however for ASCII interchange this growth is sub-linear (i.e. linear and then some) - due to the additional character overhead associated with printing increasingly large index values. This means that whilst it is possible to accurately estimate the storage requirements for binary interchange files if the number of vertices are known, for ASCII interchange files it is recommended to use at least a 2-factor overestimate - i.e. budget for the storage requirements of an ASCII file to be roughly twice that of a binary equivalent. Practically - this implies that by partitioning a large multi-component mesh into separate scripts - one can reduce the overhead associated with storing it in ASCII form. Note: the recommended partitioning threshold is 2^{16} vertices (less than 65,536) - as the 65K limit is common to pre-existing APIs, engines and DCC systems (such as OpenGL-ES and Unity-3D).

Input-Output: Mesh Representation Compatibility
The kernel's N-gonal mesh may or may not be trivially tessellatable by applications such as meshlab depending on the presence of complex-polygons. This means that if the polygons that define an output mesh contain holes and islands - typical mesh-loaders will fail to load them correctly. In such cases - use triangles for the surface representation and the N-gon variant for poly-loop edge representation.

***

So ends this guide. I hope you enjoy using the command-line kernel as much as I enjoy making it.