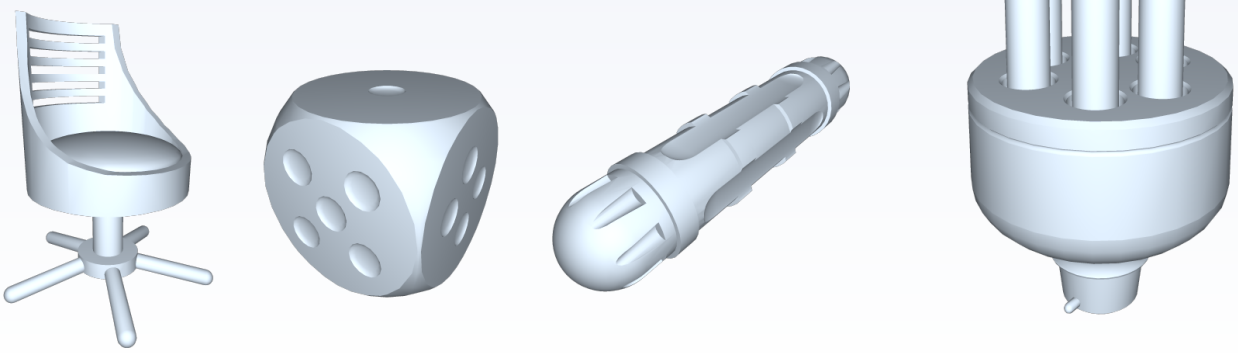
QMSH

Procedural Geometry Kernel



QMSH

Android Mobile-Editor

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 Mobile-Editor is a stand-alone (sand-boxed) IDE (integrated development environment) that provides an end-to-end work-flow for scripted procedural polyhedral mesh creation. The app aims to be intuitive, concise and accessible to all. It includes components for script-editing, parsing, mesh-assembly, rendering, visualisation, documentation and IO. This document provides a guide to the use of the app on Android devices for the purpose of creating 3D geometry on-the-go.

Table of Contents

1 Introduction

This short guide provides practical instructions on the use of the Quick-Mesh Mobile-Editor for Android devices. The primary aim of this guide is to help and direct new users by explaining the behaviour of the tools and menus and the main steps to follow to create 3D models with the app.

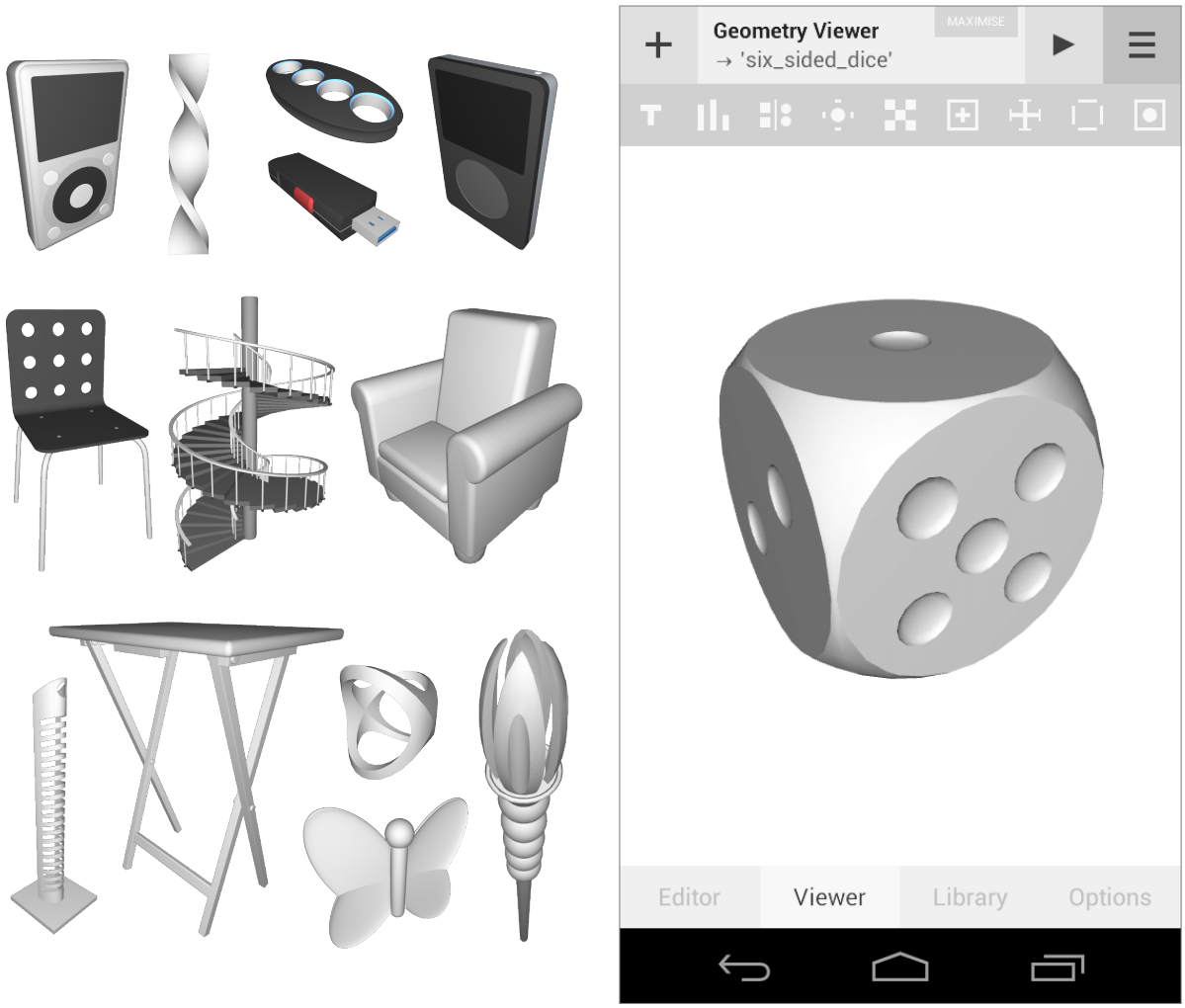


Figure 1: screenshot of the Quick-Mesh Mobile-Editor running on Android (right) alongside an assortment of example polyhedral entities (rendered left) - for which each mesh is defined and constructed with the Mobile-Editor

1.1 Pre-Requisites

*** Items Required to Follow this Guide and Model 3D Objects using the Mobile-Editor ***

- Hardware : Touchscreen Mobile-Device (i.e. a smart-phone or tablet)
  minimum system requirements:
  → 1GB RAM, 4GB ROM, 2x CPU-Cores, 360-dp+ Display
  recommended system requirements:
  → 2GB-4GB RAM, 8GB-16GB ROM, 4x-8x CPU-Cores, 360-dp+ Display
- Operating-System : Android 4.0+ (minimum API-Version: 14)
- Software : Quick-Mesh Mobile-Editor (a self-contained Android application)

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 Mobile-Editor Effectively

- Elementary Understanding of Geometric Modelling Operations and Techniques
  e.g. using Euclidean transformations, types of primitive and polygon mesh attributes.
- 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.

2 Inputs & Outputs

The Quick-Mesh Mobile-Editor accepts scripts as input and generates geometric models (polygon-mesh) as output. Models can be exported as OBJ, OFF, PLY or STL files for use in other applications.

Quick-Mesh scripts are human-readable plain text files stored with the file extension .qmsh.

Figure 2 depicts some simple models written in the Quick-Mesh scripting language to help clarify.

cube_minus_sphere.qmsh

```
1: return cube - sphere.s(1.25);
```

cube_octahedron.qmsh

```
1: o = octahedron;
2: return cube - o.duo.s(1.5) & o.s(2.0);
```

dual_band_ring.qmsh

```
1: return torus(0.01,0.001,64,32).sy(3)
2:     - torus(0.011,0.00025,64,16).ly(2,0.00125).cy;
```

t_junction_pipe.qmsh

```
1: return cylinder(0.25,0.5,32).zy
2:     + cylinder(0.25,1,32).rz(90)
3:     - cylinder(0.2,0.5,32).zy
4:     - cylinder(0.2,1,32).rz(90);
```

mechanical_part.qmsh

```
1: m = cylinder.s(0.5,3,0.5);
2: return cube + m - m.duo.rz(90) - m.duo.rx(90);
```

wooden_dinner_chair.qmsh

```
1: return cube(1.05,0.1,1.05)
2:     + cube(0.125,1,0.125).nzy.grid(2,1,2,0.9,0,0.9).cxz
3:     + cube(0.8,0.1,0.1).tz(0.45).ringy(4).nzy.ty(-0.05).ly(2,-0.75)
4:     + cube(0.125,1.25,0.125).zy.lx(2,0.9).cx.tz(-0.45)
5:     + cube(1.05,0.1,0.15).t(0,1.25,-0.45)
6:     + cube(0.8,0.075,0.075).ty(0.25).ly(4,0.25).tz(-0.45);
```

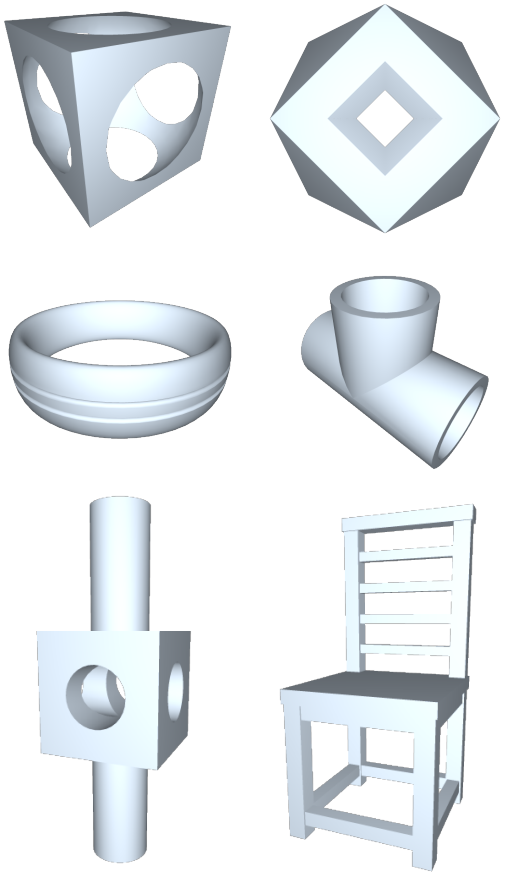


Figure 2: examples of some simple quick-mesh scripts (left) alongside the geometric objects they define (right).

Observe that each script contains a set of statements - which specify modelling operations that define a boundary-representation (a polygon-mesh) of an object. In this manner each script encodes the modelling process (i.e. the set of steps) required to define a mesh rather than the actual geometric elements (vertices/edges/faces). In simple words each script is an intermediary representation (IR).

Note: this guide focusses on the Quick-Mesh Mobile-Editor. It does not detail the Quick-Mesh Scripting-Language. To find out more about the QMSH Scripting-Language refer to the Technical-Specification and Reference-Manual and/or the QMSH Grammar Quick-Start Guide.

3 User-Action Life-Cycle

Figure 3 outlines the typical work-flow when using the Quick-Mesh Mobile-Editor. It represents the main set of actions (or tasks) that you'll undertake in order to create geometry using the app.

1) **Create or Open Script**
   a) create a new empty (blank) script
   b) create a script by copying the current script
   c) open one of the built-in example scripts
   d) open an external script file from device storage
2) **Edit Script Statements**
   a) using the built-in source-code editor
   b) referring to the built-in library documentation
3) **Save Script File**
   ...to device storage for later editing
4) **Run Script** (Assemble Mesh)
   a) as a *pure-union-mass*
   b) as a *basic-csg-solid*
   c) as an *optimal-csg-solid*
5) **Examine Mesh Interactively**
   a) using the built-in OpenGL viewer
   b) using the auto-generated parametric controls
6) **Iterate** (Optionally Repeat Steps 2-5)
   a) to make revisions that add greater detail
   b) to correct any errors in script syntax
   c) to correct any errors in spatial logic
   d) to add external control parameters
   e) to restructure the script for clarity
   f) to optimise for faster execution time
7) **Export Mesh as 3D-Model**
   ...in order to use outside of Quick-Mesh

Figure 3: outline of the work-flow of an end-user modelling with qmsh's mobile-editor - note: the similarity between traditional document editing (i.e. typesetting with a tool such as LaTeX) and the tasks qmsh users undertake.

CREATE | **OPEN**
↓
EDIT
↓
SAVE
↓
RUN
↓
EXAMINE
↓
ITERATE
↓
EXPORT

Note: remember to save your scripts.

Tip: you only need to export mesh to 3D-model files when you are ready to use your assets outside of qmsh. It is generally best (i.e. fastest with the smallest memory footprint) to save your scripts instead of your models until you are done.

Remember: script files take up very little memory, however the concrete model files they define can be quite large.

Note: that whilst there are additional tools and features built into the app, this break-down represents the general (most-common) set of tasks in using Quick-Mesh. **Note**: additionally that one can skip any unnecessary or inapplicable steps depending on the type of entity being modelled.

4 Anatomy of User-Interface

This section outlines the structure of the application's user-interface.

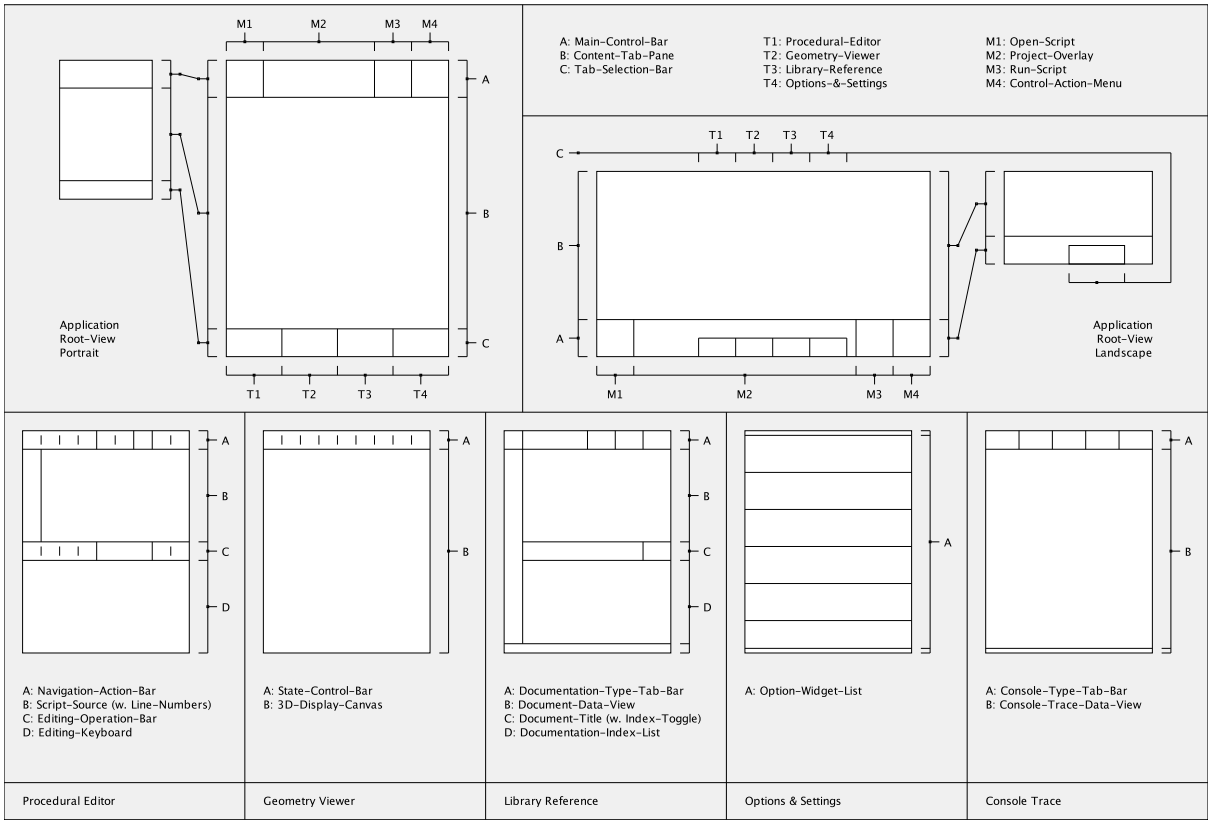


Figure 4: schematic of the layout of the mobile-editor's user-interface - representing: (top) the arrangement of the root-view's three-persistent tools in portrait and landscape screen configurations, then (below) the structure of the tools nested in the central content-tab-pane - showing (from left-to-right) the procedural-editor tab, the geometry-viewer tab, the library-reference tab, the options-&-settings tab and (finally) the console-trace tab.

The application's main UI-component is a content-tab-pane - which manages five key sub-components.

Directly above the content-tab-pane is a persistent control-bar for performing frequent actions.

Directly below the content-tab-pane is a tab-selection-bar to set which content-tab is displayed.

***

To revise: the layout of the application's user-interface is structured around three persistent tools: a control-bar, a content-tab-pane and a tab-selection-bar. Auxiliary tools and menus are overlain.

5 Key Tools

This section details the behaviour of the key tools that you shall use when modelling with the app. It begins by outlining these tools and then progresses through instructions for the use of each in turn.

- **Main Control Bar** - the primary control component - a key-action shortcut bar.
- **Tab Control Bar** - a tool selection tab-pane basebar controller.
- **Menu Action List** - a workflow action list for coordinating system tasks.
- **Project Manager** - a drop-down list for switching between open scripts.
- **Content Tab Pane** - a tabulated pane that manages the display of the Procedural-Editor, Geometry-Viewer, Parametric-Controller, Library-Reference, Options-&-Settings and Console-Trace.
- **Procedural Editor** - a quick-mesh script source-code editor - which handles user input.
- **Geometry Viewer** - an OpenGL canvas that displays 3D entities - which are the system's output - with state management, debugging and render configuration controls.
- **Parametric Controller** - a conditionally auto-generated interactive 2D GUI-controller for dynamically manipulating 3D entities constructed from scripts containing parameter-definition statements.
- **Library Reference** - an embedded reference manual for the symbols in the quick-mesh scripting language - with a summary of the key functions and extended app usage guides.
- **Options & Settings** - a scrollable control-pane tool that exposes a set of system configuration widgets - with self-documenting instructions for each control-widget included.
- **Console Trace** - a debugging console for monitoring the behaviour of the app and resolving errors.

The following subsections provide detailed instructions for the operation of these tools.

5.1 Main Control Bar

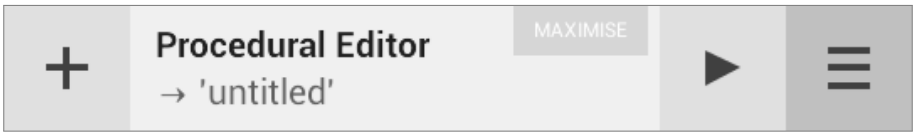


Figure 5.1: image of the application's main control bar which is used to perform four frequent actions (from left to right): the open-script shortcut, toggle project-manager indicator, run-script and toggle menu action-list.

The main control bar (also referred to as simply the *control-bar*) is the principal control component for the application. You will use it to coordinate four frequent tasks - (from left to right in Figure 5.1) these are: displaying the open-script menu, toggling the display of the project-manager tool, displaying the run-script menu and toggling the display of the main-menu action-list. Additionally - overlaying the control-bar is a maximise button which increases the screen space available to the active content-tab.

Simply tap one of the items on the control bar to perform the corresponding action.

Note: the control-bar's text-label (the project-manager toggle) also indicates which content-tab is currently active and which corresponding script, assembled mesh or sub-content-tab is currently active.

5.2 Tab Control Bar

Figure 5.2: image of the tab control bar which is used to select which tool is displayed in the content-tab-pane - illustrating the editor, viewer, library and options tool-tab selectors with the procedural-editor active.

The tab control bar (also referred to as simply the *tab-bar*) is used to set which tool-tab is currently displayed in the content-tab-pane. You can select a tool-tab by tapping its tab-bar item. When a tool-tab is active its corresponding tab-bar text-label's foreground colour changes to reflect the selected state.

Note: that the console-trace tool-tab does not have a corresponding tab-bar selection item.

Note: in order to display the console-trace - deselect the currently selected tool-tab by tapping its tab-bar item again. In this manner you can quickly toggle between displaying the console-trace and one of the other tool-tabs by repeatedly tapping the same tab-bar item.

5.3 Project Manager

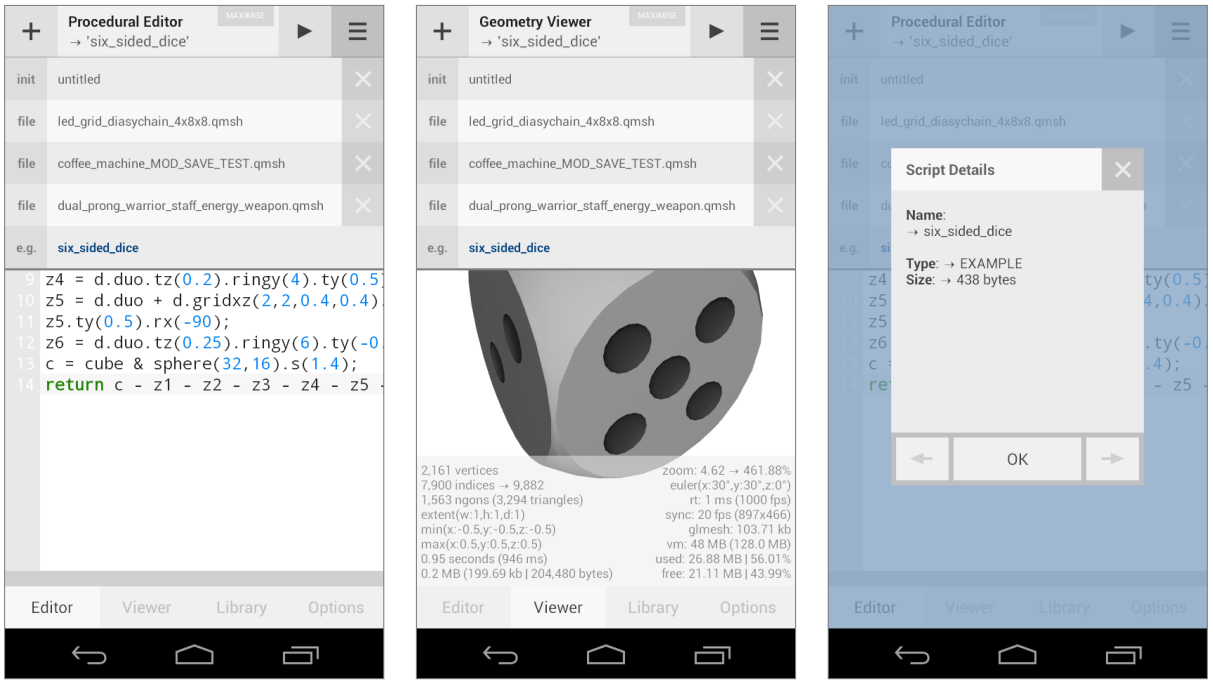


Figure 5.3: screenshots of the application's project-manager tool - illustrating (from left to right): the appearance and relative position of the tool's drop-down selector list (just beneath the main-control-bar) - which is used to switch between open scripts - with two different tool-tabs active in the content-tab-pane (the procedural-editor and the geometry-viewer), then the tool's info-overlay - which provides summary details of the state of open scripts.

The project manager tool is used to coordinate the state of multiple open quick-mesh scripts. It enables you to quickly switch between different open quick-mesh scripts for editing and assembly.

There is always one active-script within the project manager. This active-script corresponds to the script that is currently being edited and the script that is to be fed to the kernel for assembly as a mesh.

Every time you open a script a new row is added to the project manager's selector list to represent (the state of) and manage the script. The script you open also becomes the currently active script.

Each script-row encompasses a script-type-indicator (left) a script-activating name-indicator-item (centre) and (for all but the currently active script) a close-script button.

Note: you can activate an open script by tapping its name-indicator-item.

In the project-manager: the currently active script is indicated by bold typeface and by the absence of a close button for the script's row. Figure 5.3 (left and centre) helps to clarify.

Note: there is (technically) no concrete limit to the number of open scripts that may be coordinated by the project manager. However in practise the number of open scripts will be tied to the number of

entities you are working with and upon concurrently.

Note: that if the number of open scripts exceeds the maximum number of script rows that can be displayed then the project manager's list pane employs scrolling to accommodate the overflow.

You can close each non-active open script by tapping the cross to the right of its activation item (its close-script button). However if you attempt to close a script that has unsaved modifications then the project manager displays a modal confirmation dialog with which you can either confirm the close operation without saving (which will result in the modifications being discarded) or cancel the close operation (which will allow you to save the script's modifications to device storage).

You can display summary details for each open script by tapping its script-type-indicator - which triggers the display of the script-info. overlay. This includes the current size of each script (in bytes), the type of each script, and (where applicable) the absolute file location of a script.

***

To revise: the project-manager is the main script-management tool exposed by the app and is used to switch between multiple open scripts - to enable concurrent script editing.

5.4 Procedural Editor

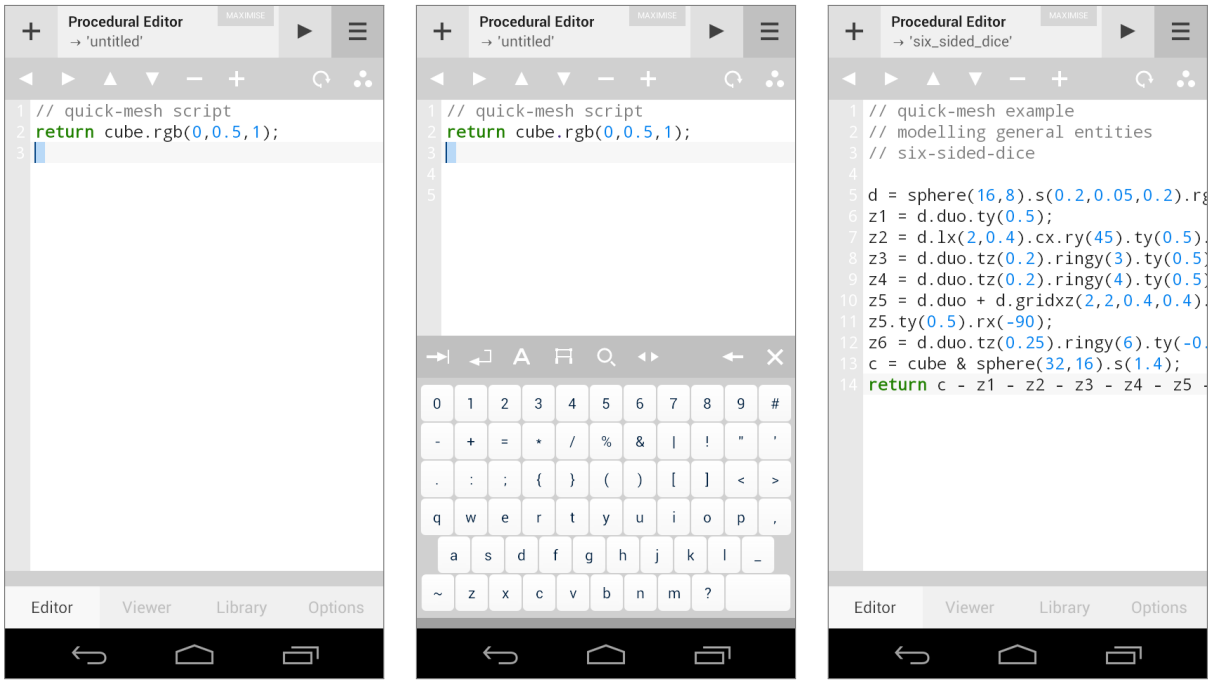


Figure 5.4a: screenshots of the app's procedural-editor tool - illustrating (from left to right): the appearance of the editor's text-viewer for a new untitled script, the appearance of the editor's input-keyboard upon selecting characters to edit and (finally) the appearance of the editor's text-viewer after opening an example generative modelling script.

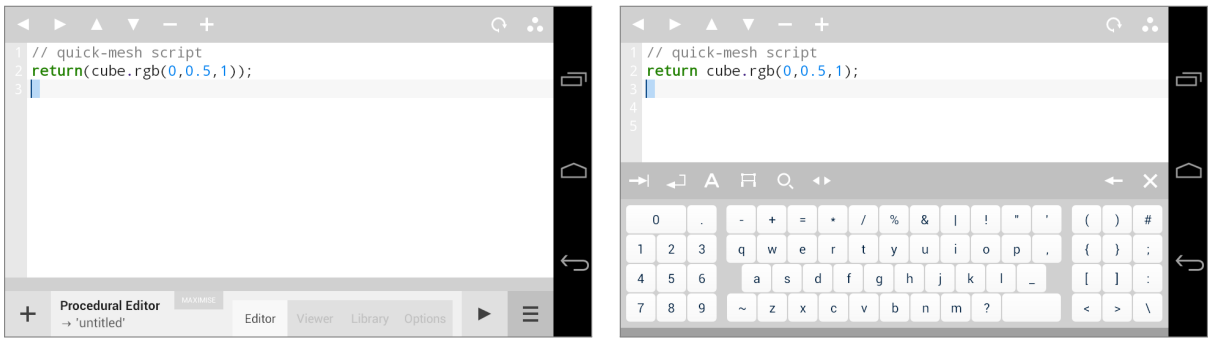


Figure 5.4b: screenshots of the app's procedural-editor tool in its landscape configuration - illustrating (from left to right): the editor's position relative to the landscape control-bar, and the alternate layout of the keyboard.

The procedural-editor is the main user-input component for the app - and the default content-tab that is displayed upon program launch. You will use it to write and modify modelling scripts.

The procedural-editor has an action-bar and a script-editor widget (text-viewer + input-keyboard).

To display the keyboard simply tap script characters in the text-viewer. You can also position the cursor by tapping characters. You can also use the shortcut arrow keys from the action-bar to move the cursor left, right, up and down as you would with a physical keyboard.

To hide the keyboard use the keyboard's close button (top-right-hand-corner). **Note**: you can also hide the procedural-editor's input-keyboard using the device's hardware back-button.

To alter the font-size of the script-editor - tap the action-bar's + (plus) and - (minus) buttons.

You can also cut, copy and paste source-code amongst open scripts using the keyboard's selection and internal-clipboard menu actions. Additionally you can select the procedural-editor's action-bar's tri-dot highlight-menu button to toggle the display of auxiliary grammar-centric syntax-highlighting functionality (such as emphasising scoping symbolic-operators and dot-notation operators).

One thing to remain aware of is that the procedural-editor is NOT designed (nor intended) to be a general-purpose text-editing component. It is a (qmsh) grammar-centric component - in that it restricts the scope of editing facilities to only those required by the Quick-Mesh scripting language. Essentially it intentionally supports only a subset of the symbolic-input options provided by a device's native virtual keyboard. For example it purposefully does not apply auto-correction as you type - in order to prevent undesirable script modifications - and its input-keyboard is a single-pane component (rather than multi-pane) such that all relevant symbolic characters are accessible from a single view. In short the procedural-editor is designed solely for the purpose of viewing and modifying Quick-Mesh scripts - nothing more. Indeed the .qmsh extension is the app's sole supported input file format.

Note: that the procedural-editor does not automatically update its syntax highlighting as you type. This is primarily to reduce editing latency on lower-power devices. To manually trigger an update use the refresh-button from the procedural-editor's action-bar when you stop typing.

Note: that for editing longer scripts you can also disable syntax highlighting all together by using the options-and-settings-tab-pane's control-item (see Options & Settings section for more).

Note: the procedural-editor's text-viewer uses a monospaced font.

Note: when the editor's keyboard is displayed it masks the content-tab-selection-bar.

***

To revise: the procedural-editor is the app's primary user-input component for script-editing. It is the principal method of controlling the behaviour of the QMSH-Kernel within the app and it is the tool you will use most frequently to coordinate the definition of 3D procedural polyhedral entities. For this reason it is worthwhile taking the time to familiarise yourself with its core operation and use.

5.5 Geometry Viewer

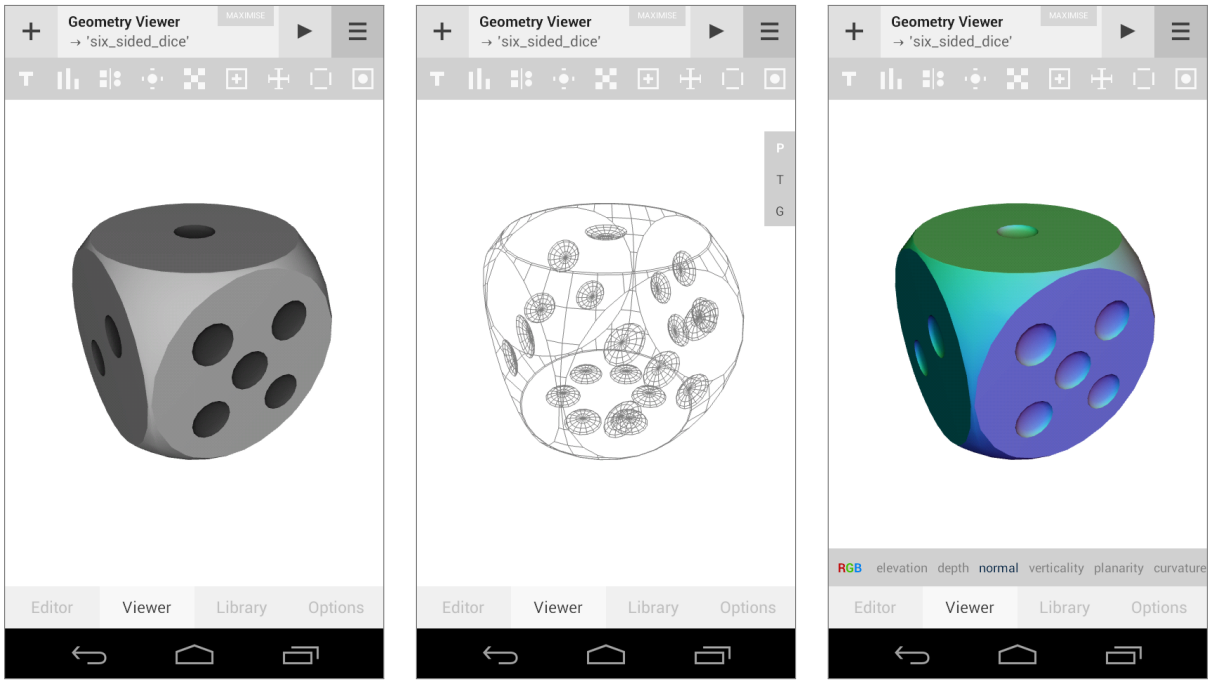


Figure 5.5a: screenshots of the geometry-viewer (post assembly of the example script *six-sided-dice*) - illustrating (from left to right): the resulting mesh rendered using the default viewer-state (showing mesh RGB colours), the mesh rendered with the *topology-option edges* active, and the mesh rendered with the *colour-mode* set to *functional* and the colourisation-bar visible with multi-channel surface-normal-derived per-vertex colours applied.

The geometry-viewer is the main system-output component for the app - and is responsible for visualising (rendering) the polyhedral entities constructed by the app's kernel. You will use it in concert with the procedural-editor to examine your procedural creations during the iterative process of development.

The structure of the geometry-viewer is similar to that of the app's user-interface - in that the geometry-viewer also provides a persistent control-bar - to configure key state.

The geometry-viewer's state control-bar overlays an OpenGL canvas into which the application renders 3D graphical representations of assembled entities. Most of the options exposed by the state control-bar affect the manner in which the OpenGL canvas displays its 3D content.

The geometry-viewer also has a conditionally overlain attribute-trace-pane (which can be used for debugging) and a state-triggered functional-colourisation-control-bar (which is useful for mesh debugging).

Note: you can use one-finger touch-input swipe gestures to change the camera's orbit orientation.

Note: you can use two-finger touch-input pinch gestures to change the camera's zoom level.

5.5.1 State Control Bar

Figure 5.5b: image of the geometry-viewer's state-control-bar - depicting the option selector item icons for (from left to right): the topology, colour, shading, orbit, projection, state, widget, view and capture controls.

The geometry-viewer's state-control-bar is used to configure the current state of the geometry-viewer.

Its operation is relatively simple. In order to configure state select an item from the state-control-bar (by tapping it) in order to display a corresponding option-group sub-item selector. Then select a sub-item from the group to apply the option that it maps to.

Note: that the currently selected option-group remains visible when displayed (i.e. it persists until hidden or dismissed). You can hide or dismiss it by tapping its corresponding control-bar item again.

For reference - the geometry-viewer's state-control-bar exposes the following set of control options:

Topology : { *Surface | Edges | Mixed }

Surface → *display entity surface exclusively*

Instructs the geometry-viewer to render only the polyhedral surface of constructed entities.

Edges → *display entity edges exclusively*

Instructs the geometry-viewer to render only the polyhedral edges of constructed entities.

Mixed → *display entity surface and edges simultaneously*

Instructs the geometry-viewer to render both the polyhedral surface and edges of constructed entities.

Colour : { *RGBA | Dark | Light | Function }

RGBA → *display entity surface with user-defined vertex colours*

Instructs the geometry-viewer to render polyhedral surfaces using colours specified by a user (i.e. invoking colourisation functions within a script) as the source for per-vertex entity colouring.

Dark → *display entity surface with dark-override vertex colours*

Instructs the geometry-viewer to render polyhedral surfaces using a system specified dark override colour for entity visualisation. This results in entities appearing to be a dark monolithic colour irrespective of user specified per-vertex colours. Note: that this option does not modify entity data and is provided primarily as a convenience for inspecting mesh.

Light → *display entity surface with light-override vertex colours*

Instructs the geometry-viewer to render polyhedral surfaces using a system specified light override colour for entity visualisation. This results in entities appearing to be a light monolithic colour irrespective of user specified per-vertex colours. Note: that this option does not modify entity data and is provided primarily as a convenience for inspecting mesh.

Function → *display entity surface with functionally-defined vertex colours*

Instructs the geometry-viewer to render polyhedral surfaces using functional colours derived from an entities' geometry, topology or semantics. When applied this option triggers the display of the geometry-viewer's functional colour bar - with which you can select the function used to compute entity colours.

Shading : { Flat (Faceted) | *Smooth (Gouraud) | Transparent-Mode | Specular-Mode }

Flat (Faceted) → *display entity surface with flat shaded faceting effect*

Instructs the geometry-viewer to render polyhedral surfaces as flat-shaded (faceted) mesh wherein surface-normals are not interpolated over faces adjacent to vertex. This option emphasises discontinuity between faces and results in *crisp* edges.

Smooth (Gouraud) → *display entity surface with smooth shaded effect*

Instructs the geometry-viewer to render polyhedral surfaces as smooth-shaded (Gouraud) mesh where surface-normals are interpolated for faces adjacent to vertex resulting in surfaces appearing to exhibit smoothly changing continuous transitions. This option masks discontinuity between faces in shading.

Transparent-Mode → *display entity surface with alpha-transparency pseudo-modifier effect*

Instructs the geometry-viewer to render polyhedral surfaces with an alpha transparency based *see-through* shading effect. This option does not modify entity data and is provided primarily as a convenience in lieu of an x-ray effect. Note: this option can be combined with the *mixed-topology* option.

Specular-Mode → *display entity surface with specular-highlighting effect*

Instructs the geometry-viewer to render polyhedral surfaces with specular-highlights to provide a sense of shininess. This option enhances the appearance of entities during orbiting events by simulating a common visual phenomena - however this does not modify entity data - the effect is in shading only.

Orbit : { *Euler-Gimbal | Quaternion | Free-Roam | About-Center }

Euler-Gimbal → *use simplified euler-angle rotation in camera orbiting events*

Instructs the geometry-viewer to employ Euler rotations in the coordination of touch-input triggered camera orbiting events. This option constrains the scope of camera orientations attainable - however it simplifies the process of preserving entity up-right orientation whilst carousel-style orbiting is applied.

Quaternion → *use simplified quaternion rotation in camera orbiting events*

Instructs the geometry-viewer to employ Quaternion rotations in the coordination of touch-input triggered camera orbiting events. This option provides a wider range of possible camera orientations (relative to the Euler-Gimbal option) - however it affords less intuition in tasks such as preserving entity

up-right orientation. Note: this option overrides the application of the Euler-Gimbal orbit option.

Free-Roam → *use generalised free-roam camera motion for orbiting events*

Instructs the geometry-viewer to employ an eye-lookat based motion controller in the coordination of touch-input triggered camera orbiting events. This option offers the most flexibility in camera control (relative to the Euler-Gimbal and Quaternion options) largely because it allows the focal point to be varied - vastly simplifying the examination of hard to reach portions of an entity. Note: enabling this option also triggers the display of an accompanying free-roam motion-pad - which can be used alongside one-finger move and two-finger pinch events to control the entity viewing position. Note: this option overrides the application of the Euler-Gimbal and Quaternion orbit options.

About-Center → *use entity aabb-midpoint as orbiting pivot instead of origin*

Instructs the geometry-viewer to use the axis-aligned bounding-box middle-point of an entity as the pivoting point in camera orbiting events - rather than the origin (0,0,0).

Projection : { Orthographic | *Perspective }

Orthographic → *render with orthographic camera projection*

Instructs the geometry-viewer to render polyhedral surfaces using an orthographic camera projection - which has the effect of preserving parallelism in the display of entities.

Perspective → *render with perspective camera projection*

Instructs the geometry-viewer to render polyhedral surfaces using a perspective camera projection - which induces the appearance of depth correction in the display of entities (i.e. elements that are further away appear smaller then near objects). Note: the field-of-view is fixed at 60°.

State : { *Attribute-Trace | Fit-Viewport | Reset-Transform }

Attribute-Trace → *toggle display of geometry-viewer attribute-trace*

Instructs the geometry-viewer to show or hide the conditionally overlain attribute-trace component.

Fit-Viewport → *modify camera zoom to fit entity in viewport*

Instructs the geometry-viewer to modify the camera's zoom level to roughly fit the extents of the assembled entity in the OpenGL viewport. Note: this option does not modify the camera's orbit's orientation unless the currently active orbiting mode is set to free-roam.

Reset-Transform → *reset camera zoom and orbit to initial (default) state*

Instructs the geometry-viewer to reset the camera's current zoom level and orbit orientation to their default (start-up) state. This option applies to both Euler-Gimbal and Quaternion orbiting state.

Widgets : { Axes | Grids | Bounds }

Axes → *toggle display of origin axes indicator widget*

Instructs the geometry-viewer to show or hide the principal axes indicator.

Grids → *toggle display of multi-scale wire grid indicator widget*

Instructs the geometry-viewer to show or hide the XZ grid indicator.

Bounds → *toggle display of entity axis-aligned-bounds indicator widget*

Instructs the geometry-viewer to show or hide the entity bounds indicator.

View : { Top | Base | Left | Right | Front | Back }

Top | Base | Left | Right | Front | Back → *modify camera orbit to 'look-at'*

Instructs the geometry viewer to alter the current camera orbit attributes so to 'look-at' the side selected. When coupled with an orthographic projection - this can be helpful in the creation of engineering-styled plan-views. Note: this option is non-persistent - the side-view selected is only preserved up until the next orbit event. Note: the application of this option does not alter the camera's zoom-level.

Capture : { Snapshot | Directive }

Snapshot → *display image snapshot capture overlay*

Instructs the geometry viewer to capture a snapshot image of the active entity using the current camera configuration. Note: the snapshot is displayed by the image-capture-pane with which you can save or dismiss the snapshot. Refer to the Image Capture Pane section for more on snapshot creation.

Directive → *display camera kernel-directive creator overlay*

Instructs the geometry viewer to display a free-roam camera viewpoint kernel-directive creator utility - which can be used as a helper to generate source-literals and to manually set the configuration of the free-roam controller. Note: that the directive item is only visible when the currently active orbiting mode is set to free-roam (as it depends upon eye-lookat configurability) - and is hidden otherwise.

5.5.2 Attribute Trace

The geometry-viewer's attribute-trace-pane displays readouts of key metrics - related to the current entity and the visualisation state. The attribute-trace is a conditionally overlain component which has a translucent backdrop and can be toggled on and off using the state-control-bar's attribute-trace item.

Note: the attribute-trace is enabled by default.

You typically use the attribute-trace to monitor the attributes of an entity during the iterative process of editing. These attributes include: the number of vertices, indices, polygons and triangles in a mesh, as well as spatial measures such as the axis-aligned-bounding-box (AABB) extents (width, height, depth), minimum and maximum - alongside computational properties such as the time taken to assemble a mesh and the amount of memory required to store its elements on disk as a raw binary data-file.

You can also use the attribute-trace to check the state of the interactive-visualisation coordinated by the geometry-viewer - such as the camera's zoom-level and orbiting properties - alongside the application's lower-level measures - including the virtual-machine load and the GL (on-GPU) mesh size.

5.5.3 Colourisation Bar

Figure 5.5c: image of the geometry-viewer's functional-colourisation-bar - depicting (left RGB) the multi-channel or grayscale toggle next to (right) the scrolling function selector with the *colour-by-normal* function selected.

The geometry-viewer's functional-colourisation-bar is used to select the data-driven entity colouring function that is applied to assembled entities for the purpose of in-editor mesh analysis and debugging.

Note: that the colourisation-bar's actions are non-persistent - in the sense that they do not mutate the state of a mesh permanently (i.e. on export) but rather only in-editor. This behaviour is equivalent to the state-control-bar's dark and light colour override items. To display the functional-colourisation-bar select the function item from the state-control-bar's colour option drop-down menu.

Figure 5.5.c illustrates a close-up view of the colourisation bar in order to help clarify.

Note: you can select the lefthand RGB button to toggle between multi-channel and grayscale modes.

The following enumeration outlines the behaviour of the supported data-driven colouring functions.

- **Elevation** : colours an entity's vertex by relative height - i.e. akin to a height-map.
- **Depth** : colours an entity's vertex by relative depth - i.e. akin to depth-map.
- **Normal** : colours an entity's vertex according to unit-length surface-normal.
- **Verticality** : colours an entity's vertex by a surface-normal derived scalar verticality measure.
- **Planarity** : colours an entity's vertex by a surface-normal derived variance-based measure of local planarity - with values normalised across the entity to highlight the minima and maxima.
- **Curvature** : colours an entity's vertex by a surface-normal derived variance-based measure of local curvature - normalised across the entity. Note: this is the inverse to colour-by-planarity.
- **Valence-NGON** : colours an entity's vertex by relative valence using N-gonal indices - i.e. by a scalar measuring the number of polygonal elements that share a vertex.
- **Valence-TESS** : colours an entity's vertex by relative valence using triangulation indices - i.e. by a scalar that measures the number of triangle elements that share a vertex.
- **Declaration-Order** : colours an entity's vertex according to the order in which they are declared.
- **Operand-Identifier** : colours an entity's vertex according to the operand to which they belong - such that vertex corresponding to the same underlying primitive share the same colour.
- **Face-Identifier** : colours an entity's vertex by the facet-index in the operand to which they belong.
- **Group-Identifier** : colours an entity's vertex according to Gouraud smooth-shading group.

5.5.4 Image Capture Pane

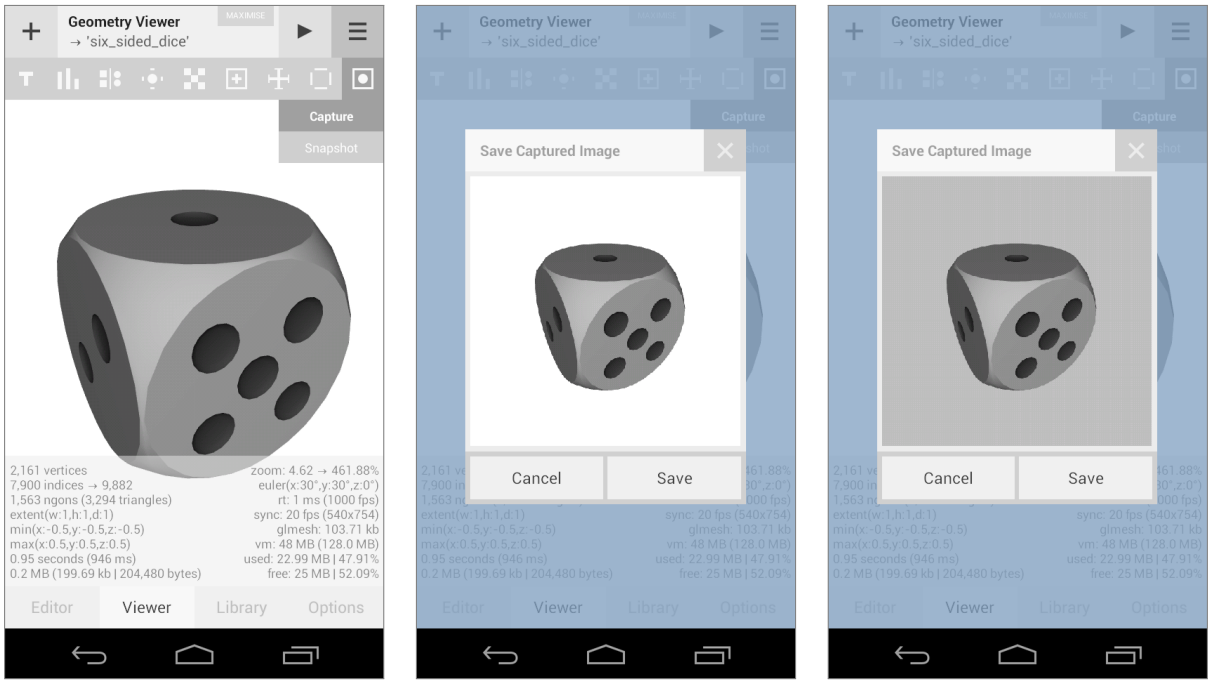


Figure 5.5d: screenshots of the geometry-viewer's image snapshot functionality - illustrating (from left to right): the state-control-bar's capture menu, then (post-capture) the overlain image-capture pane with (centre) the default clear colour applied and (right) an alternative clear colour selected by tapping the pane's image preview.

The geometry-viewer's image-capture-pane is a simple tool for quickly creating snapshots of an entity using the Open-GL canvas. Selecting the state-control-bar's capture-snapshot item will create a square image (of the active entity) whose camera properties are copied from the Open-GL canvas.

Note: you can cycle through preset backdrop grayscale clear colours by tapping the image preview.

You can save the captured snapshot to device storage as a time-stamped PNG by selecting the save button. Alternatively select the cancel button (or the close button) to dismiss the image-capture-pane.

***

To revise: the geometry-viewer is responsible for displaying the system's output 3D mesh. It is the primary means of inspecting and examining the entities generated by the app's kernel. Its interface-layout comprises an OpenGL-canvas, a state-control-bar, an attribute-trace-pane and a functional-colourisation-bar. It also exposes an image-capture-pane - which overlays the UI's content when active.

5.6 Parametric Controller

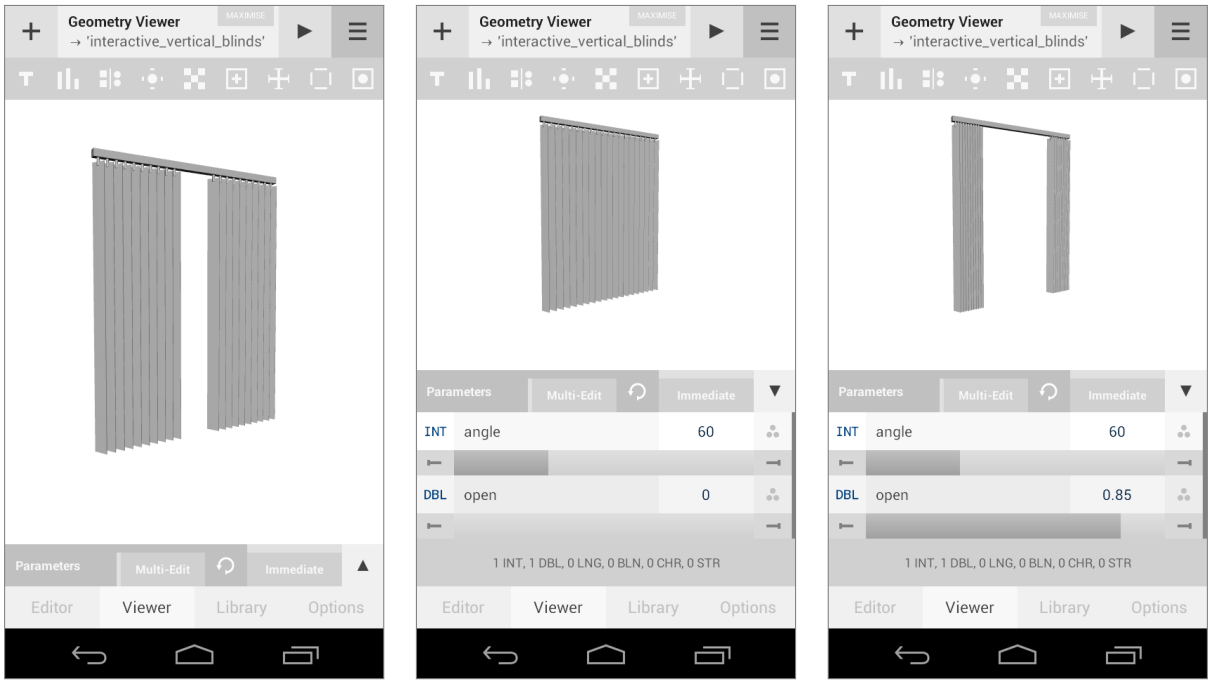


Figure 5.6a: screenshots of the parametric-controller component - illustrating (from left to right): its appearance in the geometry-viewer for a parametric-entity (*interactive-vertical-blinds.qmsh*) when the parametric-controller is collapsed, then with the parametric-controller expanded and a mutation applied to the entity (by altering the parameter-values using the auto-generated parameter-widgets), and then expanded with a further modification applied.

The parametric-controller is the mobile-editor's main interactive entity control component.

Its appearance alongside the geometry-viewer post-assembly of an entity is conditional on the presence of parameter-definition statements in a script.

Note: the parametric-controller is displayed below the geometry-viewer's Open-GL canvas - and reduces the available screen-space which can be used by the canvas (see figure 5.6a).

The controller's layout comprises an action-bar and an auto-generated parameter-widget list.

The action-bar contains (from left-to-right) a parametric-variant trigger button, a multi-edit-mode toggle, a reset button, an immediate-mode toggle and an expand-collapse toggle.

The parametric-variant trigger displays a parametric-variant kernel-directive creator and selector tool as an overlain dialog - which can be used to coordinate discrete parametric instances of an entity.

The multi-edit-mode toggle turns multiple-parameter-mutation mode on and off. Multi-edit-mode applies to all parameters and essentially enables you to edit multiple values before explicitly triggering re-assembly - which can be useful in coordinating parameter-edits for longer running assembly tasks.

The reset action causes the current parameter configuration to be restored to the default (initial) state specified in the defining script. Due to the fact that this will cause the current state of the active entity to be lost - the reset action requires confirmation through a modal dialog.

The immediate-mode toggle turns instant re-assembly on and off. Immediate-mode applies to constrained parameters. Note: when immediate mode is enabled the multi-edit-mode toggle is hidden.

The expand-collapse toggle enables you to show and hide the controller so to temporarily increase the available screen-space that the Open-GL canvas can occupy. This can be helpful in examining the effect that changing a parameter has on an entity for example during inspection via functional colourisation.

Additionally - the action-bar remains visible when the controller is collapsed.

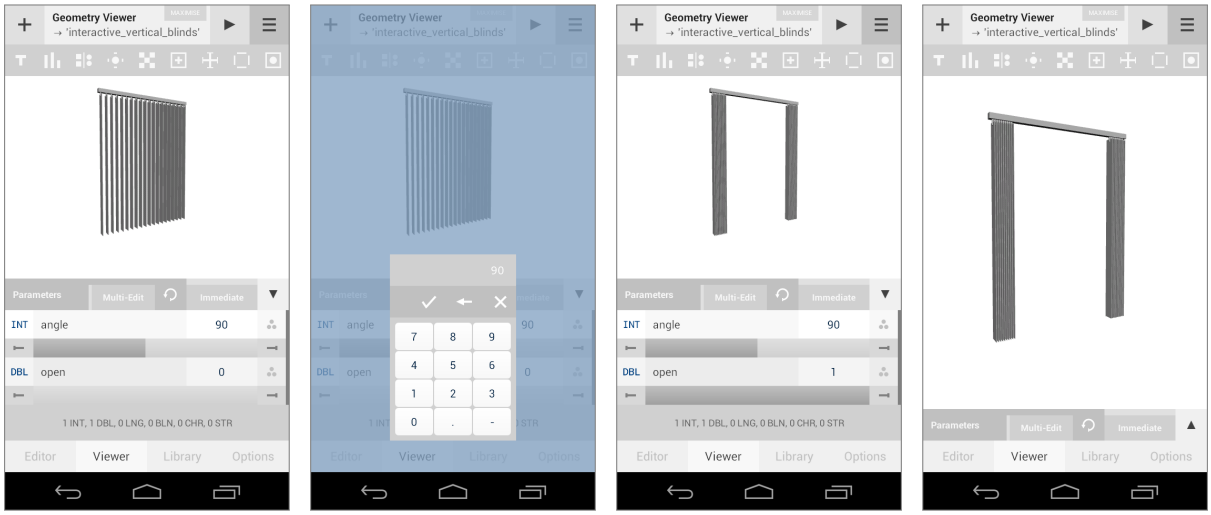


Figure 5.6b: further screenshots of the parametric-controller component - illustrating (from left to right): the appearance of the controller through various states of interactive parametric entity examination - including (second from left) the overlain number-pad which can be used (in lieu of the sliders) to set parameters to specific values.

The parameter-widget list displays a simple controller for each parameter defined by a script.

A script's unconstrained parameters are exposed as single-row parameter-widgets - with a text-label, type-indicator and value-field. Constrained parameters are exposed as double-row parameter-widgets - i.e. as unconstrained parameters with the addition of a secondary numeric slider row.

Note: an interesting feature of the quick-mesh grammar is the ability to declare parametric entity generators in a platform and environment agnostic manner. The behaviour of this component represents a concrete manifestation of this abstract capability. Whenever you alter a parameter's value the polyhedral mesh representation of the entity is re-assembled using the updated parameter value.

For both unconstrained and constrained parameters these updates can be triggered by changing the corresponding field value. Note: that the default behaviour of the numeric sliders exposed for constrained parameters is to trigger updates only on tracking-end events (and not on tracking-start and

tracking-slide events). The immediate-mode toggle alters this behaviour such that updates are also triggered for tracking-slide events. When immediate-mode is enabled - if the time taken to re-assemble a mesh is near negligible then the mutation has the effect of appearing to be a realtime animation. However if the time taken to re-assemble the mesh exceeds a fraction of a second then the parametric-controller displays an interaction blocking mask over the widget-list to prevent further alterations until the running re-assembly task has finished and its effect has been loaded into the Open-GL canvas. During these parameter-lock periods the geometry-viewer's canvas remains responsive for interaction such that you can continue to examine your creation whilst the parametric re-assembly task is running. When the task is complete the revised mesh is displayed and the parameter-mutation-lock removed.

The last noteworthy aspect of the parametric-controller is the variant creator and selector (see fig. 5.6c).

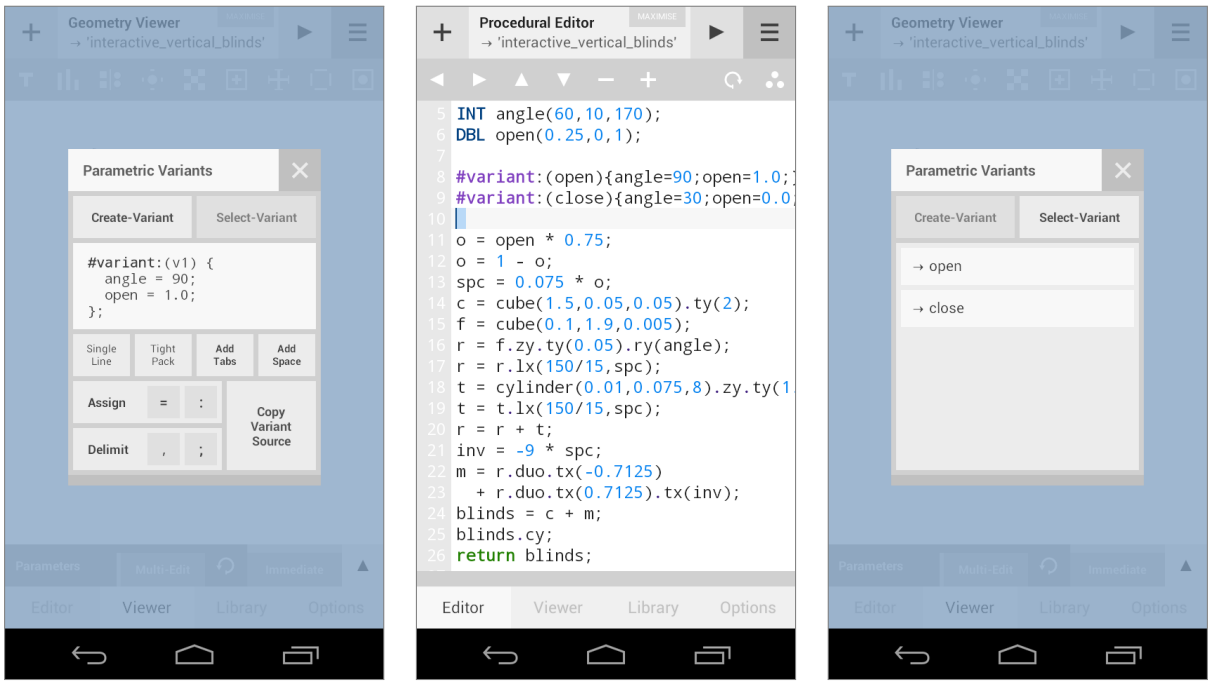


Figure 5.6c: screenshots of the parametric-controller's variant configurator - illustrating (from left to right): the appearance of the variant creator, then the entity's source with variant's added then (finally) the variant selector.

The variant creator (fig. 5.6c left) generates parametric-variant kernel-directive source-code for the current entity - whilst the variant selector (fig. 5.6c right) enables you to quickly apply distinct variants.

***

To recap: the parametric-controller is a powerful feature of the mobile-editor that is not shared by command-line use of the kernel. The thing to remember is that the responsiveness of the parametric controls is governed by how long a script takes to assemble. Scripts that execute quickly can effectively be treated as functionally defined interactive entities - whilst long running scripts generally can not be.

5.7 Library Reference

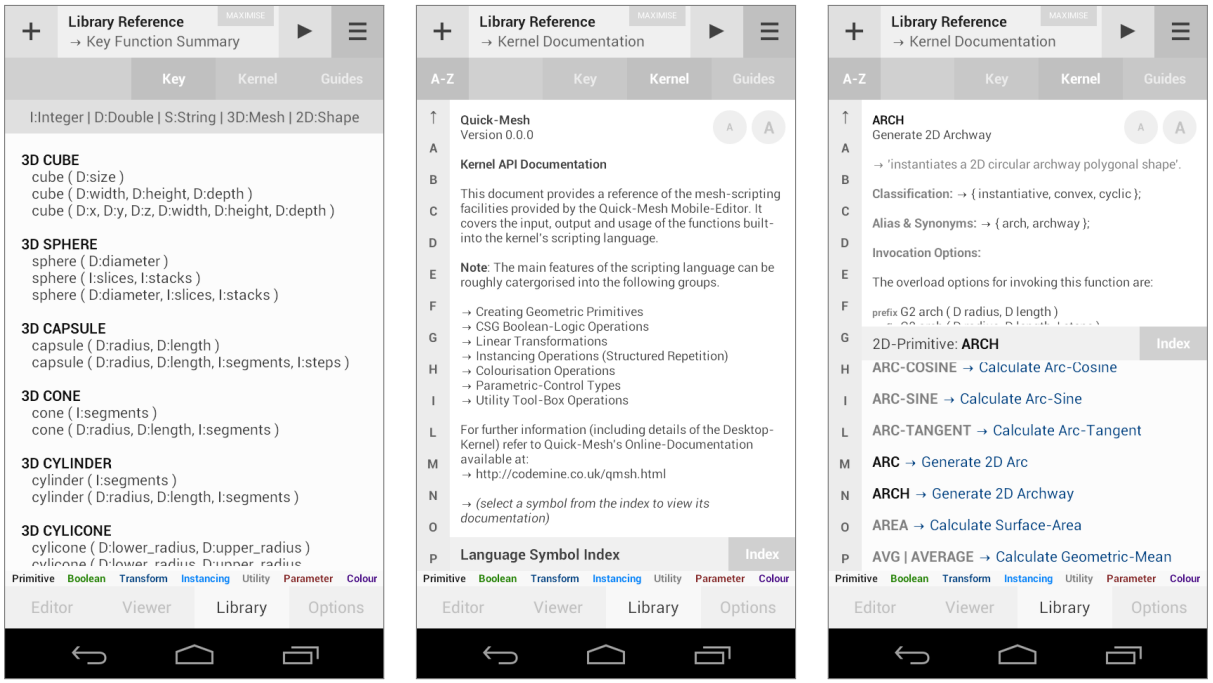


Figure 5.7: screenshots of the library-reference component - illustrating (from left to right): the key function summary pane (the default library-reference sub-tab), the kernel symbol reference pane's introductory overview and the kernel symbol reference pane with the symbol-selection-index visible and the *arch* symbol selected.

The library-reference tool-tab is the mobile-editor's technical-documentation component and is responsible for displaying manual pages for each native function in the quick-mesh grammar. It also includes a summary of commonly used functions for quick-reference and extended usage guides explaining aspects of the quick-mesh grammar, kernel and editor. Refer to figure 5.7 for clarification.

Its layout comprises a tab-bar which allows you to switch between the function summary, the full kernel-documentation and the extended-guides. There is a colour-coded key below the library-reference's central pane whose items map to the operational classes to which each manual entry belongs.

Note: indexing components are included to simplify the traversal of manual entries. For example you can use the symbol-index list and the A-Z jump-to shortcuts to navigate the set of native functions.

When you first start using the mobile-editor you will find you are constantly referring to the library-reference to check which functions to use and how to use them. As time progresses and you grow more familiar with the kernel and the grammar you will find you use the library-reference tool less and less.

Note: the reason the documentation is embedded in the editor is to ensure it is alway accessible. This means that you can learn how to use the editor without an internet connection simply by exploring the inline documentation. The other logistical benefit is that it ensures that the version of the documentation you are reading is always in sync with the version of the kernel and editor that you are actually using.

Remember: whenever you are unsure of how to use a function - check the library-reference.

5.8 Options & Settings

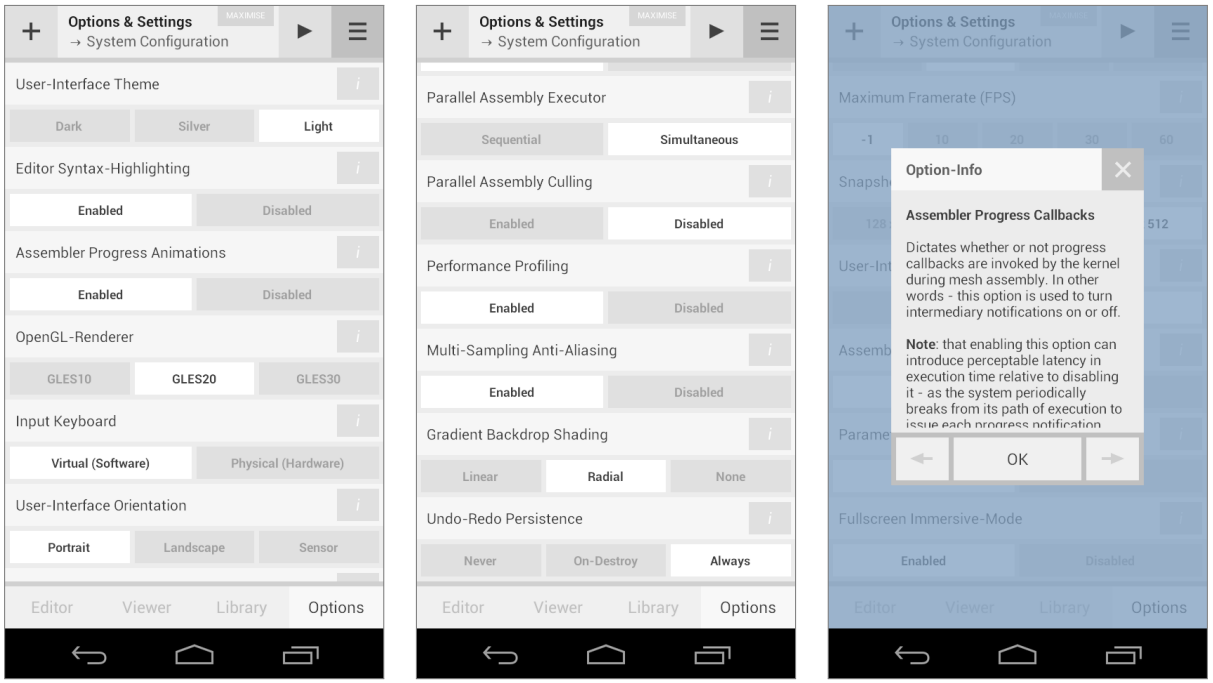


Figure 5.8: screenshots of the options-and-settings component - illustrating (from left to right): the appearance of the tool when scrolled to top (with the first six option widgets visible), then when scrolled to tail (with the last six option widgets visible) and (finally) the option info. overlay with the *assembler-progress-callbacks* option selected.

The options-&-settings tool-tab (see figure 5.8) is the main configuration component for the mobile-editor - and exposes a scrollable list of widgets for controlling the state of global options and settings.

Its operation is relatively simple to intuit. In terms of its structure each option-widget comprises a text-label (indicating the name of the option), an info. button (which is used to display extra information about the option) and a radio-group enumeration selector (to choose the configuration and control the state of the option). The info-button for each option-widget triggers the display of an overlain dialog within which each option's description is presented. In this manner this tool is largely self-documenting.

Note: the options-&-settings tool does not by default cache state between restarts of the mobile-editor. This means that (unless the session-persistence option is explicitly enabled) then all options and settings are restored to their default state whenever the application is launched.

The remainder of this section documents the behaviour of the options and settings exposed.

Note: for each of the succeeding options the default state is indicated by an asterisk.

User-Interface-Theme : { Dark | *Silver | Light }
→ Determines the colour-scheme that is employed for the user-interface components. This option applies to the procedural-editor, geometry-viewer, library-reference, console-trace and the menus and

dialogs. Note: you can set the colour-scheme in order to adapt the display to your current environmental conditions. For example when it is dark (such as at night) the Dark colour-scheme helps to minimise the brightness of the display and puts less strain on the eyes - whereas in heavy ambient-light conditions (such as outside on a sunny day) the Light colour-scheme can help provide greater display clarity due to the nature of the contrast. Note: each theme's colour-scheme is preset and immutable.

Editor Syntax-Highlighting : { *Enabled | Disabled }

→ Determines whether or not the procedural-editor display scripts with highlighted syntax - or as mono-colour text. When this option is enabled the procedural-editor highlights numbers, literals, types and keywords with distinctive colours so to enhance the legibility of a script's structure.

Assembler Progress Animations : { *Enabled | Disabled }

→ Determines whether or not the immersive or basic progression animations are displayed during mesh assembly. Note: that enabling this option will likely introduce additional latency in execution time relative to disabling it - as the system uses resources (CPU) to coordinate repaint events whilst assembly is occurring. For bench-marking performance (script evaluation times) it is recommended that this option is disabled to provide the most stable indication of runtime.

OpenGL-Renderer : { GLES10 | *GLES20 | GLES30 }

→ Controls the version of the OpenGL standard used by the geometry-viewer to render 3D content. Note: that the more recent versions typically produce higher-quality graphics in terms of being better at mitigating artefacts such as aliasing. However the version of renderer employed does not alter the geometry that is displayed - only its visual presentation. Note: support for GLES20 and GLES30 varies by device and is contingent on hardware support for multi-sampling anti-aliasing (MSAA).

Input Keyboard : { *Virtual (Software) | Physical (Hardware) }

→ Determines which type of keyboard should be used to provide symbolic input to the procedural-editor. This option enables one to control which input-source is used in the event that there is an alternative USB or Bluetooth peripheral connected to the device. Note: that if an alternative peripheral is not present - this option can be used to toggle the display of the procedural-editor's keyboard on and off - for example to use the procedural-editor as an immutable script-viewer rather than as an editor.

User-Interface Orientation : { *Portrait | Landscape | Sensor }

→ Controls the orientation of the user-interface.

Kernel-Execution Memory-Limit : { 64 MB | *128 MB | 256 MB | 512 MB }

→ Determines the maximum amount of memory (in terms of RAM in megabytes) that may be allocated (by the operating-system) to the kernel during mesh assembly.

Maximum Framerate (FPS) : { *-1 | 10 | 20 | 30 | 60 }

→ Controls the maximum number of repaint events per second during user-interaction events. This option applies to the visualisation and examination of mesh in the geometry-viewer. Note: that -1 corresponds to an unspecified maximum framerate. Note: that this option can also be used (at least in principle) to extend battery life of an executing device, by reducing the frequency with which repaint-events are triggered during orbiting and zoom events.

Snapshot Image-Resolution : { 128 x 128 | 256 x 256 | *512 x 512 }

→ Controls the width and height (in pixels) of snapshot images generated by the geometry-viewer's capture tool. Note: that support for each image resolution is dependent on the physical screen dimensions of the device and the current interface configuration. To clarify: the maximum snapshot resolution

that the geometry-viewer can support on a device in a given screen-configuration (portrait or landscape) is determined by the minimum of the OpenGL canvas' available width and height.

User-Interface Animations : { Enabled | *Disabled }

→ Determines whether or not translational and alpha-transparency animations are employed during interface transitions - such as moving between tabs and toggling the display of the menu and the various widgets and dialogs. Note: that (generally) enabling animation introduces slight latency relative to disabling animations - which may be undesirable on lower-power devices.

Assembler Progress Callbacks : { *Enabled | Disabled }

→ Dictates whether or not progress callbacks are invoked by the kernel during mesh assembly. In other words - this option is used to turn intermediary notifications on or off. Note: that enabling this option can introduce perceptible latency in execution time relative to disabling it - as the system periodically breaks from its path of execution to issue each progress notification. Hence: as for the (related but distinct) assembly-progress-animation option - it is recommended that this option is disabled for bench-marking performance. Note: the progress notifications are internal to the mobile editor and do not pollute the system bar or trigger re-display when the editor is no longer the foreground process.

Parameter Synchronisation : { *Enabled | Disabled }

→ Determines whether or not external control parameters are (when applicable) automatically synchronised between successive invocations of a script. Note: that this option only applies to scripts containing parameter definition statements. Enable this option to preserve parameter-state whilst entity editing.

Fullscreen Immersive-Mode : { Enabled | *Disabled }

→ Controls whether or not the application's window is rendered in fullscreen immersive mode (without the system bar displayed) or in standard windowed mode. Note: this option does not modify the display of the navigation bar. Enabling this option will increase the available screen-space that the mobile-editor can use (by the size of the system-bar) - relative to this option being disabled.

Session Persistence : { Never | *On-Destroy | Always }

→ Dictates the manner in which the application handles the persistent storage of editing session state between re-launches. This option can be used to control how the editor behaves whenever it is closed explicitly (using the quit-confirm dialog) or implicitly (either by the operating system - in order to free up memory - or as the result of a script triggered termination - or via an external application-manager).

Virtual Keyboard IME : { *Grammar-Centric | System-Default }

→ Dictates which IME (input-method-engine) is used by the virtual software keyboard in order to provide symbolic input to the procedural-editor. The grammar-centric IME provides an application-specific custom interface - whilst the system-default IME provides the standard interface.

Tactile Input-Feedback : { Global | Keyboard | *None }

→ Controls whether or not touch input events are accompanied by tactile device vibrations. The global option enables tactile feedback for all major discrete interface actions (including events such as menu-button taps). The keyboard option enables tactile feedback only for events stemming from the procedural-editor's keyboard. The none option disables tactile feedback altogether.

Library-Reference Structure : { *Symbol-Centric | Module-Centric }

→ Determines the arrangement of language-symbol entries in the library-reference's kernel documentation index list. Essentially this option dictates the order in which documentation entries are displayed. The symbol-centric option corresponds to the default alphabetical ordering which interleaves entries.

The module-centric option groups symbols by generative module such that related functions appear next to each other. Note: both the symbol-centric and module-centric orderings make exceptions for certain types of entry. For example boolean-operators are always placed at the head of the index whilst parameter-types, entity-types and kernel-directives are always located at the end of the index.

Editor Multi-Touch Interaction : { *Enabled | Disabled }

→ Determines whether or not the procedural-editor responds to multi-touch interaction events such as two-finger pinch-zoom and pan-move gestures.

Parallel Assembly Executor : { *Sequential | Simultaneous }

→ Controls the manner in which parallel-assemblies are processed. Note: this option applies exclusively to scripts containing the parallel-assembly kernel-directive at root-operative scope. When sequential is selected the processing of parallel-assembly sub-elements occurs in sequence (or series) one after another - such that at most one sub-element shall be being assembled at any one point in time. When simultaneous is selected the processing of parallel-assembly sub-elements occurs concurrently (or in tandem/parallel) such that two or more sub-elements may be being assembled at the same point in time. Note: irrespective of the parallel-assembly executor selected the results should be equivalent.

Parallel Assembly Culling : { *Enabled | Disabled }

→ Controls whether or not successive invocations of parallel-assemblies are coordinated with sub-element source-to-source equivalency-based execution-culling. When parallel-assembly culling is enabled sub-elements are only re-assembled between successive invocations if their source-definition blocks exhibit variance. When parallel-assembly culling is disabled all sub-elements are re-assembled between successive invocations irrespective of whether or not they exhibit source variations.

Performance Profiling : { Enabled | *Disabled }

→ Determines whether or not script execution is augmented to include performance-tracking breakpoints. Put simply this option can be used to turn performance profiling instrumentation on or off.

Multi-Sampling Anti-Aliasing : { *Enabled | Disabled }

→ Determines whether or not the geometry-viewer's 3D canvas uses multi-sampling to mitigate the effect of visual aliasing. Note: enabling MSAA may reduce the maximum framerate of the geometry-viewer's pinch-zoom and pan-orbit touch-input event rendering updates on older devices.

Gradient Backdrop Shading : { Linear | Radial | *None }

→ Dictates the backdrop rendering method employed by the geometry-viewer's 3D canvas.

Undo-Redo Persistence : { Never | *On-Destroy | Always }

→ Dictates the manner in which the application handles the persistent storage of edit-stack undo-redo state between re-launches. This option is tied to the related (but distinct) Session-Persistence option in that this option only has an effect in instances when an editing session persists over a re-launch.

***

To recap: the self-documenting options-&-settings tool-tab component houses a set of widgets which you will use to configure and control global (app-wide) state. Each option widget has a text-label, a button to display more information about the option (in the info-overlay) and a state-setting radio-group. Remember that the configuration does not (by default) persist over restarts of the mobile-editor.

5.9 Console Trace

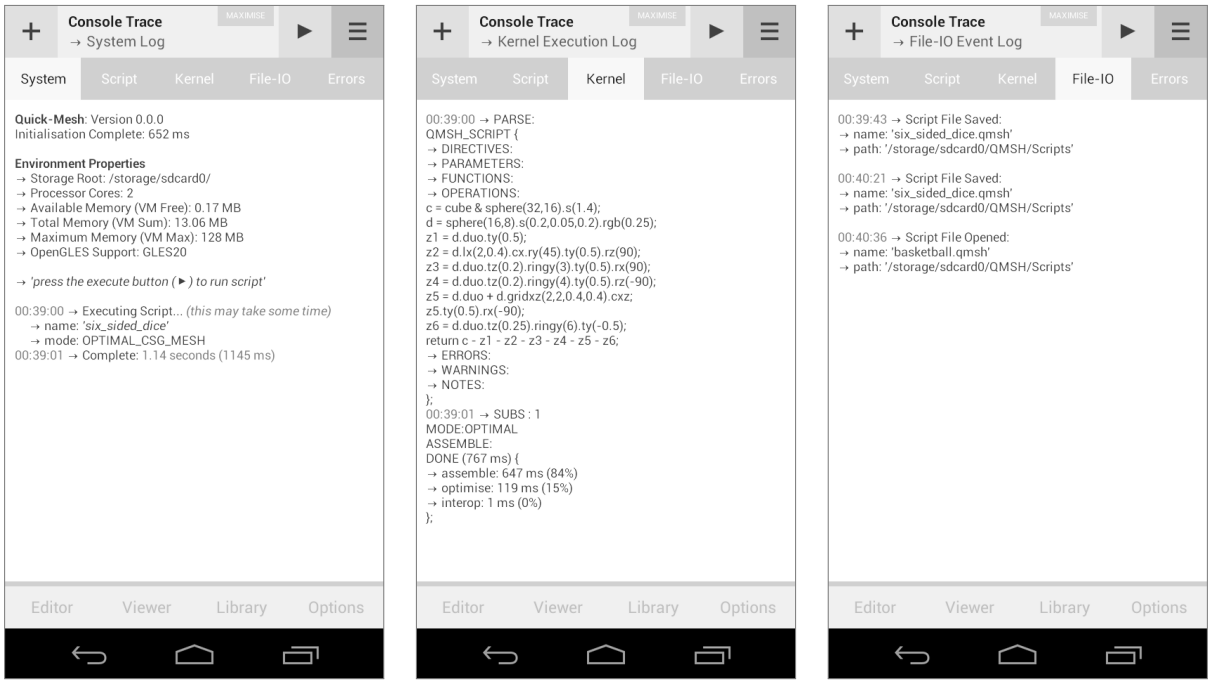


Figure 5.9: screenshots of the app's console-trace component - illustrating (from left to right): the appearance of the tool's system-log tab, its kernel-execution-log tab and its file-input-output-log tab.

The console-trace tool is the mobile-editor's debugging component - and manages a set of trace-logs.

Note: unlike the other key tools (that are managed by the content-tab-pane) - the console-trace tool-tab does not have a corresponding tab-control-bar item-button. In order to display and dismiss the console-trace - tap the currently selected tab-control-bar item. In this manner you can quickly toggle between the active tool-tab and the console-trace by repeatedly tapping the same item.

Note: use the tool's trace-selector-bar to switch between the different types of operation log exposed.

The set of trace-logs (managed by the console-trace) are outlined next for reference.

- **System-Log**: displays messages related to the general operation of the mobile-editor. In-particular this includes the start and end times for each assembly task coordinated by the kernel. Note: that the system-log is the default console-trace log displayed upon launch.
- **Script-Log**: displays messages resulting from invocation of the *print* function within scripts.
- **Kernel-Log**: displays messages produced by the kernel as a product of executing a task.
- **File-IO-Log**: displays messages related to input and output file operations - such as opening scripts from, and saving scripts to device storage - and exporting mesh interchange files.
- **Error-Log**: displays messages describing errors encountered in the operation of the mobile-editor.

6 Key Menus

This section details the behaviour of the the key menus that you shall interact with when using the app. It first outlines these menus and then provides instructions covering the use of each in turn.

- **New Script** - a modal windowed pop-up menu-dialog for creating a new script - either by cloning the current script (i.e. the *fork* operation) or by creating a new empty script.
- **Run Script** - a modal windowed pop-up menu-dialog for running the current script with the app's kernel (parsing and assembly) - with option widgets to set the kernel meshing execution-mode.
- **Open Script** - a modal full-screen overlay menu-dialog for opening scripts - either from the suite of grouped built-in example scripts or from device storage (i.e. internal-memory or SD-card).
- **Save Script** - a modal full-screen overlay menu-dialog for saving the currently active script to device storage as a .qmsh file for later editing.
- **Export Mesh** - a modal windowed pop-up menu-dialog for exporting the currently assembled entity to device storage as a 3D mesh interchange format file - with mesh attribute option widgets.

6.1 New Script Menu-Dialog

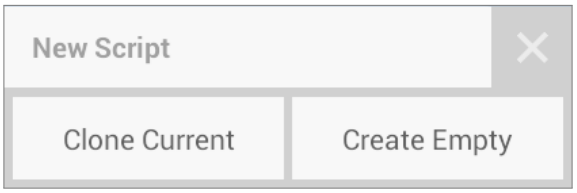


Figure 6.1: image of the new-script menu-dialog depicting its two action buttons which are used to create a new script by either cloning the current script's source or starting afresh with an empty untitled script.

The new-script menu-dialog is used (as its name suggests) to create new scripts.

It exposes two script creation actions - which are outlined below for reference.

- **Clone Current Script** : creates a new script by copying the contents of the currently active script. This action is roughly equivalent to a fork or branch operation. The name of the cloned script is the same as the source script with the postfix ' (copy)' applied.
- **Create Empty Script** : creates a new script by starting fresh - with empty and untitled script source. This action is analogous to beginning with a clean slate. The name of the created script is defined the first time the script is saved - up until that point it remains 'untitled'.

To recap: simply select one of the dialog's buttons to perform the corresponding new-script action.

6.2 Run Script Menu-Dialog

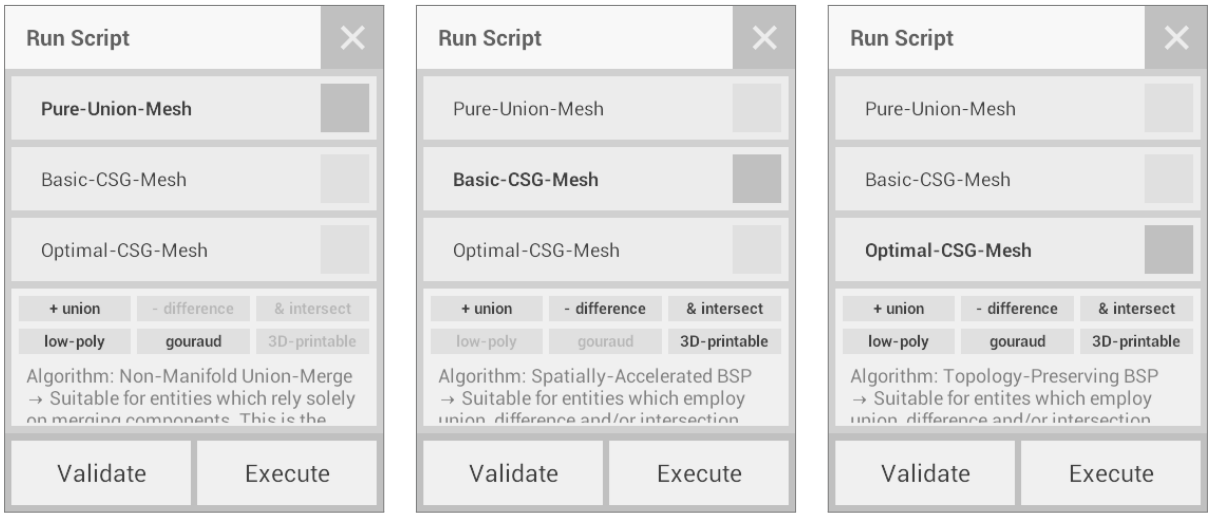


Figure 6.2a: images of the run-script menu-dialog illustrating: (from left to right) the appearance of the component when (respectively) the pure-union, basic-CSG and optimal-CSG kernel-execution meshing-modes are selected.

The run-script menu-dialog is used to invoke the quick-mesh kernel in order to assemble a 3D polyhedral entity (which is displayed by the geometry-viewer) from the currently active script's source-code.

As the new-script menu-dialog, the run-script menu-dialog's operation is relatively simple. Use the central-mode-pane to select a kernel-execution meshing-mode for running the active script (by tapping one of the items to select it). Then use the base-action-bar to initiate a kernel-execution task (again by tapping to trigger the action). Refer to figure 6.2a for further clarification.

Next - the supported kernel-execution meshing-modes and kernel-execution tasks are explained.

Kernel-Execution Meshing-Mode : { Pure-Union-Mesh | Basic-CSG-Mesh | *Optimal-CSG-Mesh }

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. The key aspects of the supported meshing-modes are stated below for reference.

Pure-Union-Mesh → *Algorithm: Non-Manifold Union-Merge*

→ Suitable for entities which rely solely on merging components. This is the fastest means to instantiate a mesh but is the least generalised - as it does not support difference or intersection operations. However it works well with parametric control. More clearly: entities for which a *geometrically-coherent* representation is attainable in pure-union meshing-mode are typically amenable to realtime (low-latency) parametric manipulation. This results from the overhead associated with assembly in pure-union mode being near negligible relative to the heavy-weight nature and associated expense of the basic-CSG and optimal-CSG modes. Note: pure-union compatible entities are considered a special subset of the infinite-set of possible entity-descriptors within the abstract space defined by the quick-mesh grammar.

Although they are incredibly useful for realtime applications they are typically less common and can be harder to formulate when the entity being represented includes constructive features.

Basic-CSG-Mesh → *Algorithm: Spatially-Accelerated BSP*

→ Suitable for entities which employ union, difference and/or intersection operations. Note: that whilst this is a generalised method - it is also heavy-weight and may not be suitable for interactive parametric control. Additionally this does not preserve the topology of the resulting mesh - i.e. it introduces T-junctions and causes faceting (the striping of shared vertex). This can be considered a baseline algorithm supporting the full set of boolean-logic operations via binary-space-partitioning - with conservative spatial accelerations and data-driven culling mechanisms. The mesh produced in this mode are capable of representing closed volumes, however the topology of each mesh-surface is sub-optimal - with redundancy present. This mode exists primarily as a convenience for debugging purposes - to provide a means to compare un-optimised mesh to optimised-mesh - and to help determine if any assembly errors are the product of constructive-solid operations or minimal-vertex topology optimisation.

Optimal-CSG-Mesh → *Algorithm: Topology-Preserving BSP*

→ Suitable for entities which employ union, difference and/or intersection operations. This is similar to the spatially-accelerated BSP, but yields mesh that minimise the number of vertices in the result - whilst simultaneously preserving smoothing-groups. This is the most advanced method supported (in terms of meshing performance) - and may not be suitable for interactive parametric control.

For reference the features supported by each meshing-mode are summarised in the table in figure 6.2b.

Meshing-Mode	Boolean Union +	Boolean Difference -	Boolean Intersect &	Low-Polygon Count	Gouraud Shadable	3D-Printable Volume
Pure-Union	✓	-	-	✓	✓	-
Basic-CSG	✓	✓	✓	-	-	✓
Optimal-CSG	✓	✓	✓	✓	✓	✓

Figure 6.2b: table summarising the behavioural features supported by each kernel-execution meshing-mode.

Kernel-Execution Task : { Validate | Execute }
The mobile-editor enables you to initiate two kernel-execution tasks to address parsing and assembly.

Validate (Parse) → *initiates a script grammatical error checking task*

→ Initiates a kernel-execution task that iterates over the source-code of the active script in order to check that each statement is valid and flag up grammatical and syntax errors that would interfere with assembly such that they may be corrected by the script author.

Execute (Assemble) → *initiates a script-to-3D-entity assembly task*

→ Initiates a kernel-execution task that assembles a polyhedral mesh using the source-code of the active script - and (on successful completion) loads the 3D entity into the geometry-viewer for examination.

To review: the run-script menu-dialog is used to initiate kernel-execution tasks. You will use it frequently to assemble 3D mesh and to check for errors whilst editing script source-code.

6.3 Open Script Menu-Dialog

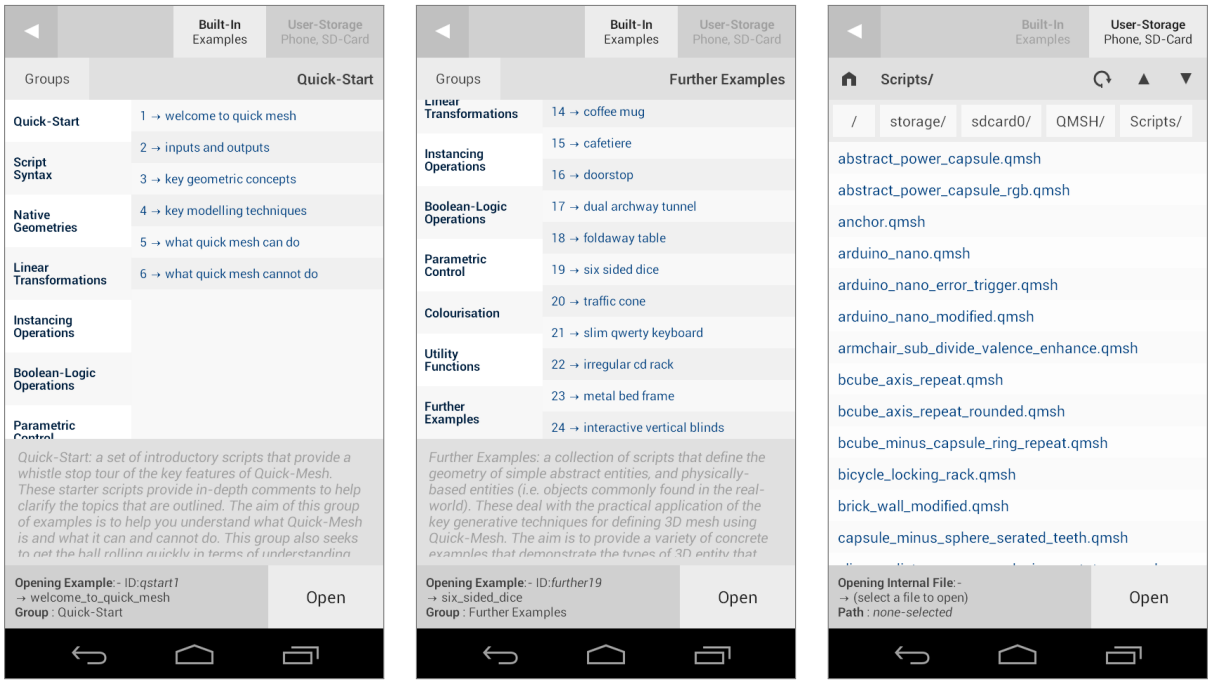


Figure 6.3: screenshots of the open-script menu-dialog - illustrating (from left to right): the appearance of the built-in example-browser in its default state with the *Quick-Start* example group active, then the example-browser with the *Further-Examples* group active and the *six-sided-dice* example script selected to open, and (finally) the user-storage script file-browser for opening script files stored in internal or external device memory.

The open-script dialog is used to open quick-mesh scripts for editing and assembly. It enables you to either open a script file stored in device memory - or to open a script from the suite of built-in examples.

In terms of its operation - use the open-dialog's titlebar to switch between opening an example-script or a script-file (see figure 6.3). Then navigate to and select the script you would like to open using the dialog's central-pane. Once you've selected a script the dialog's basebar displays the name and path (or group for examples) of the script to be opened (for confirmation) and you can select the dialog's open button to load the script into the procedural-editor which will update the project-manager.

Note: that if an instance of the selected script is already open in the project-manager then the open-dialog displays a notification explaining the conflict and switches to the open instance. If you intended to open the script afresh - then first close the open instance.

Note: the sole supported input file format for the mobile-editor's open-dialog is .qmsh.

To reiterate: the open-dialog is used to open built-in example scripts and script files in device memory.

6.4 Save Script Menu-Dialog

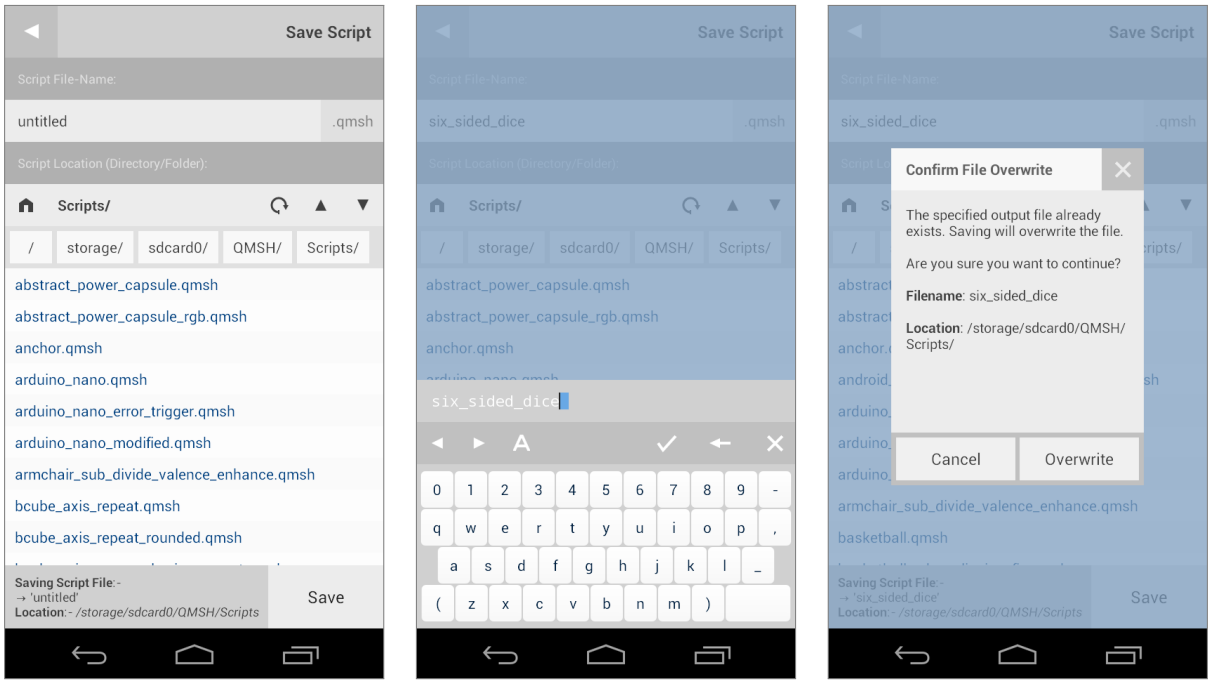


Figure 6.4: screenshots of the full-screen save-script menu-dialog (which is used to save .qmsh files to device storage) - illustrating (from left to right): the appearance of the script filename and file-path mutation components (i.e. the name field and the path, folder or directory browser), then the appearance of the *field-board* overlay (used to edit the output script's filename) and (finally) the appearance of the *confirm-file-overwrite* overlay dialog.

The save-script dialog is used to save the active quick-mesh script to device-storage as a .qmsh file. This enables you to pause and resume editing of a script and is vital to safe operation of the mobile-editor. In-particular in some cases the mobile-editor may encounter a critical-error during assembly which may cause the application's process to terminate. By using the save-script dialog frequently you minimise the risk of losing any of your unsaved edits in the event of an error, mistake or oversight.

The structure of the save-dialog's central pane is similar to that of the open-dialog's with the addition of a script file-name editing field. If you modify a script's file-name before saving you do not need to include the extension (.qmsh) as it is automatically appended. Refer to figure 6.4 for clarification.

You can change the path where a script is saved by navigating to a different location. The default path is the user-script location (.../QMSH/Scripts/). However you can save a script to any path for which you have permission to write. When you are ready select the save button to save a script. The save-dialog then issues an ephemeral notification to indicate when the save operation is complete.

Note: that if a script already exists at the specified path a confirm file overwrite dialog is displayed.

Note: the sole supported output file format for the mobile-editor's save-dialog is .qmsh.

To summarise: use the save-dialog as often as possible to save your scripts to device storage.

6.5 Export Mesh Menu-Dialog

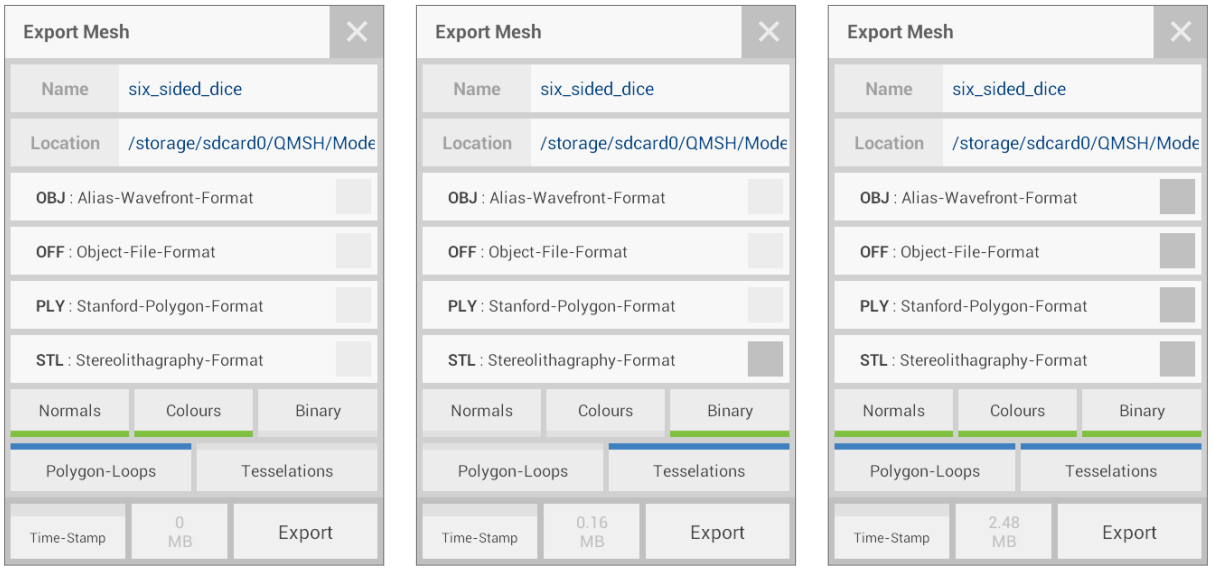


Figure 6.5: screenshots of the export-mesh menu-dialog (used to configure 3D file output) - illustrating (from left to right): the appearance of the default export configuration post-assembly of the six-sided-dice entity (without an interchange format specified), then the appearance of a binary STL export configuration and (finally) the appearance of a full-suite (i.e. all supported interchange formats with n-gon and triangle topology) export configuration.

The export-mesh dialog is the 3D file interchange component for the mobile-editor and is responsible for managing the creation of portable 3D mesh files that may be used in external applications, visualisations, simulations or as input to a 3D printer or similar additive-manufacturing-device.

The export-dialog's layout comprises a header pane, a central option configuration pane, and a footer control-bar. You can use the header to change the name of exported mesh, the central pane to control the manner in which mesh files are stored and the footer to initiate an export task (see figure 6.5).

Upon export - the configuration is checked and a notification displayed either instructing you to address any issues or allowing you to verify the files to be exported prior to running the task. As with the save-script menu-dialog, the export-dialog requires confirmation to overwrite any conflicting pre-existing files.

Note: the footer also includes a storage-space-estimator for the current export configuration (based on the size of the active entity) and a toggle to append a timestamp to each exported file's name.

The remaining portions of this section cover the supported options for configuring exported mesh.

Export-Mesh File-Format Options : { OBJ | OFF | PLY | STL }

OBJ → *Export 3D Mesh in Alias-Wavefront-Format*

An ASCII interchange format which supports triangle and N-gon mesh with colour and derivative data - and which is commonly used in interactive digital-media settings (such as to facilitate geometric asset-importing in game-engines and digital-content-creation systems) particularly for its ubiquitous support.

OFF → *Export 3D Mesh in Object-File-Format*

An ASCII or binary interchange format which supports triangle and N-gon mesh with colour and/or derivative data - and which is commonly used in research settings for its parsability. For compatibility reasons the mobile-editor only exports ASCII .off files.

PLY → *Export 3D Mesh in Stanford-Polygon-Format*

An ASCII or binary interchange format which supports triangle and N-gon mesh with colour and/or derivative data - and which is commonly used in research settings for it representational versatility.

STL → *Export 3D Mesh in Stereolithagraphy-Format*

An ASCII or binary triangle-mesh-only interchange format commonly used in 3D-printing settings. This format does not support entity colour data or the inclusion of Gouraud smooth-shading normals.

Note: for interchange formats that support both triangle and polygonal representations (OBJ, OFF and PLY) an additional topological indicator token is appended to each exported file's name to denote the type (or class) of reader necessary to open each file. These topological post-fix take the form *_TESS* (to indicate triangle elements) or *_NGON* (to indicate polygonal elements).

Export-Mesh Attribute Options : { *Normals | *Colours | Binary }

Normals → *Include Smooth-Shading Surface-Normal Data*

Includes per-vertex unit-length Gouraud surface-normals in applicable exported mesh files.

Colours → *Include Entity Colour Data*

Includes per-vertex RGBA (red, green, blue, alpha) colour data in applicable exported mesh files.

Binary → *Export Binary Data*

Exports binary-encoded mesh data instead of ASCII-encoded mesh data for PLY and STL outputs.

Export-Mesh Topological Options : { *Polygon-Loops | Tessellation }

Polygon-Loops → *Export Simple and Complex Polygon Faces*

Exports polygon-loop (N-gonal) mesh files for applicable interchange formats.

Tessellation → *Export Triangle Faces*

Exports triangulated mesh files for maximum compatibility with readers and loaders.

Note: the 3D mesh interchange files generated by the mobile-editor are typically (indeed almost always) substantially (orders-of-magnitude) larger than the scripts that define them.

In order not to exhaust device storage - save script files often - export 3D mesh files sparingly.

7 Additional Notes

Finally this section closes this quick-start guide with additional handy notes and useful tips.

This section first outlines some high-level pointers to steer your use of the app and then finishes with practical performance considerations that will help you to get the most out of the mobile-editor.

- **Think Before You Mesh!**
- **Save Your Scripts!**
- **If In Doubt → Read The Manual!**

7.1 Performance Considerations

Note: although computational performance is somewhat secondary to the representativeness of generated geometric arrangements (i.e. spatial and structural validity and coherence) - the following aspects of the mobile-editor (and its underlying kernel) should still be considered during the construction and editing of polyhedral entities - to guide ones decision making when multiple scripting approaches exist.

Boolean-Logic:-
The mobile-editor's boolean-logic (CSG) operations are simultaneously one of the most versatile features of the quick-mesh grammar and kernel - and the most computationally expensive to run (relative to the majority of the native operations). The secret to getting the most out of them is to use them wisely. When performance is a concern - aim to arrange the entities in your script so to minimise the number of boolean-operations required without the loss of geometric character. Profiling is key - for example if you are unsure which of a set of approaches is best - try them each and note their performance.

65K-Vertex-Threshold:-
The mobile-editor uses OpenGL-ES to render polyhedral-entities. The standard behaviour under the Android operating system is for mesh indices to be represented as 2-byte shorts - which means the maximum number of indexable vertices (in a single mesh that is pushed to the GPU) is 2^{16} (65536). If the number of vertices in a polyhedral entity generated by the kernel exceeds this threshold - then the entity is partitioned prior to being rendered. This partitioning is seamless (in the sense that its effect is imperceptible visually) - however it requires an extra step to coordinate which has the potential to bloat an entity's overall injection time. In simple terms: mesh with fewer than (approximately) 65K vertices are simpler and faster for the mobile-editor to process and display than mesh that exceed this threshold.

Parameter-Definitions:-
The mobile-editor's support for interactive parametric entity manipulation depends largely on the execution speed of a script. The faster a script executes the more responsive parametric mutations are. As such - if a script is suitable for the pure-union kernel-execution mode then you should aim to favour this mode for parameter changes. Alternatively (for faster-execution) try and reduce the number and complexity of boolean-logic operations in a script by taking advantage of symmetries, regularities, spatial-relations and variant native-geometries. Additionally - sometimes you can replace constructive features in a script with pure-union compatible natives such as revolutions, toroids, extrusions and profile-rails.

***

So ends this guide. I hope you enjoy using the Android mobile-editor as much as I enjoy making it.