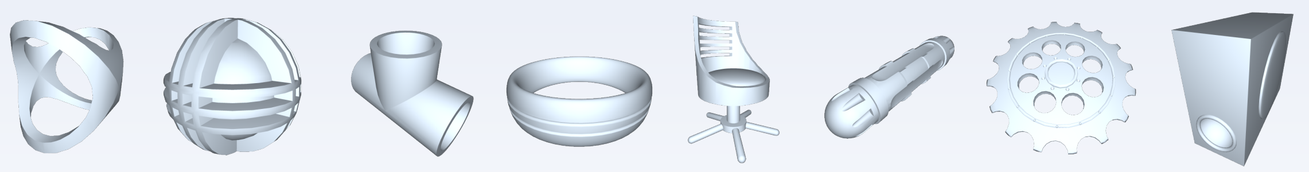
.qmsh
The Quick-Mesh Scripting Language



Aims, Objectives, Background, Context and Motivations

QMSH → A 3D Generative Shape-Grammar for the *'Procedural-Age-of-Man'*

Intuitive → a language that's easy to read, easy to write, easy to parse - a procedural delight...
Concise → succinct scripts, dependency-free streamlined kernel, with low-poly-count support...
Flexible → portable 3D logic, interoperable with DCC-suites, with versatile language constructs...

Grammar: Syntax, Constructs and Semantics

The Quick-Mesh scripting language is an imperative, closed-form, high-level, object-oriented, procedural geometric DSL - whose primary grammatical construct is the sequential expression chain effected with dot-notation semantics. It employs a declaration-order execution-model over bracketed expressions (or precedence-based symbolic operators). The grammar's syntax draws on the C-family of programming languages - adding constructs that help to reduce symbolic redundancy.

Types of Entity: Numeric | Literal | Geometric | Utility

Numeric-Types
I | INT → Integer
D | DBL → Double
L | LNG → Long
→ { 1, 2.3, 4L }

Literal-Types
B | BLN → Boolean
C | CHR → Character
S | STR → String
→ { true, 'k', "do" }

Geometric-Types
2D | SHP → Polygonal-Shape
→ { box, circle, arch }
3D | MSH → Polyhedral-Mesh
→ { cube, cylinder, torus }

Utility-Types
A[] | ARY → Array
→ Generic Entity-Arrays
→ Typesafe Data-Arrays
T | ET → Entity-Type

Types of Statement: PARAM-DEF | ASSIGN | ACTION | RETURN

Parameter Definition Statements - declare and define attributed public external control parameters that drive the modelling process - and (once defined) can only be changed (set) outside of a script.

Object Assignment Statements - specify private internal variables which store intermediary modelling state that is only accessible within a script - and (vitally) are specified without a type-name.

General Action Statements - are used to explicitly invoke functions and operations that do not alter object references (i.e. inline) - such as setting the colour of an object or translating an object.

The Return Statement - tells a kernel's geometric assembler which dynamically generated object should be output as the result of evaluating a script - e.g. these statements specify the return value.

Classes of Operations: Inline | Instantiative | Utility

PRIMITIVE TRANSFORM INSTANCING BOOLEAN MODIFIER MEASURE COLOUR UTILITY

3D Modelling Techniques Supported by the Grammar

→ **Parametric-Modelling** : using parametric-primitives and user-defined control-parameters.
→ **Generalised-Cylinders** : using extrusions, revolutions and rails (swept-profiles).
→ **Constructive-Solid-Geometry** : using boolean-logic on polyhedral volumes.

PF-DN-SEC: Post-Fix Dot-Notation Sequential-Expression-Chains

→ **PF-DN-SEC**s : *"method-chaining on steroids"* → **DOIEO** : Declaration-Order-IS-Execution-Order!

Flexible Functions: Flexible Invocation and Flexible Definition

FFI: Flexible-Function-Invocation: Syntax Sweeteners for Calling Native Functions
→ Pre-Fix Modifier-Symbols (Concise Polyhedral-Primitive Revisions and Instancing-Mode Control)
→ Post-Fix Axis-Specifiers (Simultaneously Remove Ambiguity and Redundancy in Scripts)
→ AAP: Automatic-Action-Promotion (Optional Parenthesis for Calling Zero-Arg. Actions)
→ AAL: Automatic-Argument-Linearisation (Argument-Re-Use/Re-Cycling amongst Invocations)
→ Case Insensitivity and Clarifying Underscores (Accommodate Varying Expressive Styles)
→ Synonym-Support: Native Shorthands and Aliasing (suit both Compactification and Exposition)

FFD: Flexible-Function-Definition: Typing-Discipline-Agnostic User-Defined Functions
→ Free(dom) Functions: Pure, Abstract, Dynamically-Typed, High-Level Function Definitions
→ Typed Functions: Conventional (C-style) Statically-Typed Function Signature Definitions
→ Bounded Functions: Constrained Function Definitions - More Stringent Checking on Input Args.
→ Hybrid Functions: Combine (Pix-and-Mix) Varying Typing-Discipline Constructs in Signatures
→ Nested Functions: Temporary Functions Bound to an Operative Scope (great for Encapsulation)

3D Geometric Primitives

3D Platonic Solids

cube | cuboid | box3D creates an axis-aligned cube
→ (**D**:size);
→ (**D**:width, **D**:height, **D**:depth);
→ (**D**:x, **D**:y, **D**:z, **D**:width, **D**:height, **D**:depth);

tetrahedron | m_tetrahedron → (**D**:size);
octahedron | m_octahedron → (**D**:size);
dodecahedron | m_dodecahedron → (**D**:size);
icosahedron | m_icosahedron → (**D**:size);

3D Axis-of-Revolution Solids

sphere creates a polar-pointed smooth uv-sphere
→ (**D**:diameter);
→ (**I**:slices, **I**:stacks);
→ (**D**:diameter, **I**:slices, **I**:stacks);

cone → (**D**:radius, **D**:length, **I**:segments);
cylinder → (**D**:radius, **D**:length, **I**:segments);

cylicone creates a non-uniform cylindrical-cone
→ (**D**:lower_rad, **D**:upper_rad, **D**:length, **I**:segments);

torus creates a torus from major and minor circles
→ (**D**:maj_rad, **D**:min_rad, **I**:maj_steps, **I**:min_steps);

capsule creates a parameterised capsule
→ (**D**:radius, **D**:length, **I**:segments, **I**:steps);

ellipsoid creates a non-uniformly scaled sphere
→ (**D**:width, **D**:height, **D**:depth);
→ (**D**:width, **D**:height, **D**:depth, **I**:slices, **I**:stacks);

3D Fixed-Form Solids

pyramid | **gable** | **wedge** | **diamond** | **bracket**

Colourisation

User-Driven Colourisation

RGB | RGBA sets red-green-blue (alpha) colour
→ (**D**:intensity); *...for grayscale - note: all colourisation functions expect decimals in the range 0.0 to 1.0*
→ (**D**:red, **D**:green, **D**:blue);
→ (**D**:red, **D**:green, **D**:blue, **D**:alpha);

HSB | HSBA sets hue-saturation-brightness colour
→ (**D**:hue); *... decimals in range [0.0:1.0]*
→ (**D**:hue, **D**:saturation, **D**:brightness);
→ (**D**:hue, **D**:saturation, **D**:brightness, **D**:alpha);

A | alpha sets alpha-transparency (translucency)
→ (**D**:alpha); *... decimals in range [0.0:1.0]*

Euclidean Transformations

Transform-Grammar Keycodes

The QMSH Transform-Grammar exposes absolute (explicit) and relative (implicit) transformations which can be optionally invoked using succinct keycode alias.
→ These ops also support post-fix axis-specifiers (swizzling).

T = translate
S = scale
R = rotate
I = invert
C = center
Z = positive-zero
NZ = negative-zero

Explicit (Absolute) Transforms

T | translate moves (offsets the position of) an entity
t | txyz → (**D**:dx, **D**:dy, **D**:dz);
tx → (**D**:distance_x); *...translate along the X-axis*
ty → (**D**:distance_y); *...translate along the Y-axis*
tz → (**D**:distance_z); *...translate along the Z-axis*

S | scale resizes (alters the extents of) an entity
s | sxyz → (**D**:uniform_scale_factor);
s | sxyz → (**D**:sf_x, **D**:sf_y, **D**:sf_z);
sx → (**D**:scale_factor_x); *...scale along the X-axis*
sy → (**D**:scale_factor_y); *...scale along the Y-axis*
sz → (**D**:scale_factor_z); *...scale along the Z-axis*

R | rotate performs a euler-rotation on an entity
rx → (**D**:degrees_x); *...rotate about the X-axis*
ry → (**D**:degrees_y); *...rotate about the Y-axis*
rz → (**D**:degrees_z); *...rotate about the Z-axis*

I | invert | mirror | reflect mirrors an entity about an axis such that its vertex positions are inverted/negated/reflected - and re-orients the entity's faces
ix | iy | iz → ( ); *...invert along X, Y or Z axis*

Implicit (Relative) Transforms

C | center translates an entity such that it is centered at the origin - i.e. its midpoint coincides with (is at) 0
c | cxyz → ( ); *...center along X, Y and Z axes*
cx | cy | cz → ( ); *...center along X, Y or Z axis*

Z | zero translates an entity such that it begins at the origin - i.e. its minimum-point coincides with (is at) 0
z | zxyz → ( ); *...zero along X, Y and Z axes*
zx | zy | zz → ( ); *...zero along X, Y or Z axis*

NZ | nzero | negative_zero translates an entity such that it ends at the origin - i.e. its maximum-point coincides with 0 - this is the dual of the zero transform
nz | nzxyz → ( ); *...negative-zero X, Y and Z axes*
nzx | nzy | nzz → ( ); *...negative-zero X, Y or Z*

W | width scales an entity to set its aabb width
→ (**D**:new_width); *...autoscales to match width*

H | height scales an entity to set its aabb height
→ (**D**:new_height); *...autoscales to match height*

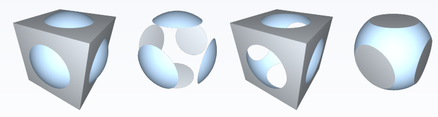
D | depth scales an entity to set its aabb depth
→ (**D**:new_depth); *...autoscales to match depth*

Post-Fix Axis-Specifiers (Swizzling)

Unary → [X, Y, Z]
Binary → [XY, XZ, YX, YZ, ZX, ZY]
Ternary → [XYZ, XZY, YXZ, YZX, ZXY, ZYX]
Exact-Angles → [90, 180, 270, PI, N90]

Constructive Modelling

CSG Boolean-Logic Operations

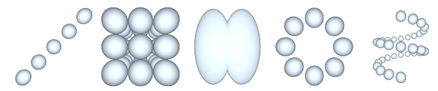


a + b **a - b** **b - a** **a & b**

+ | **union** | add | or | plus
- | **difference** | minus | not | subtract
& | **intersect** | and | intersection
% | **xor** | symmetric_difference | exclusive_disjunction

Instancing Operations

Structured Repetition



linear **grid** **butterfly** **ring** **spiral**

Cartesian Arrangements

L | linear_repeat instances an entity along a line
l | lxyz → (**I**:count, **D**:dx, **D**:dy, **D**:dz); *...along a line*
lx → (**I**:count, **D**:offset_x); *...repeat along X-axis*
ly → (**I**:count, **D**:offset_y); *...repeat along Y-axis*
lz → (**I**:count, **D**:offset_z); *...repeat along Z-axis*

G | grid | cartesian_grid_repeat instances an entity in a regularly spaced 2D or 3D grid arrangement
g | gxyz → (**I**:x, **I**:y, **I**:z, **D**:dx, **D**:dy, **D**:dz); *...3D grid*
gxy → (**I**:x, **I**:y, **D**:dx, **D**:dy); *...2D grid in XY-plane*
gxz → (**I**:x, **I**:z, **D**:dx, **D**:dz); *...2D grid in XZ-plane*
gyz → (**I**:y, **I**:z, **D**:dy, **D**:dz); *...2D grid in YZ-plane*

bfly | butterfly | butterfly_repeat instances an entity by a reflection operation (mirroring) along an axis
bflyx | bflyy | bflyz → ( ); *...butterfly in X, Y or Z axis*

Polar Arrangements

ring | radial_ring_repeat instances an entity in a rotary ring (i.e. incremental rotation) about a principal axis
ringx | ringy | ringz → (**I**:count);
ringx | ringy | ringz
→ (**I**:count, **D**:start_degrees, **D**:end_degrees);

spi | spiral | axis_spiral_repeat
spix | spiy | spiz
→ (**I**:count, **D**:angle_offset, **D**:distance_offset);

Data-Driven Arrangements

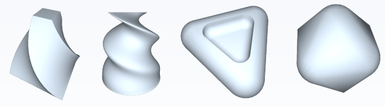
inst | instance | general_instance_repeat
instx | insty | instz → (**D**[]:offsets...);

Pre-Fix Mode-Control Symbols

_ | auto → *automatic merge (the default behaviour)*
f | fast → *merge without modifying topology*
u | union → *merge with boolean union operation*

Modifications and Utilities

Utility Toolbox Operations



Hulls and Envelopes

CH | cnvx | convex | convex_hull constructs a faceted convex hull or convex envelope (i.e. crisp edges)
ch | chxyz → ( ); *...faceted hull in X, Y and Z*
chxy | chxz | chyz → ( ); *...faceted envelopes*

GCH | gcnvx | gconvex | gouraud_convex_hull constructs a smooth-shaded (i.e. shared-vertex/manifold) convex hull or convex envelope
gch | gchxyz → ( ); *...smooth hull in X, Y and Z*
gchxy | gchxz | gchyz → ( ); *...smooth envelopes*

Deformation Modifiers

tap | taper deforms an entity via a tapering operation applied linearly along an axis from aabb min to max
tapx → (**D**:min_sf_x, **D**:max_sf_x); *...taper along X*
tapy → (**D**:min_sf_y, **D**:max_sf_y); *...taper along Y*
tapz → (**D**:min_sf_z, **D**:max_sf_z); *...taper along Z*
tapx → (**D**:min_y, **D**:min_z, **D**:max_y, **D**:max_z);
tapy → (**D**:min_x, **D**:min_z, **D**:max_x, **D**:max_z);
tapz → (**D**:min_x, **D**:min_y, **D**:max_x, **D**:max_y);

twst | twist deforms an entity by a rotational twisting operation along an axis from aabb min to max
twstx → (**D**:min_revs_x, **D**:max_revs_x); *...along X*
twsty → (**D**:min_revs_y, **D**:max_revs_y); *...along Y*
twstz → (**D**:min_revs_z, **D**:max_revs_z); *...along Z*

shr | shear applies an affine (linear) shearing transform to an entity along an axis from aabb min to max
shrx → (**D**:min_dy, **D**:min_dz, **D**:max_dy, **D**:max_dz);
shry → (**D**:min_dx, **D**:min_dz, **D**:max_dx, **D**:max_dz);
shrz → (**D**:min_dx, **D**:min_dy, **D**:max_dx, **D**:max_dy);

Utility Functions

print → (**S**:message); *...print string to the console*

duo | duplicate | clone → (**3D**:operand);

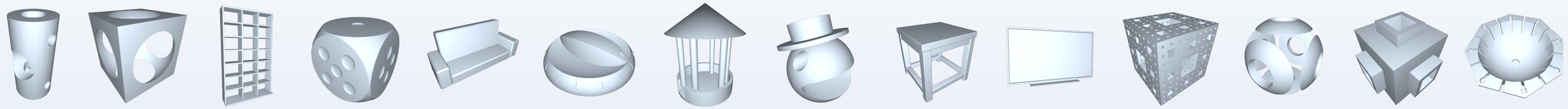
NMM | merge | non_manifold_merge a fast merge
→ (**3D**:lhs, **3D**:rhs); *...binary overload*
→ (**3D**[]:operands); *...variadic overload*

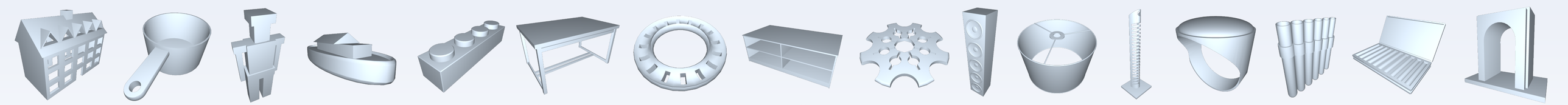
Measurements and Queries

min | minimum compute an entity's aabb minimum
max | maximum compute an entity's aabb maximum
mid | middle compute an entity's aabb mid-point
avg | average | mean compute geometric mean
span | extent compute an entity's aabb extent
minx | maxx | midx | avgx | spanx → ( );
miny | maxy | midy | avgy | spany → ( );
minz | maxz | midz | avgz | spanz → ( );

area | surface_area computes a shape's area in R2
vol | volume computes a solid's volume in R3

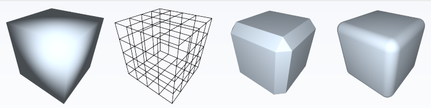
QMSH
Quick-Mesh
Grammar:V0
→ Reference-Sheet
K. Edum-Fotwe
Codemine-Ind.





3D Extended Primitives

Modified Native Geometries (Variants)



m-cube | **s-cube** | **b-cube** | **r-cube**

Native geometries such as the cuboid and cylinder expose structural and topological variants. These modified natives are expressed using symbolic prefix and (when applicable) supplementary arguments.

Pre-Fix Modifier Symbol Key

Character	Full-Name	Meaning-in-Grammar
B b	bevel	beveled (chamfer) edges
R r	round	rounded edges
S s	subd	bi-linearly sub-divided
C c	cap	capped (omit poles)
P p	polar	add polar-points
O o	open	open (non-cyclic)
M m	manifold	share-vertex (platonics)
G g	gouraud	smooth-shade (for hulls)

Note: pre-fix modifier symbols are not applicable to all native geometries - so check before use.

3D Polar-Pointed Variant Solids

pcone | **pcylinder** | **pcylicone** each accept the same input arguments as their respective non-polar variants

3D Polar-Capped Variant Solids

csphere | **ccapsule** | **cellipsoid** accept the same arguments as their respective un-capped variants

3D Bi-Linearly Sub-Divided Solids

octo_cube creates an octant-split (2x2x2) cube - this is a shorthand convenience for the scube function - and supports the same input arguments as the cube

scube creates a bi-linearly sub-divided cuboid
→ (**D**:size, **I**:ncount);
→ (**I**:xcount, **I**:ycount, **I**:zcount);
→ (**D**:width, **D**:height, **D**:depth, **I**:ncount);
→ (**D**:width, **D**:height, **D**:depth, **I**:xcount, **I**:ycount, **I**:zcount);
→ (**D**:x, **D**:y, **D**:z, **D**:width, **D**:height, **D**:depth, **I**:xcount, **I**:ycount, **I**:zcount);

scone → (**I**:xcount, **I**:ycount);
→ (**D**:radius, **D**:length, **I**:segs, **I**:xcount, **I**:ycount);
scylinder → (**I**:xcount, **I**:ycount);
→ (**D**:radius, **D**:length, **I**:segs, **I**:xcount, **I**:ycount);

scylicone → (**D**:lower_radius, **D**:upper_radius, **D**:length, **I**:segments, **I**:lower_xcount, **I**:upper_xcount, **I**:ycount);

scapsule → (**D**:radius, **D**:length, **I**:segments, **I**:steps, **I**:sub_divide_count);

3D Extended Primitives (cont.)

3D Beveled/Chamfered Solids

bcube creates a beveled/chamfered cuboid
→ (**D**:size, **D**:bevel);
→ (**D**:width, **D**:height, **D**:depth, **D**:bevel);
→ (**D**:x, **D**:y, **D**:z, **D**:w, **D**:h, **D**:d, **D**:bevel);

bcone → (**D**:radius, **D**:length, **I**:segments, **D**:bevel);
bcylinder → (**D**:rad, **D**:length, **I**:segments, **D**:bevel);

bcylicone → (**D**:lower_rad, **D**:upper_rad, **D**:length, **I**:segments, **D**:bevel);

bpyramid | **bgable** | **bwedge** | **bdiamond** | **bbracket**
→ (**D**:size, **D**:bevel);

3D Rounded Solids

rcube creates a rounded (smooth) cuboid
→ (**D**:size, **D**:radius, **I**:steps);
→ (**D**:width, **D**:height, **D**:depth, **D**:radius, **I**:steps);
→ (**D**:x, **D**:y, **D**:z, **D**:width, **D**:height, **D**:depth, **D**:round_radius, **I**:round_steps);

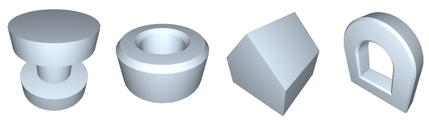
rcone → (**D**:radius, **D**:length, **D**:segments, **D**:round_radius, **I**:round_steps);
rcylinder → (**D**:radius, **D**:length, **D**:segments, **D**:round_radius, **I**:round_steps);

rcylicone → (**D**:lower_rad, **D**:upper_rad, **D**:length, **I**:segments, **D**:round_radius, **D**:round_steps);

rpyramid | **rgable** | **rwedge** | **rdiamond** | **rbracket**
→ (**D**:size, **D**:round_radius, **I**:round_steps);

3D Generalised Cylinders

Extrusions, Revolutions and Rails



3D Extrusion Solids

extrude_x | **extrude_y** | **extrude_z** a linear extrusion
→ (**2D**:shape, **D**:length);
→ (**2D**:shape, **D**:min, **D**:max);

3D Revolution Solids

revolve | **prevolve** creates a revolution about Y-axis
→ (**2D**:open_half_profile_shape);
→ (**2D**:open_half_profile_shape, **I**:segments);

toroid creates a toroidal (full-profile-revolved) solid
→ (**2D**:closed_profile_shape);
→ (**2D**:closed_profile_shape, **I**:segments);

3D Profile-Rail Solids

rail_xy | **rail_xz** | **rail_yz** creates cyclic (closed) rails
→ (**2D**:profile, **2D**:cylic_rail);

orail_xy | **orail_xz** | **orail_yz** for non-cyclic (open) rails
→ (**2D**:profile, **2D**:open_rail);

2D Geometric Primitives

2D Rectangular Polygon Shapes

box | box2D | rect | rectangle creates a 2D box
→ (**D**:size);
→ (**D**:width, **D**:height);
→ (**D**:x, **D**:y, **D**:width, **D**:height);

bbox | bbox2D | brect | brectangle a 2D beveled-box
→ (**D**:size, **D**:bevel);
→ (**D**:width, **D**:height, **D**:bevel);
→ (**D**:x, **D**:y, **D**:width, **D**:height, **D**:bevel);

rbox | rbox2D | rrect | rrectangle a 2D rounded-box
→ (**D**:size, **D**:radius, **I**:steps);
→ (**D**:width, **D**:height, **D**:radius, **I**:steps);
→ (**D**:x, **D**:y, **D**:width, **D**:height, **D**:radius, **I**:steps);

2D Circular Polygon Shapes

circle creates a circular polygon shape
→ (**D**:radius);
→ (**D**:radius, **I**:segments);
→ (**D**:ox, **D**:oy, **D**:radius, **I**:segments);

oval | ellipse creates an axis-aligned squashed circle
→ (**D**:width, **D**:height);
→ (**D**:width, **D**:height, **I**:segments);
→ (**D**:x, **D**:y, **D**:width, **D**:height, **I**:segments);

arc creates a closed circular arc shape
→ (**D**:radius, **D**:steps, **D**:start_angle, **D**:end_angle);
pie creates a circular pie segment shape
→ (**D**:radius, **D**:steps, **D**:start_angle, **D**:end_angle);

arch | archway creates a closed circular arch shape
→ (**D**:radius, **D**:length);
→ (**D**:radius, **D**:length, **I**:steps);
→ (**D**:radius, **D**:y, **D**:length, **I**:steps);

cap2D | capsule2D creates a circular capsule shape
→ (**D**:radius, **D**:length);
→ (**D**:radius, **D**:length, **I**:steps);
→ (**D**:radius, **D**:x1, **D**:y1, **D**:x2, **D**:y2, **I**:steps);

2D Common Polygon Shapes

tri | triangle creates an equilateral or custom triangle
→ (**D**:radius); *...equilateral and centered at the origin*
→ (**D**:ax, **D**:ay, **D**:bx, **D**:by, **D**:cx, **D**:cy); *...custom*

quad | quadrilateral creates a custom quadrilateral
→ (**D**:ax, **D**:ay, **D**:bx, **D**:by, **D**:cx, **D**:cy, **D**:dx, **D**:dy);

pgram | parallelogram creates a parallelogram from 3 points which define the vertices of the first two-edges
→ (**D**:ax, **D**:ay, **D**:bx, **D**:by, **D**:cx, **D**:cy);

square | **pentagon** | **hexagon** | **septagon** | **octagon** | **nonagon** | **decagon** | reg[4:10] creates a named 2D regular polygon shape centered at the origin - defined by a radius or an initial vertex
→ (**D**:radius); *...define by radius*
→ (**D**:vx, **D**:vy); *...define by initial-vertex*

reg | regular | regular_polygon creates an N-sided 2D regular polygon centered at the origin
→ (**I**:sides); *...define by N with default radius (0.5)*
→ (**D**:radius, **I**:sides); *...define by N and radius*
→ (**D**:vx, **D**:vy, **I**:sides); ...define by N and a vertex

2D Geometric Primitives (cont.)

2D Data-Driven Polygon Shapes

poly | polygon creates a polygon shape
→ (**D**[]:xy_coordinates);

bpoly | bpolygon creates a beveled-polygon shape
→ (**D**:bevel, **D**[]:xy_coordinates);

rpoly | rpolygon creates a rounded-polygon shape
→ (**D**:radius, **I**:steps, **D**[]:xy_coordinates);

2D Parametric Path Shapes

line creates a bounded line defined by two vertex
→ (**D**:x1, **D**:y1, **D**:x2, **D**:y2);

oarc creates an open circular arc path
→ (**D**:radius, **D**:steps, **D**:start_angle, **D**:end_angle);

oarch | oarchway creates an open circular arch path
→ (**D**:radius, **D**:length);
→ (**D**:radius, **D**:length, **I**:steps);
→ (**D**:radius, **D**:y, **D**:length, **I**:steps);

2D Data-Driven Path Shapes

path | polyline creates a polyline path
→ (**D**[]:xy_coordinates);

bpath | bpolyline creates a beveled-polyline path
→ (**D**:bevel, **D**[]:xy_coordinates);

rpath | rpolyline creates a rounded-polyline path
→ (**D**:radius, **I**:steps, **D**[]:xy_coordinates);

Parametric Control

Dynamic Entity Descriptors

→ Parameter definition statements specify the inputs (or the control-arguments) to a quick-mesh script.
→ Use them to express dynamic entity behaviours.
→ Remember: parameters are immutable in a script.
→ They occur exclusively at root-operative-scope.
→ Remember: parameter definitions must only be composed of non-referential constants and literals.

Unconstrained Parameters

INT | **DBL** | **LNG** | **BLN** | **CHR** | **STR**
→ T name = init;
→ T name (T:init, S:label);

Constrained Parameters

INT | **DBL** | **LNG** (...numeric types only)
→ T name (T:init, T:min, T:max);
→ T name (T:init, T:min, T:max, S:label);

Tips and Advice

Key Things to Remember

→ **DOIEO** - Declaration-Order-IS-Execution-Order!
→ Think Before you Mesh! → Save your Scripts!
→ Vitally: If in Doubt - Check the Documentation!

Applications, Use-Cases, Strengths & Limitations

Practical Examples: Abstract and Physically-Based QMSH Entities

Figure 1: examples of 3D entities defined with the QMSH grammar - illustrating (from left to right, top-down): office-chair, slim-qwerty-keyboard, electric-keyboard, keyring-extended-variant-with-laser-diode, dual-core-wireless-micro-controller-breakout, surge-protector, silver-coin, electric-cooker-top, plastic-two-tone-usb-pen-drive-without-lid, simple-helicopter, treasure-chest-light, kitchen-faucet, modified-capsule-difference, spiral-spring, spiral-screw-continuous-pair-modified, time-of-flight-distance-sensor -breakout, simple-spiral-staircase-extended, cupboard-door-hook, portable-audio-player-sd-card and hardback-book.

Advantages & Key-Benefits vs. Disadvantages & Unsupported-Features

→ Low-Polygon-Count 3D Model Generation (great for Realtime Systems, Games & Environments)
→ Purely Procedural General-Purpose 3D Content (great for High-Compression Scene Description)
→ Dynamic Entity Descriptors (great for LOD-ing, Object-Families and Behavioural-Modelling)
→ Excellent for Man-Made Structures (Electronics, Architecture, Mechanisms, Fixtures, Furniture)
→ Not-Great/Terrible for Organic Structures (Terrain, Vegetation, Creatures and Living-Organisms)
→ Does NOT Generate UV (automatic UV-unwrapping or manual texture-mapping may be required)
→ Does NOT Support Sub-Division or Free-Form Surface Modelling, Fractals or L-Systems

Puzzle-Time: Test Your Understanding

Match the following QMSH Scripts to the 3D Entities they Define

Each of the following quick-mesh scripts is illustrated in one of the images in this document's headers and footers. See if you can find the corresponding image for each script to test how well you've absorbed the main aspects of the grammar. Note: the first three should be relatively easy.

```
1: return bcube(0.25,0.1,1,0.01).l(8,0.125,0.1,0);
2: return cube - sphere(1.25);
3: return torus(0.01,0.001,64,32).sy(3) - torus(0.011,0.00025,64,16).ly(2,0.00125).cy;
4: f = cylinder(0.25,2,32) - cylinder(0.2,2,32); return f & f.duo.rx(90);
5: cube - octahedron(1.5) & octahedron(2.0);
6: scube(1,5,1,10,50,10).twistY(0,1);
7: a = torus(1,0.25); return a - cube(0.125,0.5,0.25).tz(1).ringy(16) - a.ty(0.25);
```

Note: if you get stuck you can assemble the mesh defined by each script using the QMSH-Kernel.

Further Information, Tools and Resources

Reference Implementations and Project Home

For more info, news and to download the reference implementations of the QMSH-Kernel for Linux, Mac-OS and Windows operating systems - head to: `https://www.qmsh.org` (the project's online home). There you'll also find links to the QMSH-Editor for Android and the QMSH-Plugin for Unity.

Documentation and Quick-Start Guides

Official documentation for the QMSH-Grammar is available at: `https://www.qmsh.org/doc.html`. There is also a suite of PDF quick-start guides (to accompany this reference-sheet) covering the kernel, editor and plugin available on the project's homepage (see the Documentation section).