Introduction to the Generative Modeling Language

Sven Havemann

Institut für ComputerGraphik,
TU Braunschweig, Germany

s.havemann@tu-bs.de
http://graphics.tu-bs.de/genmod

Version 1.31, June 2003

Abstract

This document is supposed as a tutorial to gain some hands-on experience in how to assemble procedural shapes with the GML.

1 Introduction

The Generative Modeling Language (GML) is a very simple stack-based language. Its main purpose is to serve as a low-level language for the description of procedural shapes. The GML differs from other low-level shape representations in that a shape is understood as the result of a *sequence of operations* instead of just a bunch of geometric primitives like triangles. Triangles are just the *result* of a modeling process, and typically consume much more space than the sequence of operations needed to obtain this result. The tremendous increase of processing speed makes it possible today to generate huge amounts of geometry information only *on demand*.

The GML is based on the core of Adobe's PostScript language. It doesn't have PostScript's extensive set of operations for typesetting, though. The GML is targeted instead at 3D modeling, and exposes the functionality from a C++ library for mesh construction to a stack-based interpreter. It has many operations from vector algebra and to handle polygons, and others that convert back and forth between polygons and meshes.

While it is standard that 3D modeling packages have a built-in scripting engine, the language concept of PostScript has a number of unique features. It seems that many operations that are frequently used with 3D modeling can be conveniently described with a stack-based language, and a number of concepts from modeling nicely map to GML constructs.

The idea behind the GML is to enable *automatized* 3D modeling. The goal of the GML architecture is to facilitate the composition of higher-level tools for 3D modeling out of simpler or elementary ones. This tutorial will show how shapes can be assembled using the integrated development environment. Such GML programs can efficiently realize flexible, domain-specific modeling tools, and are typically created by advanced users.

This is not the end of the story, though: PostScript programs can be used to assemble new PostScript programs! And the GML also contains operators to react on 3D input events by calling arbitrary callback functions. Combining both features, true 3D modeling without programming will be possible, and this is the next step on our roadmap.

This document is a hands-on tutorial that focuses on how to *use* the GML. A little background material on the PostScript basis is given in the next section, but interested readers should definitely have a look at the PostScript Redbook. But those who are keen on examples can skip the next section and continue with section 3 where the IDE is explained.

2 PostScript

The GML is based on the PostScript programming language from Adobe Inc. as consicely specified in section 3 of the PostScript Language Reference, also called the *Redbook*. It is freely avaible on the internet from www.adobe.com. The following sections in particular apply also to the GML (on pages 23-56 and 106-114):

- 3.1 Interpreter
- 3.2 Syntax
- 3.3 Data Types and Objects
- 3.4 Stacks
- 3.5 Execution
- 3.6 Overview of Basic Operators
- 3.10 Functions

The GML's memory management and error handling differ from the original PostScript, and there is no file IO or binary encoding (yet).

The great thing about PostScript as a full programming language is its *simplicity*. It is really easy to write a PostScript interpreter when you have only read those 40 pages of specification. This simplicity stems from the fact that PostScript differs from most other languages in that it has no parser, but only a lexical scanner – in Unix terms, no yacc is needed, but only lex. We will use the term *tokenizer* for it.

The PostScript interpreter does not actually execute a program, it merely consumes tokens and executes them in-

dividually. Tokens come in two classes: *literal* and *executable* tokens. To execute a literal token simply means to put it on the stack.

There are basically three kinds of literal tokens:

- Atomic values: Integers, Floats, Strings, 2D/3D vectors, markers, and names
- Arrays of atomic values
- Dictionaries of key/value pairs where the key is a name, and the value is a (literal or executable) token

There are basically three kinds of executable tokens:

- Executable arrays
- Executable names
- Operators

The job of the tokenizer is just to convert a character string into an array of tokens. As an implementation detail, tokens have a fixed size of 16 bytes: four bytes of administrative data, and a `union` of three single-precision floats and three ints. This implies that name strings for instance are replaced by a name id (an integer) through tokenization.

So in order to execute a GML program, the character string is first chopped into pieces surrounded by whitespaces, then converted to a token array, and finally such an array can be executed by the GML interpreter.

2.1 What does the Interpreter do with all these Tokens?

When the interpreter encounters a literal token, it executes it by simply pushing it on the stack. This also happens to opening markers, the symbols `[` and `{`. But closing markers have a special meaning.

When a `]` is found, values are popped from the stack until the first open bracket `[` is found. An array is created from the tokens in between, which is then pushed on the stack. So a program reading `[ [ 1 2 3 ]` simply creates an array containing three integers on top of the stack, and a marker as the next stack item. This mechanism implies that the stack can never contain a `]` marker. Arrays are always referred to *by reference*: If you `dup` an array, only the refence is duplicated, not the array. If you change a particular array, this affects all other references to it. The same applies to dictionaries.

The curly braces differ in that the array created is an *executable array*: A `{` puts the interpreter in *deferred mode*, i.e., *any* token is being put on the stack. This behaviour lasts until the *matching* `}` is found: open curly brackets are counted, unlike with square brackets. A single executable array is created from all items between the first and the last curly brackets. This becomes then the topmost stack element.

Such executable arrays can be used to simply evaluate them, using `exec`, and also for loops (`for`, `repeat` etc.), and for conditionals like `if` and `ifelse`. It is important to note that *there is no difference between functions and executable arrays* in PostScript, and also in the GML.

The difference between literal and executable names is that a literal name is preceded by a slash. This is because tokenization rules are quite simple: A token is checked whether it is an integer, a float, a vector etc. using simple string matching (via the `scanf` function), and if all fails, it is assumed to be a name. This implies that names may contain any characters except whitespaces (and dots), as long as they do not match any atomic token pattern. So `/this?is;a-name` and `x12;5,3+0)` – are legal literal and executable names. Well.

When the interpreter encounters a literal name, it is being put on the stack. It can be used to define a variable:

```
/my-circle [ (4,0,0) 2.5 ] def
```

creates a literal array containing a 3D point and a float, and assigns a name to it in the current dictionary.

An executable name is treated as the name of a variable or a function. To execute it means to look it up and to execute *its value*. When this value is an executable array – a function – the array currently being executed is pushed on the execution stack, and the interpreter instead executes the new function, again token by token. When it is finished, the previous function is popped from the execution stack, and its execution continues. This is how functions are called in PostScript and the GML.

Builtin operators provide the whole functionality of the language. The GML currently contains more than 250 builtin operators, grouped into 10 libraries. What an operator does is fairly simple: Typically, it pops some input values from the stack, checks their types, processes them, and pushes some results back. But it can do many things as a "side effect", for instance – create a polygonal mesh. What the operator does on the stack is called its *signature*. An operator does not have to have a fixed signature, but most operators actually do. The notation for signatures is somewhat ad hoc, with `I` for integer, `F` for float, `N` for scalar (integer or float), `P2`/`P3` for 2D/3D vectors etc. The add operator for instance can add numbers and vectors, consequently it has more than one signature:

```
add: a:N b:N → c:N
add: a:P2 b:P2 → c:P2
add: a:P3 b:P3 → c:P3
```

So an operator can behave differently depending on the types of input values, which is an example of polymorphism in the GML. All operators and their signatures are summarized in the Appendix.

Each operator is implemented as a C++ class. When the parser finds an operator name, it creates an operator token as part of an executable array. In PostScript terms, this is called "early binding".

2.2 Name Lookup and Scoping

It is important to know how name lookup works. It is done via the *dictionary stack*, with the *current dictionary* as its topmost element. To look up a name, all dictionaries on this stack are tested, from top to bottom, and the first dictionary where the key is defined delivers the value. But the dictionary stack is independent from the execution stack, and it can be changed at any time: The `begin` operator takes a dictionary from the usual operand stack and pushes it on the dictionary stack, so that it becomes the current dictionary. The `end` operator pops the current dictionary from the dictionary stack. So this mechanism is a flexible method for

(local) scoping, but also for function overloading: When `mymethod` is a function in the current dictionary, the sequence

```
mydict begin ... mymethod ... end
```

may change the meaning of `mymethod` when it is defined in dictionary `mydict`.

This method can also be used for local variables in order to minimize "stack acrobatics". This function for example creates vector $(2x,0,3x)$ from element x on top of the stack:

```
dup 2 mul exch 3 mul 0 exch vector3
```

With more complex examples this may become a little intransparent, and local variables can help:

```
dict begin /x edef x 2 mul 0 x 3
mul vector3 end
```

This creates an empty dictionary and makes it the current dictionary. Then the topmost element is given the name `x` to create the 3D vector $(2x,0,3x)$. The problem with this technique is that with nested functions it may create a deep dictionary stack.

To overcome the problem of stack acrobatics vs. slow dictionaries, the GML has *named registers*, which can be accessed even faster than the stack. Using this technique, the above example reads like this:

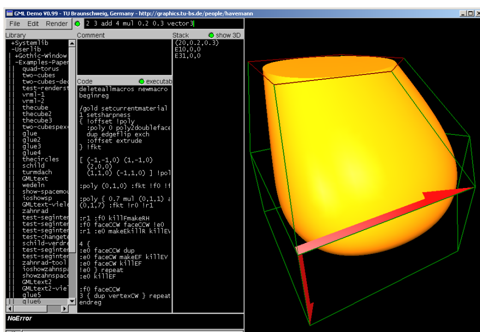
```
beginreg !x :x 2 mul 0 :x 3 mul
endreg
```

The second extension to the language is the *dot operator*. When a name is preceded by a dot, it is regarded as part of a *path expression*. When it is executed, the name is looked up it the topmost stack element, which is supposed to be a dictionary, and the object found is executed. As it is legal to leave out the spaces in between, path expressions like in C++ become possible.

```
Tools.Furniture.Chair.createLegs
```

This can be used for instance for *style libraries*, where each library implements the `Chair` tool differently.

3 The GML Development Environment



GML programs consist of plain ascii text. So in principle, they can be created using any text editor, and from other programs as well – just like PostScript output is created from a variety of different programs, including dvips or a2ps, or a PostScript printer driver.

For the domain of 3D graphics, it is desirable to have instant feedback of what your mesh construction program does. This is what the GML IDE is for. You can try out things, change parameters, add more operations, or regroup your functions, and explore the result interactively.

This section quickly presents the different sub-windows. Note that the borders between sub-windows are not fixed. They can be dragged in order to achieve a suitable view. With `Alt-F1` you can cycle through four different preset views. The `Escape` key quits the IDE, and any unsaved data are lost.

This section uses GML path expressions also for menu entries, for instance `File.Load_Library`. This functionality will also be available in the next IDE version.

3.1 The Prompt

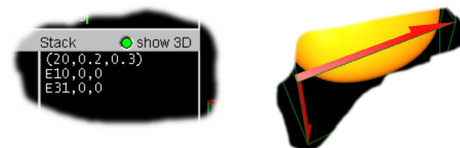
Commands entered in the prompt window are immediately executed when the enter key is pressed. There is also an immediate effect on the stack. You can see this with the following line of code, which computes $1/((3+4)\cdot 5)$:

```
3 4 add 5 mul inv
```

When you type in this line and *press enter*, you find the result 0.28... on top of the stack. In this tutorial, we will call this "to enter something at the prompt".

But what you can also do is to enter such a statement token by token: `3`, Enter, `4`, Enter etc. Then you can see all the intermediate results on the stack. This is a good way to get a feeling what happens when the interpreter executes a GML program.

All 2D and 3D points, halfedges, and polygons (i.e. point arrays) on the stack are also shown in the 3D window to the right. You can also select items on the stack, to highlight and identify them in the 3D scene.



It is important to note that there the same GML interpreter executes programs and direct input at the prompt. This means that if a program leaves the interpreter in some state (e.g. with things on the operand stack or a changed dictionary stack), this will be exactly the state used with direct input. This can be confusing for instance when an error happened between a `tmpdict begin ... end` so that a temporary dictionary is still on the dictionary stack. In this case it may help to enter `resetinterpreter` at the prompt. This will not destroy any functions or data, but reset the interpreter's internal state to the default.

3.2 Library Browser and Code Window

The library browser and the code window are the central facilities for code development and management of GML libraries. The library browser has three toplevel entries: Sys-

temlib, Userlib and Model, which are dictionaries. Any dictionary can be opened and closed by double-clicking (showing + or − signs). Double-clicking on a function though immediately executes it. A single click *selects* an item. If this is an (executable) array, the code window displays its content. If the selected item is a dictionary, the code window is empty.

When you enter or change the code, i.e. the content of the code window, you can enter it in the library (`Edit.Enter_Code`), or abandon your changes (`Edit.Forget_Changes`).

Changes also enter the library when you select something in the library browser (for instance the already selected item) or execute it via `Edit.Run` or `Alt-x`. The code window uses the usual shortcuts like shift+cursor keys to mark text, and Ctrl-C, Ctrl-X, Ctrl-V for copy, cut, paste. For a more complete keyboard reference see www.fltk.org. Note that marking with the mouse alone already copies text to the clipboard, and the middle button inserts it.

While Userlib and Model are initially empty, the Systemlib dictionary contains all libraries of builtin commands. So you can browse through the Systemlib if you are looking for a specific functionality, or you have forgotten an operator's signature: When you select a builtin operator, the comment window above the code window shows a brief comment including the operator's signature.

When the selection is a dictionary, the code window is empty. When you type in code then, this is understood as creating a new item in this dictionary. So when you try to commit the code, the item's name is prompted for.

3.3 The Debug Window

In case of an error, the debug window shows a stack backtrace. The topmost function is the one that was executed, and the bottom-most location is where the error happened. On each level, the token where the error was issued is shown "highlighted" using spaces and vertical bars.

To debug the `...  ||errorfkt||  ...`, the contents of this function will be shown on the next lower level.

The debugging facility is not yet perfect. As an example, some operators just throw an exception, for instance when trying to find an intersection between two parallel lines. This is a weaker form of an error, because exceptions lead to a program halt only if they are not catched. But currently, no backtrace is shown for uncatched exceptions.

3.4 The Menus

To load a GML library, first select a dictionary to contain this library. Typically, this is the `Userlib`. Then use `File.Load_Library` and select a `.genmod` file. Any dictionary can also be saved, with its name as default file name (but you can also name it differently). `File.Save_as_HTML` is only useful when you have the GML ActiveX control installed, that acts as a plugin for Internet Explorer. It creates a .html file for a given .genmod file.

`Edit.Animations` provides some functionality for record and play back of interactive 3D motions. Just try `Edit.Animations.Record`, play around with the mouse navigation in the 3D window, and then see what you've done with `Edit.Animations.Play`. When `Edit.Animations.Play_Outside` is on, you can see a demonstration of view cone plus backface culling. Note that with each recording session, you add to the end of the animation, unless you clear it.

The render menu has some obvious flags to control object appearance. When creating models, `Render.Control Mesh` should be enabled to see how the mesh changes. To make a screen shot, put a filename on the stack: just enter `"house.ppm"` at the prompt and press return, then select `Render.Screenshot` from the menu. If there's no filename on the stack, "screenshot.ppm" is used as default. Setting the background color works similarly, just put a 3D vector (r,g,b) on the stack. `Render.Reset View` is useful if you do not see any object and you're lost in 3D.

3.5 Undo/Redo on the level of Single Euler Operators

There is one slider on the bottom left of the IDE that can be used to scroll the internally stored sequence of Euler operators. This shows how the object was constructed. But note that in order to create a linear sequence, an (actually arbitrary) partial order of Euler macros is used! – This linearization of the construction may indeed vary, especially with complex macro dependencies.

3.6 The 3D Window

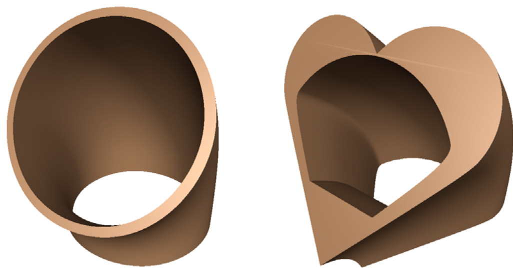
Last, but not least a word on navigation in the 3D window. When you have a 3 button mouse, navigation goes as follows:

- Left button: Rotate
- Middle button: Panning, parallel to image plane
- Right button: Zoom

In case you have no middle button, use Shift+Left button. Rotation is done around a point that lies slightly behind the surface point that covers the central pixel of the 3D viewport. The distance between the eye and this center of rotation also determines the speed for pan and zoom. When there's no surface on the center pixel, the previous center of rotation remains, which can be confusing sometimes.

But a particular surface point can be made center of rotation by picking on it with Shift+Left button. This movesc this point into the center of the viewport. When no part of the model can be seen in the 3D window, use `Render.Reset View`, which makes the origin visible at least.

4 The Pipe Tutorial



This is a step-by-step description of code and shape development. You can compare your results with `Models/MyTrial.genmod`.

1. Start the program. `Userlib` is selected in the library browser.

2. When starting from scratch, you typically first create a new dictionary via `Edit.New Dictionary`, for instance `Userlib.MyTrial`

3. Create a new function in this dictionary using `Edit.New_Item`, for instance `Userlib.MyTrial.test1`.

4. Make this item visible in the library browser by double-clicking on `Userlib.MyTrial`, and select `Userlib.MyTrials.test1`. The Code window now shows the contents of this function: It is empty.

5. Type in the following code in the code window:

```
deleteallmacros newmacro clear
(0,0,0) (0,-1,0) 1.0 4 circle
```

6. When you run it (Alt-X or `Edit.Run` or double-click on `Userlib.MyTrials.test1`), you see a quadrangle in the 3D window!

7. The circle operator expects midpoint, plane normal, radius or start vector, and number of segments on the stack. Such brief, but hopefully sufficient, information can be found in the appendix for all operators. So for the rest of this tutorial, only operator essentials will be discussed.

8. Take the `deleteallmacros newmacro clear` for now as just something every function needs to start with. The macro issue will be explained in a different tutorial.

9. Now copy the code using Ctrl-Pos1, Shift-Ctrl-End, Ctrl-C, and create a new item called `test2`. Paste the code using Ctrl-V and try Alt-X. It works. – From now on we will create a new item for each step. This helps to keep track of the changes.

10. Add the following lines to `test2` to create your first mesh:

```
/brzskin setcurrentmaterial
1 poly2doubleface
```

11. The current material belongs to the state of the interpreter. The operator pops the name and changes the state, but it doesn't push anything, so the stack top is again the polygon.

12. At program startup a number of default materials are loaded (from file Models/tubs.mtl). When you have a look at the material lib, you see that you can find out about available materials using `getmaterialnames`.

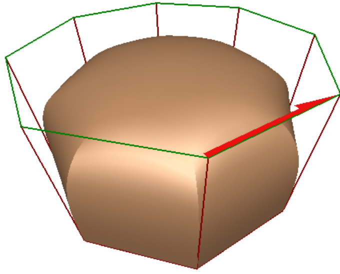
13. The stack now contains just one halfedge. Try out some double-clicks on `Systemlib.BRep .faceCCW` or `.faceCW`, or enter navigation commands in the prompt window, for instance
`dup vertexCW dup faceCCW edgeflip`

14. The cool thing now is that you can play around with parameters! Have a look at the documentation of `poly2doubleface` to see what the `1` means and try for instance `0`.

15. Now copy `test2` to a new `test3`, add a code line `(0,1,3) extrude` at the end, and run it. Now you've created a box!

    For extrude, an argument (x,y,m) means a displacement of the border of x units to the left, within the face plane, and y units in normal direction. The mode m determines the distribution of smooth and sharp edges. Now have a look in the documentation to see what m means!

16. Play around and change some parameters. For the argument of the extrude, instead of (0,1,3) try for instance (0.5,0.5,3), (0,1,2), or (0,1,4). Try more circle segments: Change the 4 in the third code line to 6,8, or 20.



17. Note that extrude pops and also pushes one halfedge! – This makes it possible to create a sequence of extrudes. Your `test7` might look like this:

```
deleteallmacros newmacro clear
(0,0,0) (0,-1,0) 1 8 circle
/brzskin setcurrentmaterial
1 poly2doubleface
(0.2,0.5,0) extrude
(0.2,0.5,0) extrude
```

18. The extrude operator is polymorphic and understands also an array of vectors. But in this case, the (x,y,m) displacementsc are *absolute*: You can obtain the same result as with the two extrudes using only (`test8`):

```
[ (0.2,0.5,0) (0.4,1,0) ] extrude
```

19. But what you cannot do within a single extrude is change the face normal direction! This is what we try next. Add the following code below a copy of `test8`:

```
beginreg !e
:e facenormal
(1,0,0) 10.0 rot_vec !n

:n 0.5 mul
:e facemidpoint add !p
:p :n mul !d

:p [ :p :p :n add ]
endreg
```

20. This uses a named register to store (`!e`) and retrieve (`:e`) the halfedge pushed by extrude. Note that in the following steps, you should add code lines *before* the `endreg` statement! – After it, you cannot access the named registers any more, of course.

21. The `:e` face's normal vector is then rotated by 10 degrees around the positive x-axis, so it stays normalized. Next we add half of this normal to the face midpoint to retrieve a displaced midpoint `:p`.

22. The GML format for a plane in 3-space is `n d`, where $n = (n_x, n_y, n_z)$ is the normalized face normal vector, and the scalar d is the signed distance of the plane from the origin. Given a point p on the plane, d can be computed as the dot product $d = \langle p, d \rangle$. In the GML, `mul : P3 → P3` gives the dot product: `:p :n mul !d`

23. The last line is just for visualization: An array (aka polygon) of two points (`:p` and `:p :n add`) is a line segment, and `:p` just places a dot in 3-space to show the location of the point.

24. Now we want to project a copy of the `:e` face polygon to the plane `:n :d`. The face polygon is obtained by `ring2poly` which pushes a polygon, a point array, on the stack. Now insert the following mysterious lines before `endreg` to obtain `test10` from `test9`:

```
:e ring2poly
{ dup (0,1,0) add :n :d
  intersect_lineplane pop }
map
```

25. Now we introduce the powerful `map` operator. It expects an array and a function on the stack. What it does is to loop through the array; it pushes each element on the stack, executes the function, and expects the function to leave something on the stack. So each time after the function, it pops the stack, and creates a new array from the elements it found. This array is what it leaves on the stack when it's done.

    This way, the `map` operator transforms one array into another array by mapping a given function to each element.

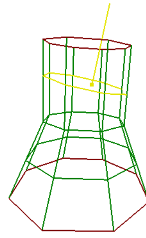
26. **GML functions are just arrays!** To see that, delete the `map` statement for a moment, run the code, and have a look at the stack. The top elements are a function and an array, and this is what map will get. Now type `aload` at the prompt and press Enter. This pushes all array elements individually on the stack, which works also for executable arrays. So what you see on the stack are the statements of your function!

27. To do the reverse, enter `7 array` at the prompt, and your operators are put together again to form an array; but it's not executable. So do a `cvx` ("convert to executable"), and you've got your function back. To try it out, just enter `map` at the prompt.

28. This demonstrates a remarkable advantage of the PostScript approach: It is extremely easy to generate program code. Just imagine you had to write a computer program that generates source code for C++ or Java!

29. But what does `map` do here? It calls an operator that intersects a line with a plane (our `:n :d`). The line is given by two points. We find one of them on the stack. We compute the second point from the first by adding $(0,1,0)$, which is `:e`'s face normal. (We could have used another local variable for it). The intersection operator returns also the t value of the intersection point along the line $p_1 + t(p_2 - p_1)$. But we don't need it, so it gets `pop`ped.

30. To see that this is indeed the projected face polygon,

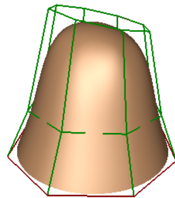


insert the following line at the end before `endreg`: `:e (0,1,1) extrude`. Switch `Render.Control mesh` on and `Render.Solid faces` off to see that it touches the mesh.

31. Now let's clean up and re-order the code, so that we obtain `test11` as:

```
deleteallmacros newmacro clear
(0,0,0) (0,-1,0) 1.0 8 circle
/brzskin setcurrentmaterial
1 poly2doubleface
[ (0.2,0.5,0)
  (0.4,1,0) ] extrude

beginreg !e
:e facenormal
(1,0,0) 10.0 rot_vec !n
:e (0,1,0) :n
:e facemidpoint
:n 0.5 mul add :n mul
project_ringplane
endreg
```

32. We actually want to project the face to the plane, not just the face polygon. So in principle we could use `moveV`, the "move vertex" operator, with `map`. But for faces, there's a builtin operator to do that. It expects an edge, a projection direction (in our example $(0,1,0)$ as `:e`'s face normal), and a plane `:n :d` on the stack.



33. As `:d` is used only once, we can compute it directly on the stack with `:e facemidpoint :n 0.5 mul add :n mul`. This is another PostScript technique! If you compute `e` from `a b c` *op1* and `e` is then used in `d e f` *op2*, you can "inline" `e`:

    `d   a b c` *op1*   `f` *op2*

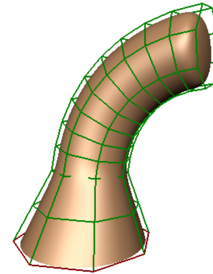
34. **Re-use of modeling operations.** The sequence of operations between `beginreg` and `endreg` is a compact little modeling tool! Now create a new dictionary `Userlib.MyTrial.Tools` with a new item `turnface`, and copy and paste the function code to it.

35. The new version of our program then behaves identically but now simply looks like this (`test12`):

```
deleteallmacros newmacro clear
(0,0,0) (0,-1,0) 1.0 8 circle
/brzskin setcurrentmaterial
1 poly2doubleface
[ (0.2,0.5,0)
  (0.4,1,0) ] extrude

MyTrial.Tools.turnface
```

36. But now that we have a single function, we can easily use it not only once but many times! The following is identical to writing down `extrude` ... `turnface` nine times:

```
9 { (0,0.1,0) extrude
    MyTrial.Tools.turnface
} repeat
```



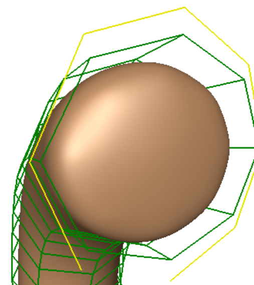
37. If we have more than one tool, it is both inconvenient and inefficient to always write down the whole pathname. In this caseit is more convenient to change the scope by pushing the `Tools` dictionary on top of the dictionary stack. So the same effect as above can be achieved by

```
MyTrial.Tools begin
9 { (0,0.1,0) extrude turnface }
repeat
end
```

38. Another advantage of this variant is that it is more easy to switch between different versions of `turnface`. By choosing the appropriate dictionary, the model "vocabulary" can be easily changed. This reflects the idea of *style libraries*.

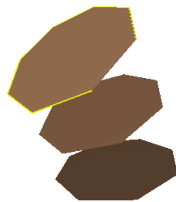
39. Now we have constructed a pipe bent by 90 degrees. But because we used projections, the profile is no longer a circle but merely ellipsoidal. We can see this if we draw a circle (an 8-gon) around the face midpoint. Note that an edge of the last face remains on the stack, so we use it to make the circle:

```
dup  facemidpoint
exch facenormal 0.6 8 circle
```



40. So let's start with a new idea to create a bent pipe: We rotate the polygon and create double-sided faces from each copy. Then we connect the faces along the pipe. We can do this so that the last copy of the polygon is always on the stack as input for the next step. So let `test14` look like this:

```
deleteallmacros newmacro clear
(0,0,0) (0,-1,0) 0.6 8 circle

dup 5 poly2doubleface pop

{ (0,0,-2) (1,0,0) 18.0 rot_pt } map
dup 5 poly2doubleface pop

{ (0,0,-2) (1,0,0) 18.0 rot_pt } map
dup 5 poly2doubleface pop
```



41. You can see immediately that the last two lines could be used in a loop, for example with `5 ... repeat`. But when a loop operates on the result from the previous loop pass, it is easier to design the loop body in an "unrolled" fashion like this.

42. The `bridgerings` operator can connect two different faces if they have the same number of vertices. It simply makes edges between corresponding vertices, traversing one face clockwise and the other counterclockwise. It starts by connecting the vertices of the two halfedges it pops from the stack. Finally it leaves one such bridge edge as result on the stack.

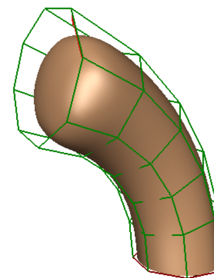
43. The idea is to remember the last face built as `!e`, rotate the polygon and make a doubleface, connect the backside to the previously built face, and store the frontside face as new `!e`. This results in the following code (`test15`):

```
deleteallmacros newmacro clear
(0,0,0) (0,-1,0) 0.6 8 circle
beginreg
dup 5 poly2doubleface !e

{ (0,0,-2) (1,0,0) 18.0 rot_pt } map
dup 5 poly2doubleface
dup edgeflip faceCCW :e
0 bridgerings pop !e

{ (0,0,-2) (1,0,0) 18.0 rot_pt } map
dup 5 poly2doubleface
dup edgeflip faceCCW :e
0 bridgerings pop !e

endreg
```

44. **Question:** Why are the 'horizontal' edges sharp and 'vertical' edges smooth in this example?
Answer: Background info.
The `poly2doubleface` mode parameter has the following meaning: With even modes 0,2,4,6 , the face has smooth edges, and with odd modes 1,3,5,7 it has smooth faces. But the four diffent numbers determine the *vertex types*. This is important for the type of the the 'vertical' edges created later when such a face is extruded. Modes 0/1 make all vertices smooth, 2/3 makes all vertices corners. But Modes 4/5 are interesting because they make vertices smooth by default, and they make a corner only if a point occurs *twice* in the polygon. And when you use `bridgerings` with mode 2, it will create smooth or sharp bridge edges according to the vertex types from `poly2doubleface`.

45. Now these four lines will basically make up the body of a loop (`test16`). Making it more general by introducing a few parameters, we can add `MyTrial.Tools.polycirclepipe` as another tool in our toolset. Note that it returns the first and last faces of the pipe because these will most likely be further processed.

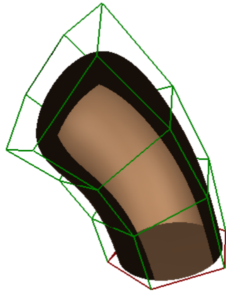
```
beginreg
!angle !k !axis !center !poly

:poly 4 poly2doubleface !e
:e edgeflip faceCCW !eStart
:poly
:k {
  { :center :axis :angle rot_pt } map
  dup 4 poly2doubleface
  dup edgeflip faceCCW :e
  2 bridgerings pop !e
} repeat pop
:eStart :e
endreg
```



46. We can use this tool now to make a true pipe with a thin wall. Topologically this is a torus, and it can in principle be made by subtracting a smaller pipe from a thicker pipe. But we can also construct the interior walls directly. Note what happens when we use our

new tool but simply reverse the orientation of the circle. This can be accomplished by using $(0,1,0)$ instead of $(0,-1,0)$ as plane normal for the `circle`:

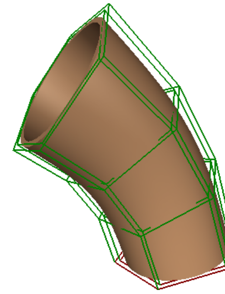
```
deleteallmacros newmacro clear
/brzskin setcurrentmaterial
MyTrial.Tools begin
(0,0,-2) (1,0,0) 3 18.0
beginreg
!angle !k !axis !point
(0,0,0) (0,1,0) 0.6 6 circle
:point :axis :k :angle polycirclepipe
endreg end
```



47. The resulting object is perfectly allright but it looks strange, because it is inverted: The faces are *clockwise* oriented. So faces closer to the viewer are actually backfaces, and faces on the backside of the object actually face the viewer. Now this is exactly what we want for the pipe interior.

48. But now we create *two* tubes, one for the outside and one, smaller and reversed, for the inside. The begin and end faces are returned by `polycirclepipe`! So all we need to do is to make the face of the smaller tube a *ring* of the larger tube, and that creates a hole! – Thanks to Euler operators.

```
deleteallmacros newmacro clear
/brzskin setcurrentmaterial
MyTrial.Tools begin
(0,0,-2) (1,0,0) 3 18.0
beginreg
!angle !k !axis !point

(0,0,0) (0,-1,0) 0.6 6 circle
:point :axis :k :angle polycirclepipe
!e0 !e1

(0,0,0) (0,1,0) 0.55 6 circle
:point :axis :k :angle polycirclepipe
!f0 !f1

:f0 :e0 killFmakeRH
:f1 :e1 killFmakeRH

endreg end
```



49. Again following our tools creation philosophy the logical step is to consider the two circles as input parameter for a new tool. We call it `MyTrial.Tools.emptycirclepipe`. It returns all four end faces:

```
beginreg
!angle !k !axis !point !poly2 !poly1

:poly1 :point :axis :k :angle
polycirclepipe !e0 !e1
:poly2 :point :axis :k :angle
polycirclepipe !f0 !f1
:f0 :e0 killFmakeRH
:f1 :e1 killFmakeRH

:e0 :e1 :f0 :f1
endreg
```

50. This reduces again our test program, to basically four lines `test19`:

```
deleteallmacros newmacro clear
/brzskin setcurrentmaterial
(0,0,0) (0,-1,0) 0.6 6 circle
(0,0,0) (0,1,0) 0.55 6 circle
(0,0,-1) (1,0,0) 5 12.0
MyTrial.Tools.emptycirclepipe
```

51. Now it's time for some variation of the input polygons. The following code creates a heart-shaped polygon:

```
(0,0,-1)    !c
(1,0,0.3)   !pr
(0,0,1)     !m
(-1,0,0.3)  !pl

:pr :m :pr :c sub neg circle_dir !ml
:pl :m :pl :c sub neg circle_dir !mr

[ :c  ]
[ :pr :midl :m ]
(0,-1,0) 0.1 2 circleseg arrayappend
[ :m :midr :pl ]
(0,-1,0) 0.1 2 circleseg arrayappend
[ :c  ] arrayappend
```

52. The `circle_dir` operator finds the center of a circle, given two points on the circle and a direction vector. A circle segment is basically specified as `[a m b]`, where *a* and *b* are points on the circle (start and

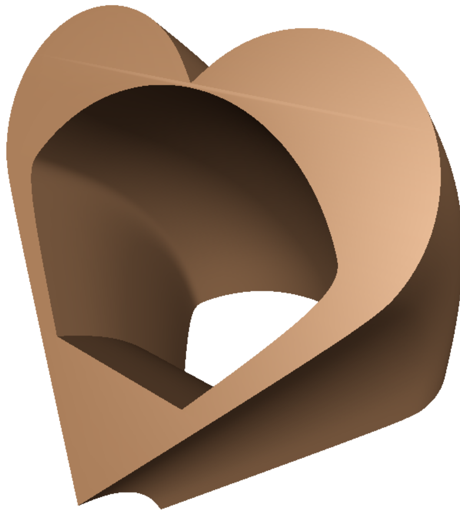
end of circle segment), and m is the center of the circle. The `circleseg` operator turns this into an array, similar to the `circle` operator. The center `:c` is appended again in the end to create a sharp corner (see background info on sharpness modes).

53. The inner polygon is created from another circle segment, and a line segment with double end points to make these sharp corners.

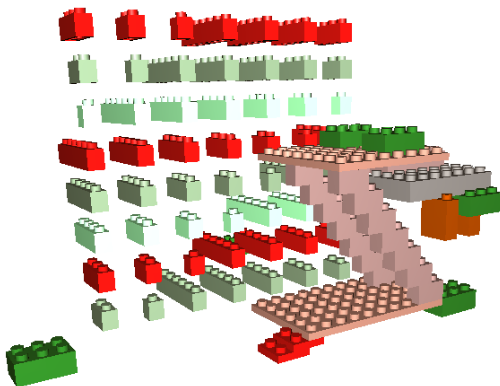
```
[ (0.9,0,0.4) (0,0,0) (-0.9,0,0.4) ]
(0,-1,0) 0.05 2 circleseg
[ (-0.4,0,-0.3) dup
  (0.4,0,-0.3)  dup ]
arrayappend reverse
```

54. With those two polygons on the stack two familiar lines suffice to make a pipe with a more interesting profile (`test22`):

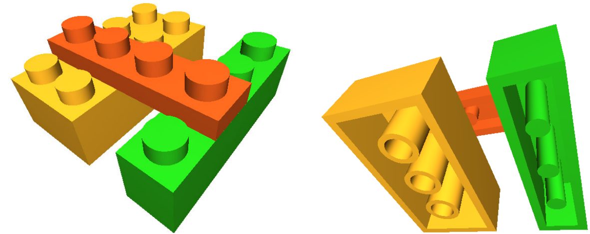
```
(0,0,-2) (1,0,0) 5 10.0
emptycirclepipe
```



5 The Lego Tutorial



- In `Lego.genmod` you find how Lego pieces are constructed. It was developed in a similar way and with similar techniques as the pipe tutorial in the last section. So it should be readily understandable.
- `Lego.stein-1` shows the basic construction of a plate with extruded border. Note that the `extrude` operator actually works for a face with a ring.
- The next step is to add the characteristic so-called *studs* to top and cylinders to the bottom. The bottom cylinders have to match in size with the studs. But the stud size is determined by the thickness of the border of the Lego piece!
- Try to change in `stein-2` the border width. It is set in the line `0.2 !rand`. Try values 0.1, 0.05 and 0.3 for the border width to see how this affects the stud and cylinder radii.
- In `stein-3` you finally see how a single lego piece is specified: With a 3D position $(2,3,3)$ and 2D extent $(lbx,lby) = (-2,2)$. Note how we need to find the signs of lbx and lby to correctly build the polygons to be extruded. And note that a Lego piece that is only one stud wide has different cylinders! You can try that out by replacing the $(-2,2)$ by $(1,2)$.



This is an example how also *conditional decisions* are needed in 3D modeling. This is an argument to have a full programming language as model representation!

- Examples `stein-4` and `stein-5` reveal the full power of a tools library: A Lego piece is in itself a complicated thing, but a 3D position and 2D extent are sufficient to specify it. This is possible with the GML:

```
/brass setcurrentmaterial
(0,3,15) (2,1) lego-pd
(0,3,12) (2,1) lego-pd
(0,1,12) (4,1) lego-pd
```

- The `stein-5` example also uses a loop to procedurally build Lego pieces, and it cycles through some materials.
- With larger constructions you see that constructing each piece individually is maybe not optimal. In this case it makes more sense to use reference objects! – But the problem remains to generate all different Lego pieces at least once...
- When you have a parameterized construction it is often not necessary to really accurately construct pieces. When suitably organized, you can change parameters later, for instance to generate Lego pieces that accurately match the dimensions of the real pieces.

... to be continued.

6 Tutorial Models

We propose the following tour through the example files in the Model directory. Note that you can load more than one library at a time. Just make sure that the `Userlib` is selected when loading.

6.1 `MyTrials.genmod`

See tutorial.

6.2 `Lego.genmod`

See tutorial.

6.3 `Examples.genmod`

This is a bunch of small GML examples each demonstrating particular aspects or techniques.

- `test-profil` demonstrates the `extrudepath` operator: Extrusion along a path.
- `quad-torus` shows the makeEkillR Euler operator.
- `vrml-1/2` are hypothetical examples how VRML functionality could be mapped to the GML.
- With `thecircles` an input polygon is transformed so that it has circular segments in the corners.
- The `test-seginterX` and `make-rooms` demonstrate line segment intersection followed by offset polygons. This is basically what you need to make a room with a number of walls. I always wanted to build a puppet's house with it.
- `test-decoration-X` is a nice shiny example for the `extrudearray` operator.
- `branch` connects offsets of polygons and of circle segments.
- `zahnrad`, german for *gear*, is an example of a complex procedural shape with only a few input parameters. If you happen to have a Spacemouse (with serial input), you can interactively change the input parameters of the gear with `ioshowzahnspace`.

6.4 `Gothic.genmod`

This is actually only a collection of tools used by `Haus-Arkade.genmod`. In general, you can tell the difference between simple models and just tools because models use to start with `deleteallmacros`, while tools start with `beginreg`.

Being a tools library, it nevertheless provides some test models in `Gothic.Test.gothicwindow-X`. They represent test models on the way to do more complex things with Gothic architecture.

6.5 `Haus-Arkade.genmod`

This has two main issues: The construction of simplistic house shapes, and to connect arches to form an arkade with a ledge. The point of departure though is a simple polygon: `Model.polygon` is defined when you execute `Haus-Arkade.generate-polygon`. You can also alter this function in a straightforward way.

- `arkadeX` shows what you can do with offset polygons of offset polygons.
- `arkade-with-house-X` are actually stress tests producing large amounts of geometry. Your PC should have enough RAM for that.
- `bogen-X` shows the versatility of a simple "doorway" tool that can be used in various ways and may even be recursively applied. This is the basis for the arkade, by the way.
- `eckhaus-X` constructs a house shape only through offset polygons and stable extrudes.
- `polyhaus-X` does the same with a more complex polygon, and also uses line intersection.

6.6 `Eulermacros.genmod`

The GML's underlying shape description are *Progressive Combined BReps* (short PCBReps). It has a built-in facility for level-of-detail not only via view dependent tesselation of subdivision surface, but also on the level of the control mesh. Internally, all Euler operations are stored in so-called *macros*. The examples in this library demonstrate how massive numbers of macros are handled by the engine.

In order to see this, execute `build-manymacros-3` for instance and switch on `Render.Macro culling`. Then zoom the object far away behind the backplane, and zoom in again. You see that parts of the model are constructed and taken away depending on their extent on the viewport. If you want to know more exactly how it's working, try `build-manymacros-2` and switch on `Render.Macro spheres`.

On a fast big machine `build-manymacros-4` should run fine. Needs much RAM.

6.7 `Gothic-Window.genmod`

Demonstrates some more intricate modeling, and the use of a style library. The point with Gothic architecture is that there's much self-similarity, so styles can be applied recursively.

`window-style8` has about seven million triangles at highest resolution.

7 GML Operator Summary

Core Library	
aload	[a1..aN] → a1..aN puts the elements of an array on the stack
append	[array] x → appends x to array
array	v1..vN N → [v1..vN] turns the N last objects on the stack into an array. 0 array is legal, -1 array is not.
arraypop	[x] n → removes the last n elements from an array
arrayremove	[x] n → removes element n from array
arrayappend	[arr1] [arr2] → [arr1 arr2] appends arr2 to arr1 and leaves the combined arr1 on the stack
begin	d:D → takes a dict from the stack and pushes it to the dictionary stack (→current scope).
bind	f → f' f and f' are equal except that executable names in f referring to operators are replaced by that operator directly
break	→ jumps immediately to the end of the currently executed array
catch	f catch → /Exception executes f and puts the name of the exception on the stack, or /NoError if nothing happened or a throw was issued. Example: 1 2 throw 3 catch → 1 2 /NoError Example: 1.0 0.0 div catch → /NumericError
clear	clears the stack
cleartomark	x..x [y..y → x..x pops all elements up to the first mark [encountered
copy	x1..xn n:I copy → x1..xn x1..xn a:A → a:A a':A d:D → d:D d':D either makes a copy of the last n elements on the stack, or performs a (shallow) copy of a dict or array.
count	x1..xN → x1..xN N counts the stack size
counttomark	[y1..yN → [y1..yN N counts the number of elements until the a mark [is encountered
currentdict	→ d:D pushes the topmost element from the dict stack to the stack. Doesn't change the dict stack
cvlit	name → /name

	array → [array] sets the executable flag of x to false
cvx	/name → name [array] → array sets the executable flag of x to true
def	/name object → defines object as name in current dictionary
dict	→ <dict> creates a new dictionary and puts it on the stack
dup	x → x x duplicates the topmos element on the stack.
eappend	x [array] → appends x to array, acts like exch append
edef	object /name → defines object as name in current dictionary. Useful: dup /name edef
end	→ pops the topmost element from the dictionary stack
eput	x [array] k → sets array[k] to x x <dict> /name → sets dict[name] to x f:F p:P2|P3 i:I → sets p.x to f for i=0 etc. That's to say, it is 'roll 3 2 put'
eq	x y → 0|1 Compares two objects for equality. Issues error if types do not match. floats are compared with eps 1e-4
exch	x y → y x exchanges topmost two objects on the stack
exec	x → executes the topmost element on the stack. Changes nothing for literals. Does something for operators, executable names and executable arrays. Useful together with load and/or cvx
exit	terminates the execution of the body of a for, repeat, forall, map, twomap, or loop statement
flatten	[[a0].ai.[an]] → [a0.ai.an] makes all elements of nested arrays elements of the main array. It's not legal to apply it to an executable array
for	init:I incr:I limit:I f → puts a number on the stack and executes f. Should work as well if floats are used instead of integers.
forall	[array] f puts each element of array on the stack and executes f

<table>
<tr><td></td><td>then</td></tr>
<tr><td>ge</td><td>x y → 0|1
Compares two objects and
returns 1 iff x>=y ('greater equal').
Compares only numbers and points.
For points, lexicographic ordering
is applied.
Issues error if types do not match</td></tr>
<tr><td>get</td><td>[a0..aN] k → ak
<dict> /name → o_name
(x,y,z) 0|1|2 → x|y|z
(x,y) 0|1 → x|y
Retrieves an element by index
or name from an array, point or dict.
k=-1..-(N+1) retrieves last..first</td></tr>
<tr><td>gt</td><td>x y → 0|1
Compares two objects and
returns 1 iff x>y ('greater than').
Compares only numbers and points.
For points, lexicographic ordering
is applied.
Issues error if types do not match</td></tr>
<tr><td>if</td><td>1 f → f exec
0 f →
executes f in fact
iff the first arg is
not equal 0 (or 0.0)</td></tr>
<tr><td>ifelse</td><td>1 f g → f exec
0 f g → g exec
executes f in fact
iff the first arg is
not equal 0 (or 0.0)</td></tr>
<tr><td>ifpop</td><td>x y 0 → x
x y 1 → y
is just exch if pop</td></tr>
<tr><td>index</td><td>vN..v0 k:I → vN .. v0 vk
puts the k'th object on
the stack on the top of it</td></tr>
<tr><td>keys</td><td>d:D → [/key1../keyN]
creates an array of literal
names from all keys
of a given dictionary.</td></tr>
<tr><td>known</td><td>dict:D /name → 0|1
checks if name is the key
of an element in dict.</td></tr>
<tr><td>le</td><td>x y → 0|1
Compares two objects and
returns 1 iff x<=y ('less equal').
Compares only numbers and points.
For points, lexicographic ordering
is applied.
Issues error if types do not match</td></tr>
<tr><td>length</td><td>[a0..an] → N
<dict> → N_keys
p:P2 → 2
p:P3 → 3
puts the length of an
object on the stack.</td></tr>
<tr><td>load</td><td>/name → x
looks in the dictionary stack
for an object with /name and
puts it on the stack</td></tr>
<tr><td>loop</td><td>f →
executes f forever - until
for instance an exception</td></tr>
</table>

	is thrown, or exit is called
lt	x y → 0|1 Compares two objects and returns 1 iff x<y (’less than’). Compares only numbers and points. For points, lexicographic ordering is applied. Issues error if types do not match
map	[array] f → [array’] puts each element of array on the stack, executes f, and collects the topmost element to create array’ which has the same length as array then
ne	x y → 0|1 Compares two objects for inequality. Issues error if types do not match
pop	x → removes the topmost element from the stack
pops	x0 x1 ... xn n → pops → x0
put	[array] k x → sets array[k] to x <dict> /name x → sets dict[name] to x p:P2|P3 i:I f:F → sets p.x to f for i=0 etc.
repeat	n:I f → executes f n times
resetinterpreter	clears the interpreter’s internal state including stack, dictstack, execution stack, error state, exit state, clears the pops, basically calling GMLInterpreter::reset
reverse	[a1..aN] → [aN..a1] ATTENTION: This reverses the order of the _original_ array. First make a copy if you don’t want that: copy reverse.
roll	x1..xN N k roll : x1..xN N 1 roll → xN x1..xN-1 x1..xN N -1 roll → x2..xN x1 rolls N elements on the stack, up (towards the top) with k>0, down (away from top) with k<0
throw	throws /NoError exception, popping the execution stack until the first catch is encountered.
twomap	[a1] [a2] f → [a’] a1 and a2 must have the same size, otherwise: error. Loops over both arrays and puts two elements on the stack and executes f. Collects the topmost element then to create a’ (has same length as a1, a2).
type	x → /type:S puts x’s type as a literal name on the stack
undef	dict:D /key → undefines a key in a dict. Issues error if key doesn’t exist.

values	d:D → [val1..valN] creates an array from all vars stored in a given dict. Order is guaranteed to be the same as with the keys operator
where	/name → d:D 1 /name → 0 d is the first dict of the current dictionary stack that has /name defined. 0 is pushed if /name cannot be found in the whole dict stack.

CoreMath Library	
abs	a:N → b:N a:P2 → b:N a:P3 → b:N b is 'a a mul sqrt'.
add	a b → c adds integers, floats, P2, and P3 iff a and b agree in type
and	a:N b:N → 0|1 returns 1 iff both a and b do not equal 0 (or 0.0)
atan	a:F → b:F computes the arcus tangens
atan2	x:F y:F → b:F (x,y):P2 → b:F returns the arctangent of y/x in the range -180 to 180 using the signs of both arguments to determine the quadrant of the return value.
ceiling	a → b:I ceiling: b equals a iff 'a 1.0 mod' equals 0.0. otherwise b equals 'dup 1.0 mod sub 1 add' except that it is an int.
cos	a:N → b:F computes the cosine function
div	a:N b:N → c:N a:P2 b:N → c:P3 a:P3 b:N → c:P3 divides an number or point through a number.
exp	a:N → b:F computes the exp(a) exponential function (to basis e)
floor	a → b:I ceiling: b equals a iff 'a 1.0 mod' equals 0.0. otherwise b equals 'dup 1.0 mod sub' except that it is an int.
inv	a → b inverts a number, i.e. b=1.0/a. b is always a float
ln	a:N → b:F computes the natural logarithm of a (basis e)
log	a:N → b:F computes the decimal

	logarithm of a
mod	a:N b:N → c computes c as a modulo b. a and b may be floats.
mul	a:N b:N → c:N a:P3 b:N → c:P3 a:N b:P3 → c:P3 a:P2 b:N → c:P3 a:N b:P2 → c:P3 a:P2 b:P2 → c:F a:P3 b:P3 → c:F multiplies two objects which can be numbers or points. If a and b are points, performs dot product (yeah!)
neg	a → b negates an object, i.e. b=-a. Works for numbers and points.
not	a:N → 0|1 returns 1 iff a equals 0 or 0.0
or	a:N b:N → 0|1 returns 1 iff either a or b or both do not equal 0 (or 0.0)
pi	→ 3.14159265359
pow	a:N b:N → c takes a to the power of b.
round	a → b:I rounds a float to the nearest integer.
sin	a:N → b:F computes the sine function
sqrt	computes the square root of a number.
sub	a b → c subtracts integers, floats, P2, and P3 iff a and b agree in type
tan	a:N → b:F computes the tangens function
truncate	a → b:I chops off the decimal digits after the point. Equals floor for a>0 and ceiling for a<0.

CoreVector Library	
aNormal	v:P3 → v':P3 chooses a unit vector v' such that v v' dot is zero, ie. v' is perpendicular to v.
cross	u:P3 v:P3 → w:P3 returns the cross product of u and v, ie. w is perpendicular to both u and v.
determinant	u:P2 v:P2 → det:Float u:P3 v:P3 w:P3 → det:Float computes 2x2 or 3x3 determinant
dist	p0:P3 p1:P3 → d:F p0:P2 p1:P2 → d:F d is the distance between points p0 and p1.
getX	(x,y):P2 → x (x,y,z):P3 → x

getY	(x,y):P2 → y (x,y,z):P3 → y
getZ	(x,y,z):P3 → z
normal	u:P3 v:P3 → w:P3 face:E → w:P3 returns the cross product of u and v, ie. w is perpendicular to both u and v. For a face, return the face normal.
normalize	v:P3 → v':P3 v' has unit length iff v is not (0,0,0), in which case GeometricError is thrown.
planemul	u:P3 v:P3 (x,y):P2 → w:P3 computes w = x*u + y*v
putX	(x,y):P2 x' → (x',y) (x,y,z):P3 x' → (x',y,z) replaces a point's x coordinate.
putY	(x,y):P2 y' → (x,y') (x,y,z):P3 y' → (x,y',z) replaces a point's y coordinate.
putZ	(x,y,z):P3 z' → (x,y,z') replaces a point's z coordinate.
vector2	x:N y:N → (x,y) turns two numbers into a P2
vector3	x:N y:N z:N → (x,y,z) turns three numbers into a P3

Register Library	
beginreg	→ pushes a new register frame
endreg	→ pops the current register frame, reverting to the last

BRep Library	
tesselate	→ update the mesh tesselation including subdivision surfaces and triangulations.
makeVEFS	p0:P3 p1:P3 → e:E e is directed from p0 to p1, ie. 'e vertexpos' gives p0.
makeEV	e0:E e1:E p:P3 → e:E splits an edge or vertex Example: 'e dup vertexCW p makeEV' inserts a new point on e, and e's vertexpos is now p, and the result is e vertexCW.
makeEVone	e0:E p:P → e1:E draws a new edge from e0 vertex to p in e0's face. Result e1 has p as vertexpos.
makeEF	e0:E e1:E → e:E e0 and e1 must belong to different vertices of the same face. Creates a diagonal and a new face, which is to the LEFT of the line e1→e0. The resulting edge has e1's vertexpos.
makeEkillR	eRing:E eFace:E → e:E eRing's face must be a ring of eFace's face, obviously. Creates a new edge e between the respective vertices. e has the same vertexpos as eRing.

makeFkillRH	eRing:E → eRing is an edge that belongs to a ring. This ring is stolen from its baseface and turned into a face of its own right.
killVEFS	E → issues error if it's not an isolated component
killEV	e:E → performs an 'edge collapse' of both endpoints of e.
killEF	e:E → merges two faces by deleting the edge between them. e's face is deleted, actually. NOTE: e and e edgeflip must not have the same face!
killEmakeR	e:E → eRing:E e and e edgeflip must have the same face. e's vertex: belongs to ring e edgeflip's vertex: to baseface
killFmakeRH	eFace:E eBaseface:E → eFace belongs to a face that ideally lies geometrically inside baseface. eFace's face is then turned into a ring of eBaseface's face.
moveV	e:E p:P → The position of e's vertex is changed to be p.
moveE	e:E v:P → The position of both of e's vertex is translated by v.
moveF	e:E v:P3 → all vertices of the ring or face are translated by offset v.
sharpE	e:E 0 → e:E 1 → sets the sharpness flag of e
sharpF	eFace:E 0|1 → sets the sharpness of all edges of a face.
sharpV	eFace:E 0|1 → sets the sharpness of all edges around a vertex.
faceCCW	e0:E → e1:E iterates one edge further in the same face, ie. in CCW direction. Very cheap operation.
faceCW	e0:E → e1:E iterates one edge back in the same face, ie. in CW direction Has cost in order of face degree.
vertexCCW	e0:E → e1:E iterates one edge back around the same vertex, ie. in CCW direction. Has cost in order of vertex degree.
vertexCW	e0:E → e1:E iterates one edge further around the same vertex, ie. in CW direction. Very cheap operation.
edgeflip	e0:E → e1:E flips on the reverse of e0,

	ie. e0's mate. Very cheap operation.
nextring	e0:E → e1:E jumps to the next ring of the face. Jumps back to baseface after the last ring. If the face has no rings, this op is the identity, obviously.
baseface	e0:E → e1:E jumps to the baseface of a ring. If the face has no rings, this op is the identity, obviously.
vertexpos	e:E → p:P3 returns the position of e's vertex.
valence	e:E → n:I e's vertex is incident to n edges.
edgedirection	e0:E → v:P3 vector along edge e0 in faceCCW direction. Equals in effect to: ' e0 faceCCW vertexpos e0 vertexpos sub '
facedegree	e:E → n:I e's face or ring has n edges. Doesn't count next rings, or the baseface.
facemidpoint	e:E → pmid:P3 returns the midpoint of e's face (or ring). NOTE: This is an expensive operation, because the midpoint is computed at every call!
facenormal	e:E → nrml:P3 returns the face normal of e's face. NOTE: This is an expensive operation, because the normal is computed at every call!
faceplanedist	e:E → dist:F returns the distance of e's averaged face plane to the origin. NOTE: This is an expensive operation, because the face plane is computed at every call!
faceplane	e:E → n:P d:F returns the plane equation of e's averaged face plane to the origin. NOTE: This is an expensive operation, because the face plane is computed at every call!
minfacedist	e0:E e1:E → e0':E e1':E d:F looks for the closest pair of vertices in faces of e0 and e1. Returns them and a their distance.
hasrings	e:E → 0|1 checks if e belongs to a face with rings, or if it is a ring with a next ring.
isBaseface	e:E → 0|1 checks if e belongs to a baseface.

issharp	e:E → 0|1 returns e's sharpness flag
sameFace	e0:E e1:E → 0|1 returns 1 iff both edges belong to the same face or ring (NOT: same baseface!)
sameEdge	e0:E e1:E → 0|1 returns 1 iff both edges are either equal or mates
sameVertex	e0:E e1:E → 0|1 returns 1 iff both edges are incident to the same vertex
connectedvertices	e0:E e1:E → 0 e0:E e1:E → e01 1 returns 'e01 1' iff the vertices of e0,e1 are connected via e01. In that case, e01's vertex equals e0's vertex.
isValidEdge	e:E → 0|1 checks whether you may still use e. This is not true for instance if e belongs to an erased or an inactive macro.
checkR	e:E → checks if any of the rings of the face of e edgeflip have to be moved to e's face. (and does so then) Should be done after makeEF on a face with rings, as by default the new face has no rings at all.
setsharpness	0|1 → sets the current edge sharpness setting that applies to all Euler ops
getsharpness	→ 0|1 retrieves the current edge sharpness setting that applies to all Euler ops
pushsharpness	→ pushes the current edge sharpness on a stack without changing it
popsharpness	→ 0|1 sets the edge sharpness to the value that was last pushed on the sharpness stack with pushsharpness
pointinface	pt:P3 face:E → 0 face:E pt:P3 → 1 face:E pt:P3 → e:E flag:I flag = 0: point not in face flag = 1: point lies inside face flag = 2: point lies on edge flag = 3: point lies on vertex for flag=2, point lies on segment from [e, e faceCCW]
rayintersect	pt:P3 dir:P3 → t:F e:E p:P3 flag pt:P3 dir:P3 → 0 shoots a ray (pt,dir) to the mesh flag = 0: no hit, no further items flag = 1: face was hit flag = 2: edge was hit

	flag = 3: vertex was hit t is the parameter for the hit point p: p = pt+t*dir e is the edge that was hit, for face hit (flag=1) it's the edge closest to p
rayintersecttwice	pt:P3 dir:P3 → t1:F e1:E p1:P3 flag1:I t2:F e2:E p2:P3 flag2:I 2 pt:P3 dir:P3 → t1:F e1:E p1:P3 flag1:I 1 pt:P3 dir:P3 → 0 shoots a ray (pt,dir) to the mesh and looks for the FIRST TWO hits, as (t1,e1,p1,flag1) and (t2,e2,p2,flag2): flag = 1: face was hit flag = 2: edge was hit flag = 3: vertex was hit t is the parameter for the hit point p: p = pt+t*dir e is the edge that was hit, for face hit (flag=1) it's the edge closest to p
rayintersectinface	face:E pt:P3 dir:P3 → 0 face:E pt:P3 dir:P3 → t:F pRay:P pFace:P e:E 1 face:E pt:P3 dir:P3 → t:F pRay:P pFace:P e:E 2 projects the ray (pt,dir) in face flag = 0: no hit, no further items flag = 1: edge was hit flag = 2: vertex was hit t is the parameter for the hit point: p = pt+t*dir e is the edge that was hit, for edge hit (flag=1) pFace lies on segment from [e, e faceCCW]
intersect_faceplane	face:E n:P3 d:f → [p:P e:E h:I] returns the intersections of plane (n,d) with the face. Each hit record contains an integer h=1: edge was hit h=2: vertex was hit for h=1, p lies on the segment from [e, e faceCCW]
findbackwall	wall:E start:E nmax:I → eMin:E nmax<0: nmax=1000

BRepMacro Library	
clearmacro	M → M /name → This will - undo the macro (with children) - kill all child macros - delete all stored Euler ops - and make it the current macro returns newly created macro macro is already dead. If name is given, macro will be updated/created and defined in current dict under /name.
deleteallmacros	→ M creates a new macro, makes it the current macro and puts it on the stack
currentmacro	M → gets a macro which will become the active macro (maybe again). All possibly existing children are actually killed. Issues 'BRepError' if macro

	is dead
deletemacro	M → Kills a macro. Issues 'BRepError' if the macro is already dead
loadmesh	filename:string → M loads a mesh from given obj file, creates a macro, makes it the current macro, and puts it on the stack.
unloadmesh	→ once a mesh is loaded, it cannot be removed usind deleteallmacros, use unloadmesh instead
savemesh	filename:string → saves the current mesh in .obj format, as a Combined BRep, of course.
newmacro	→ M creates a new macro, makes it the current macro and puts it on the stack
endmacro	→ finishes the current macro. current macro is undefined then.
redomacro	M → This will redo M, but not its children. Issues 'BRepError' if macro is dead
redomacrodepth	M l:int → gets a macro to redo and a number of child levels to redo. -1 means 'redo all children to any depth'. Issues 'BRepError' if macro is dead
undomacro	M → This will - undo all children of M - and then undo M. Issues 'BRepError' if macro is dead
edge2macro	E → M returns the macro which has created E
macroisdirectparent	m0:M m1:M → t:I t=1 if m0 is a direct parent of m1, t=0 otherwise
macroisdirectchild	m0:M m1:M → t:I t=1 if m0 is a direct child of m1, t=0 otherwise
macroisparent	m0:M m1:M → t:I t=1 if m0 is a parent of m1, t=0 otherwise. Recursive version of macroisdirectparent.
macroischild	m0:M m1:M → t:I t=1 if m0 is a child of m1, t=0 otherwise. Recursive version of macroisdirectchild.
macrosetLOD	p:P3 dist:F →
macrosetLODparam	angleUndo:F angleRedo:F activeFrames:I → macros with solid angle > angleRedo are activated

	solid angle < angleUndo are deactivated so you should have angleUndo < angleRedo, for example 0.2 and 0.3 relative to view cone size

Modeling Library	
faceneighborhood	faces:[E] → neigh:[E] each edge in the ’faces’ array represents a face in the mesh, and it is regarded as a face set. neigh is the set of all edges with vertex on a face of ’faces’, but on both sides with faces not from ’faces’. So, neigh is the set of edges ’emanating’ from face set ’faces’. ATTENTION: Expensive.
extrude	e:E width:F height:F sharp:I → E e:E (w,h,m):P3 → E extrudes e’s baseface, respecting the rings smoothness s = a+b where m HORIZ VERTICAL 0 smooth smooth 1 sharp smooth 2 smooth sharp 3 sharp sharp 4 smooth like vertex 5 sharp like vertex 6 smooth continue 7 sharp continue
extrudeAsRing	E width:F → E
extrudering	E width:F height:F sharp:I → E
extruderingarray	e:E p:[P3] m:I rel:0|1 → E e:E p:P3 m:I rel:0|1 → E face extrusion according to array last point is repeated if array too small rel=0: p is array of extruded positions rel=1: p is array of offset vectors m HORIZ VERTICAL 0 smooth smooth 1 sharp smooth 2 smooth sharp 3 sharp sharp 4 smooth like vertex 5 sharp like vertex 6 smooth continue 7 sharp continue
extrudestable	[E] [P] → [E] too complicated to explain, but definitely extremely cool. Remark: it may be a bit unstable (thus the name), especially if the faces are too small in extent (size of 50 should do though)
extrudepath	[E] [P] → [E] [E] P → [E] E [P] → E E P → E
extrudearray	[E] [P] mode:I → [E]
subdivedge	[P] e:E→ [E] turns the points array into an edges array. Skips e vertexpos and stops before e mate vertexpos, if they

	are part of the array.
makehole	e0:E e1:E flag:I → E flag = 0: smooth edges flag = 1: sharp edges flag = 2: sharp if e0 OR e1 corner flag = 3: sharp if e0 AND e1 corner flag = 4: sharp if e0 corner flag = 5: sharp if e1 corner
gluefaces	E E →
makeladder	e0:E mode:I → [E] creates quadrangles by making edges following e0 in CW and CCW ways mode=0: smooth edges mode=1: sharp edges mode=2: sharp if 1 endpoint sharp mode=3: sharp if 2 endpoints sharp
faceborder	[E] → [[E]..[E]] Returns an array of closed paths, slightly slower than faceborderset
faceborderset	[E] → [[E]..[E]] Returns an array of boundaries, but edge arrays do probably not form nice paths
slickExtrude	E height:F width:F → slickExtrude → E
project_ringplane	r:E dir:P n:P d:F → moves all vertices of ring r in direction dir so that they lie in plane (n,d). Throws GeometricError if that's not possible.
ring2poly	E → [P] turns a ring into a polygon.
path2poly	[E] → [P] turns a path into a polygon. If the path is closed, the last point (which equals the first) is omitted.
poly2doubleface	[P] m:sharpmode → E turns a polygon into a double-sided isolated face. Mode m is like with extrude.
project_polygonface	[P3] e:E dir:P3 → [E] projects polygon in direction dir on face e, following to neighbour faces if necessary. Ray from first point to face must hit e's face.
polys2faceswithrings	[[P0]..[PN]] nrml:P3 0|1 → [E] turns a number of coplanar polygons into a collection of faces with rings. The last parameter 0|1 determines if rings are also inserted on the backfacing side!
extrudepolygon	[P] (w,h,m):P3 → top:E bottom:E turns a polygon into an extruded object. the mode m is like in extrude.

Geometry Library	
angle_2vec	u:P v:P → angle:F Computes the angle between vectors u and v, which is in [0,pi]. Throws GeometricError if u or v have zero length.
anglenormal_2vec	u:P v:P n:P → angle:F Computes the angle alpha between

	vectors u and v, which is in [0,pi]. If (u,v,n) form a right-handed coordinate system, return alpha, otherwise return 2*M_PI-alpha. Throws GeometricError if u or v have zero length.
angle_3pt	p0:P p1:P p2:P → angle:F Computes the angle between vectors p0-p1 and p2-p1, which is in [0,180]. Throws GeometricError if either vector has zero length.
angle_3ptpoly	p0:P p1:P p2:P → angle:F Computes the angle between vectors p1-p0 and p2-p1, which is in [0,180]. Throws GeometricError if either vector has zero length.
coordabs_2vec	u:P3 v:P3 → s:F t:F Normalizes u, then computes u' such that u and u' span a plane containing v, but u.u'=0 and |u'|=|u|=1. Then returns s and t such that v=s*u + t*u'. Throws GeometricError if u has zero length.
coordAbs_3pt	p0:P p1:P p2:P → s:F t:F Let u=p1-p0 and v=p2-p0. Normalizes u, then computes u' such that u and u' span a plane containing v, but u.u'=0 and |u'|=|u|. Then returns s and t such that v=s*u + t*u'. Throws GeometricError if u has zero length.
coordsame_2vec	u:P3 v:P3 → s:F t:F Computes u' such that u and u' span a plane containing v, but u.u'=0 and |u'|=|u|. Then returns s and t such that v=s*u + t*u'. Throws GeometricError if u has zero length.
coordsame_3pt	p0:P p1:P p2:P → s:F t:F Let u=p1-p0 and v=p2-p0. Computes u' such that u and u' span a plane containing v, but u.u'=0 and |u'|=|u|. Then returns s and t such that v=s*u + t*u'. Throws GeometricError if u has zero length.
dist_ptplane	p:P n:P d:F → dist:F Computes the signed distance of point p from the plane (n,d).
dist_ptseg	p:P p0:P p1:P → d:F Computes the distance of point p from the closed segment [p0,p1].
intersectnormal_2vec	v0:P v1:P → q:P3 n:P3 v0 and v1 specify two planes (n0,d0) and (n1,d1): namely (v0/|v0|,|v0|) and (v1/|v1|,|v1|). This computes the intersection

	line q+t*n of these planes, with q a point on the line and n the direction vector along the line. Throws GeometricError if v0,v1 are parallel.
intersectSTPQ_2line	p0:P3 p1:P3 q0:P3 q1:P3 → s:F t:F p:P3 q:P3 Pairs p0,p1 and q0,q1 specify two lines. Computes s,t and p,q such that p=p0+s*(p1-p0), q=q0+t*(q1-q0), and p and q are the points where both segments come closest. Throws GeometricError if the lines are parallel.
intersectST_2line	p0:P3 p1:P3 q0:P3 q1:P3 → s:F t:F Pairs p0,p1 and q0,q1 specify two lines. Computes s,t such that p0+s*(p1-p0) and q0+t*(q1-q0) are the points where both segments come closest. Throws GeometricError if the lines are parallel.
intersect_2line	p0:P p1:P q0:P q1:P → p:P s:F t:F Pairs p0,p1 and q0,q1 specify two lines. Computes the point p where they intersect. Throws GeometricError if there is no such point.
intersect_lineplane	p0:P3 p1:P3 n:P3 d:F → q:P3 t:F Computes the intersection (p,t) of the line through p0, p1 with the plane (n,d), as p=p0+t*(p1-p0). Throws GeometricError if plane and segment are parallel, regardless of their distance.
intersect_2plane	n0:P d0:F n1:P d1:F → q:P3 n:P3 computes the intersection line q+t*n of planes (n0,d0) and (n1,d1), q a point on the line and n the direction vector along the line. Throws GeometricError if n0,n1 are parallel.
intersect_line_ellipse	p0:P3 p1:P3 mid0:P3 mid1:P3 radius:F → sec0:P3 sec1:P t0:F t1:F Computes the intersection of line (p0,p1) with the ellipse (mid0,mid1,radius). radius should of course be greater than the distance(mid0,mid1). t0,t1 are the parameters on the line p0+t(p1-p0), and t0<=t1. sec0,sec1 are the respective intersection points. Throws GeometricError if there's no intersection.
intersect_line_ellipse2D	p:P2 d:P2 mid0:P2 mid1:P2 radius:F → t0:F t1:F Computes the parameters t0,t1 of the intersection points of ray p+t*d with the ellipse (mid0,mid1,radius). radius should of course be greater than the distance(mid0,mid1). Throws GeometricError if there's no intersection. Note that t0,t1 do not have to be in [0,1].

	It's guaranteed though that t0<=t1.
intersect_spheres	mid0:P3 rad0:F mid1:P3 rad1:F → (x,y):P2 computes the intersection circle of spheres (mid0,rad0) and (mid1,rad1). Let v = (mid1-mid0) / |mid1-mid0|, so that the intersection circle lies in a plane normal to v. Then (x,y) are returned such that the plane contains the point (mid0 + x*v), and has radius y. Throws GeometricError if they don't intersect.
intersect_circles	m0:P3 r0:F m1:P3 r1:F nrml:P3 → p0:P3 p1:P3 computes the intersection point of circles (m0,r0) and (m1,r1) in plane nrml. If you look the plane from above, with [m0,m1] a horizontal line segment with m0 to the left and m1 to the right, then p1 is above the segment and p0 is below. Returns incorrect intersection points if [m0,m1] is not parallel to plane nrml. Throws GeometricError if they don't intersect.
intersect_circleseg_circle	[p0:P3 p1:P3 p2:P3] mid:P3 rad:F nrml:P3 → p:P3 Throws GeometricError if they don't intersect.
intersect_2circleseg	[p0:P3 p1:P3 p2:P3] [q0:P3 q1:P3 q2:P3] nrml:P3 → p:P3 Throws GeometricError if they don't intersect.
intersect_segsphere	p0:P3 p1:P3 mid:P3 rad:F → sec0:P3 sec1:P t0:F t1:F computes the intersection of line (p0,p1) with sphere (mid,rad). Note that this is also the intersection of line (p0,p1) with the circle (mid,rad) in the plane spanned by points p0,p1,mid. t0,t1 are the parameters on the line p0+t(p1-p0), and t0<=t1. sec0,sec1 are the respective intersection points. Throws GeometricError if they don't intersect.
intersect_segments	[P] → [P] [[P]] → [P] takes an array of points with even size or an array of polys and returns them so that the segments intersect only at the endpoints.
segs2polygons	[P] nrml:P3 → [[P]]
line_2pt	p0:P p1:P t:F → p:P Computes p = p0+t*(p1-p0) = (1-t)*p0+t*p1.
midpoint_2pt	p0:P p1:P → p:P Computes the midpoint p=0.5*(p0+p1) of segment (p0,p1).
normalabs_2vec	u:P3 v:P3 → u':P3 Computes u' such that u and u' span a plane containing v, with u.u'=0 and |u'|=1. Throws GeometricError if u has zero length, or u and v are parallel.
normalsame_2vec	u:P3 v:P3 → u':P3 Computes u' such that u and u' span a plane containing v, with u.u'=0 and |u'|=|u|. Throws GeometricError if u has zero length,

	or u and v are parallel.
normal_2vec	u:P3 v:P3 → u':P3 Computes u'=v-(u.v)/(u.u)u, which is such that u.u'=0, and u and u' span a plane containing v. Throws GeometricError if u has zero length.
plane_3pt	p0:P3 p1:P3 p2:P3 → n:P3 d:F Computes plane (n,d) in Hessian normal form from p0,p1,p2. For p in p0,p1,p2: n.p=d, so n is the normal vector and d the distance from the origin. n is computed from (p1-p0) x (p2-p0). Throws GeometricError if these vectors are parallel
projectS_ptline	p:P3 p0:P3 p1:P3 → t:F For a given point p, compute the parameter t of the closest point q=p0+t*(p1-p0) on the line through p0,p1. Throws GeometricError if p0, p1 coincide.
project_2vec	u:P3 v:P3 → up:P3 un:P3 Splits up v in components up,un such that v=ut+un, where up is parallel to u and un is normal to u: un.u=0. Throws GeometricError if u has zero length.
project_ptline	p:P3 p0:P3 p1:P3 → P3 For a given point p, compute the closest point q on the line through p0,p1. Throws GeometricError if p0, p1 coincide.
project_ptplane	p:P3 n:P3 d:F → q:P For a given point p, compute the closest point q on the plane (n,d).
project_polyplane	poly:[P] dir:P n:P d:F → [P] creates new polygon by projecting poly in direction dir on plane (n,d). Throws GeometricError if that's not possible.
rotskew_vec	v:P3 n:P3 alpha:F → P3 Rotates v angle alpha around axis v x (w x v). Throws GeometricError if v,w are parallel.
rot_vec	v:P3 n:P3 alpha:F → P3 Rotates vector v angle alpha around axis n. n doesn't need to be normalized Throws GeometricError if n has zero length.
rot_pt	p:P3 c:P3 n:P3 alpha:F → P3 Rotates point p an angle alpha around axis n anchored at c. n doesn't need to be normalized Throws GeometricError if n has zero length.
setlength_vec	v:P l:F → v':P

	v' is (l/|v|) v. Throws GeometricError if |v|=0.
offset_2pt	p0:P p1:P n:P w:F → p:P Computes p as a point to the left of p0, when looking along segment (p0,p1) on plane n, in distance w.
offset_3pt	p0:P p1:P p2:P n:P w:F → p:P computes the intersection p of the two lines that are parallel to segments (p0,p1) and (p1,p2), left to them in distance w. Returns p1 in case the segments are collinear (or antiparallel!)
endmove_2pt	p0:P3 p1:P3 off:N → p:P3 Computes p as p1 moved by off units in direction (p1-p0). Throws GeometricError if p0 p1 eq
endmul_2pt	p0:P3 p1:P3 fak:N → p:P3 Computes p = p0 + (p1-p0)*fak
scale_vec	(x,y,z):P3 (a,b,c):P3 → (ax,by,cz):P3 Scales a vector componentwise
offsetpolygon	[P] closed:0|1 off:F nrml:P → [P] given a polygon, computes the offset polygon off units left to the original with respect to nrml. closed=1 says that the poly is considered to be a closed one. In this case, first and last point MAY coincide. The offset poly has the same size as the original.
readpolygon	filename:S → [[Loop0] .. [LoopN]] loads a polygon from a file and puts its loops on the stack
circle	mid:P3 nrml:P3 rad:F n:I|F → [P3] mid:P3 nrml:P3 pstart:P3 n:I|F → [P3] creates an-gon around mid. If n is float, it is the arclength
circleseg	[p0:P3 p1:P3 p2:P3] nrml:P3 n:I|F mode:I→ [P3] creates a poly for the circular segment of points (p0,p1,p2). Throws GeometricError if p0=p2. mode 0: creates part of an n-gon mode 1: creates exactly n+1 points mode 2: creates segments of arclength n
circle_dir	A:P3 B:P3 v:P3: → mid:P3 A:P3 B:P3 v:P3: → 0:I computes the midpoint of a circle that contains A and B and is tangential to v in A. If (A-B) and v are collinear, simply 0 is returned.
joinsmooth	[arr1] [arr2] → [arr1 arr2] appends arr2 to arr1 and leaves the combined arr1 on the stack
joinsharp	[arr1] [arr2] → [arr1 arr2] appends arr2 to arr1 and leaves the combined arr1 on the stack
makesmooth	[arr] → [arr] removes double start and end
makesharp	[arr] → [arr] assures double start and end

Interaction Library	
ioremoveall	$\rightarrow$ removes any interaction components.
ioremove	IO $\rightarrow$ removes interaction component IO. Issues an 'NameNotFoundError' if IO is not currently active. It is LEGAL to call 'myop ioremove' from WITHIN myop's callback function. This is called IO suicide.
iopicksegment	p0 p1 f $\rightarrow$ id When the user clicks on segment p0,p1, he can drag the slider, and the point and t value are put on the stack and f is called
iopickmesh	f $\rightarrow$ IOid When the user picks the mesh, the pick point and edge are put on the stack, and f is called
iopickmeshmouse	pushL dragL releaseL pushR dragR releaseR $\rightarrow$ IOid When the user picks the mesh, the pick point and mesh edge are put on the stack, and the appropriate function is called
iogetkey	f $\rightarrow$ IOid The ascii code of key pressed is put on the stack and f is called then.
iopickfaceset	f $\rightarrow$ IOid When the user picks a new face, the pick point and face are put on the stack, and f is called. Keeps track of already picked faces
iopickfacesetget	IO $\rightarrow$ [E] IO should be an interaction component created by iopickfaceset. Such a component maintains a list of faces that have been picked. iopickfacesetget retrieves the current list from it and puts it on the stack
iopickedgeset	f $\rightarrow$ IOid When the user picks a new face, the pick point and edge are put on the stack, and f is called. Keeps track of already picked edges
iopickedgesetget	IO $\rightarrow$ [E] IO should be an interaction component created by iopickedgeset. Such a component maintains a list of edges that have been picked. iopickedgesetget retrieves the current list from it and puts it on the stack
iopickray	push_f drag_f release_f $\rightarrow$ IOid Creates an interaction component that puts e:E p0:P3 p1:P3 t:Float b:Int on the stack when the user picks. e is the edge that was picked, p0,p1 is the ray that was depicted with the mouse (on front/backplane), t is the distance on the ray to the first hit (1.0 if nothing was hit), and b is the button (L,R,M: b=0,1,2). According to the button action, one of

	push_f, drag_f, and release_f is called.
iopickquad	p0:P3 v:P3 w:P3 f → IOid Creates an interaction component that puts (s,t):P2 b:Int m:Int on the stack when the user picks the quad q(s,t)=p0+sv+tw, s,t in [0,1], and calls f. b is the mouse button and m is the button action: left,right,middle → b=0,1,2 push,drag,release → m=0,1,2
renderstring	x p:P3 (s,d,m):P3 → IO:id renders x as 3D text at point p, so that it always faces the viewer. s is the font size, d the shift towards the viewer, and m the material (int between 0 and 30).
rendertext	x p:P3 dir:P3 up:P3 mat:I → IO:id renders x as 3D text at point p, with text direction (dir,up). with material mat (0<=mat<=30).
iorendertextchange	x p:P3 dir:P3 up:P3 mat:I io:IO → IO should be created by rendertext. For the signature see there.
iospacemouse	f → IO:id Every time the user changes the spacemouse, the (x,y,z) and (a,b,c) vectors of translation and rotation are put on the stack, and f is called
iorenderhook	f → IO:id The function f is called in EVERY FRAME. Be cautious with this op! It's highly fps sensitive.
screenshot	filename:string → saves a screenshot as .ppm file
showpatch	e:E → saves a screenshot as .ppm file

Materials Library	
getmaterialnames	getMaterials → [N] returns the name of all currently loaded materials.
setcurrentmaterial	matname:N → sets the current material to be used for new faces
GLmaterial	matname:N → activates the current material for immediate OpenGL rendering
getfacematerial	e:E getMaterials → matname:N returns the materialname of e's baseface, or /none in case it's undefined
savemeshwithmaterials	filename:string → saves the current mesh in .obj format, together with materials
setfacematerial	e:E matname:N → sets the material of e's baseface

ImmediateRender	
glPushMatrix	→
glPopMatrix	→

glRotate	axis:P angle:F →
glTranslate	offset:P →
glScale	scalefactor:F → performs uniform scaling
meshlib-create	filename:S → id:I Creates a meshlib object. Supported file formats: "object.pm": progressive mesh "object.obj": combined BRep "object.ttf": 3D text
meshlib-type	id:I → /none id:I → /MeshCBRep id:I → /MeshPM id:I → /Text3D returns the type of a given meshlib object.
meshlib-deleteall	→ deletes all allocated meshlib objects.
meshlib-pm-createinstance	id:I → instid:I Creates a new instance of a progressive mesh.
meshlib-pm-setLOD	lod:F instid:I id:I → sets the level of detail of pm instance instid of progressive mesh id to lod
meshlib-pm-render	instid:I id:I → renders instance instid pm with index id
meshlib-text-render	x id:I → renders x as 3D text to OpenGL, using font id
meshlib-text-renderfront	t noOfLines:F p:P id:I → renders t as 3D text on screen divided in noOfLines of text on position p=(x,y,z) with (0,0) in the upper left corner, and z (depth) in range [-10,10], using font id