OpenSDraw Documentation

Release 0.0.2

Hazen Babcock

Sep 04, 2017

Contents

A CAD program for designing LEGO(R) MOCs. OpenSDraw is a domain specific language for creating LEGO(R) models. It is a prefix notation language similar to Scheme or LISP that is used to specify the location of the bricks in the MOC. The output of a program is a LDraw format file that can be read by any LDraw compatible viewer such as LDView.

This project was inspired by OpenSCAD.

CHAPTER 1

Download

The OpenSDraw project can be obtained here.

CHAPTER 2

User's Guide

User's Guide

Dependencies

- LDraw
- LDView (For the partviewer).
- numpy
- Python
- PyQt5 (For the partviewer).
- requests (For the partviewer).
- rply
- scipy

Setup

Python path

The project root directory needs to be in your Python path. One way to do this is to edit the *opensdraw.pth* file to have the correct path, then copy this file into your Python dist-packages directory. An example file:

```
/home/username/Downloads/opensdraw/
```

LDraw path

Edit the path in *opensdraw/xml/ldraw_path.xml* to point to your LDraw directory (not the parts sub-directory). An example file:

```
<?xml version="1.0" encoding="utf-8"?>
<ldraw-path>
 <path path="/home/username/Downloads/ldraw/"/>
</ldraw-path>
```

Emacs

Any text editor can be used to create .lcad files, however emacs integration is provided. The following steps should enable this:

1. Add a sub-folder to your .emacs.d directory called *lcad-mode*.
2. Copy the *lcad-mode.el* file into this directory.
3. Edit the path to *lcad_to_ldraw.py* in the **compile()** function in *lcad-mode.el*.
4. Add the following to your .emacs file.

```
(add-to-list 'load-path "~/.emacs.d/lcad-mode")
(require 'lcad-mode)
(add-hook 'lcad-mode-hook 'lcad-disable-slime) ; You only need this if you also␣
↪use the SLIME mode.
```

Once everything is setup up this will provide syntax high-lighting and pressing **F5** will automatically convert your .lcad file to a .mpd file as well as saving it.

Usage

The basic work flow is:

1. Use the partviewer to determine the LDraw part number and LDraw color of the part you wish to add to your MOC.

```
cd /path/to/opensdraw/opensdraw/partviewer
python partviewer.py
```

Note: The first time this is run it will take a while (15 - 30 minutes) to generate the thumbnails of all the parts. By default it will use LDView with your current preferences to do this. This means that LDView must be in your path. Also if you have "Show Axes" checked for example then all your parts will be rendered with the X,Y and Z axises. It may also be a good idea to temporarily disable your screen-saver.

Note: This program will create a file called "ldview_part.mpd" that you can view with LDView (or equivalent). This file is updated with the current selected part and color.

Note: This program can query Rebrickable to provide more detailed part information such as what years it was available and in what colors. You will need an account at Rebrickable and an API key for this to work.

---

2. Edit your MOC .lcad file to include this part in the desired location.
3. Convert the MOC .lcad file to a .mpd file using *lcad_to_ldraw.py*.

```
python /path/to/opensdraw/scripts/lcad_to_ldraw.py file.lcad file.mpd
```

4. Visualize the .mpd file with LDView (or equivalent).

Example .lcad files are provided in the examples directory.

Note: LDView can be configured to automatically poll for changes to .mpd files.

Understanding Error Messages

Occasionally things will go wrong and you will get a possibly long and confusing back-trace like this:

```
!Error in function 'chain1' at line 75 in file 'chain.lcad'

Traceback (most recent call last):
  File "../scripts/lcad_to_ldraw.py", line 46, in <module>
    model = interpreter.execute(ldraw_file_contents, filename = sys.argv[1], time_
↪index = index)
  File "/home/hbabcock/Code/opensdraw/opensdraw/lcad_language/interpreter.py", line␣
↪335, in execute
    interpret(model, ast)
  File "/home/hbabcock/Code/opensdraw/opensdraw/lcad_language/interpreter.py", line␣
↪422, in interpret
    ret = interpret(model, node)
  File "/home/hbabcock/Code/opensdraw/opensdraw/lcad_language/interpreter.py", line␣
↪407, in interpret
    val = dispatch(func, model, tree)
  File "/home/hbabcock/Code/opensdraw/opensdraw/lcad_language/interpreter.py", line␣
↪311, in dispatch
    func.argCheck(tree)
  File "/home/hbabcock/Code/opensdraw/opensdraw/lcad_language/functions.py", line 115,
↪ in argCheck
    raise lce.NumberArgumentsException(self.min_args, len(args))
opensdraw.lcad_language.lcadExceptions.NumberArgumentsException: !Error, wrong number␣
↪of standard arguments, got 0 expected 1
```

This trace consists of 3 parts:

1. One or more lines telling you what line in the .lcad file caused the problem.
2. A Python traceback.
3. A final line containing the exception that was triggered and some additional information.

At some point in the future the Python traceback may disappear, but at present I don't yet have enough confidence that the .lcad traceback alone is always sufficient to figure out what went wrong.

Dumper Truck Example

This recreates the Dumper Truck example from here using OpenSDraw.

Step 1

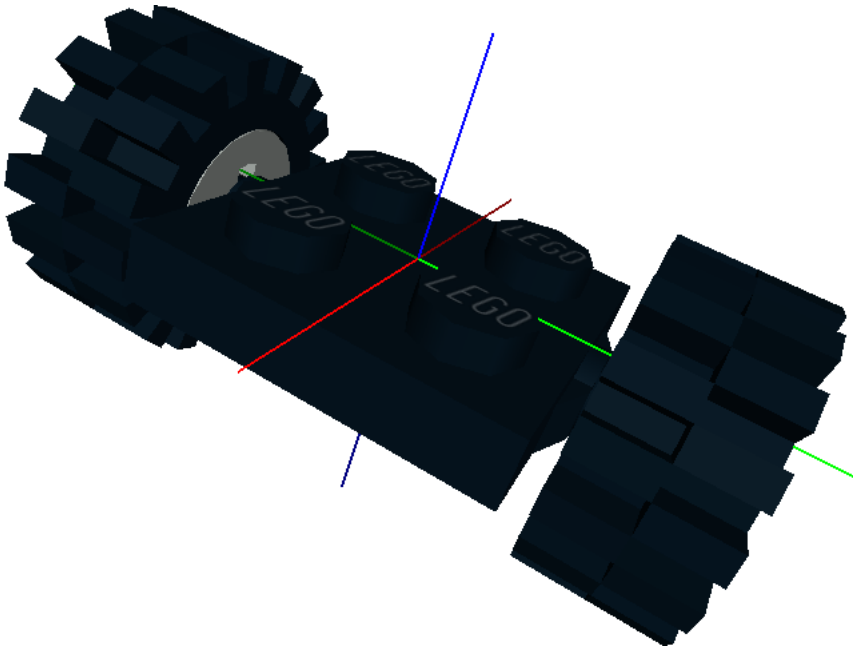
Import the *locate* library for easier part placement.

```
(import locate :local)
```

Step 2

Create the wheel assembly.

```
(def wheel-assembly ()
 (block

  ; A. Plate 2 x2 with Wheel Holders.
  (sb 0 0 0 -90 90 0 "4600" "Black")

  ; B. Wheel Rim 6.4 x 8.
  (sb 0 1.5 -0.2 90 0 0 "4624" "Light_Gray")

  ; C. Tyre 6/50 x 7 Offset Tread.
  (sb 0 1.5 -0.2 90 0 0 "3641" "Black")

  ; D. Mirror above along the y-axis to add the other wheel and tyre.
  (mirror (list 0 1 0)
   (sb 0 1.5 -0.2 90 0 0 "4624" "Light_Gray")
   (sb 0 1.5 -0.2 90 0 0 "3641" "Black"))

  ))
```



Note: The arguments for the **sb()** function (standard brick) are x, y, z position in bricks, x-axis, y-axis, z-axis rotation in degrees, LDraw part number, part color.

Note: You can use either numbers or names for part colors.

Step 3

Create the truck body.

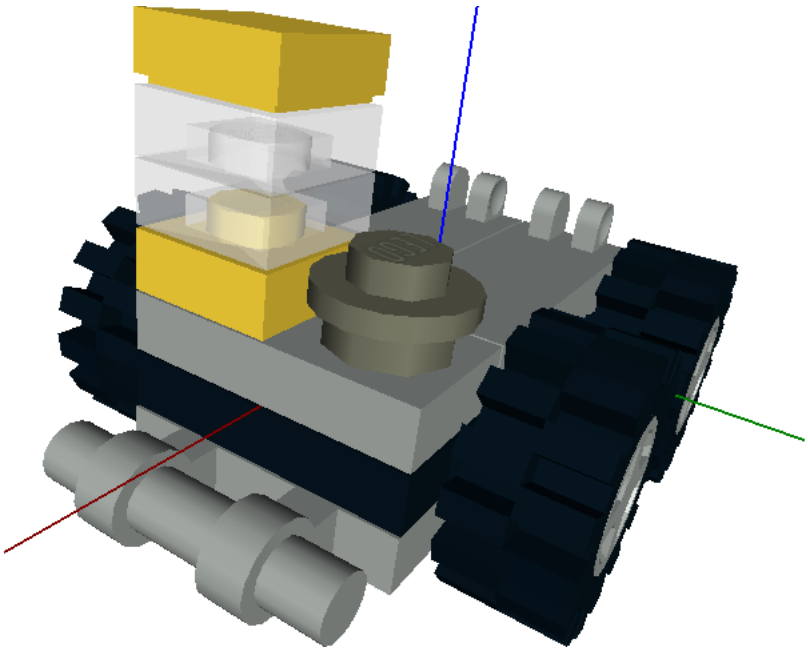
```
(def truck-body ()
 (block

  ; A. Wheel assembly
  (translate (list (bw -1) 0 0)
   (wheel-assembly))

  (translate (list (bw 1) 0 0)
   (wheel-assembly))

  ; B. Plate 1 x 2.
  (sb -1.5 0 0.33 -90 90 0 "3023" "Light_Gray")

  ; C. Hinge Tile 1 x 2 with 2 Fingers.
  (sb 0 0.5 0.66 -90 0 0 "4531" "Light_Gray")
  (sb 0 -0.5 0.66 -90 0 0 "4531" "Light_Gray")

  ; D. Plate 1 x 1.
  (sb -1.5 0.5 0.66 -90 90 0 "3024" "Yellow")
  (sb -1.5 0.5 1.0 -90 90 0 "3024" "Trans_Clear")
  (sb -1.5 0.5 1.33 -90 90 0 "3024" "Trans_Clear")

  ; E. Tile 1 x 1 with Groove.
  (sb -1.5 0.5 1.66 -90 90 0 "3070b" "Yellow")

  ; F. Plate 1 x 1 Round.
  (sb -1.5 -0.5 0.66 -90 90 0 "4073" "Dark_Gray")

  ; G. Plate 1 x 2 with Handle.
  (sb -1.5 0 -0.33 -90 -90 0 "2540" "Light_Gray")

  ))
```

Note: The **bw()** function converts brick widths to LDU (LDraw units).

Step 4

Create the dumper assembly.

```
(def dumper-assembly ()

 ; Make (0,0,0) the pivot point.
 (translate (list (bw -1.5) 0 (bh 0.15))

  ; A. Hinge Plate 1 x 2 with 3 Fingers and Solid Studs.
  (sb 0 0.5 0.0 -90 0 0 "4275b" "Yellow")
```



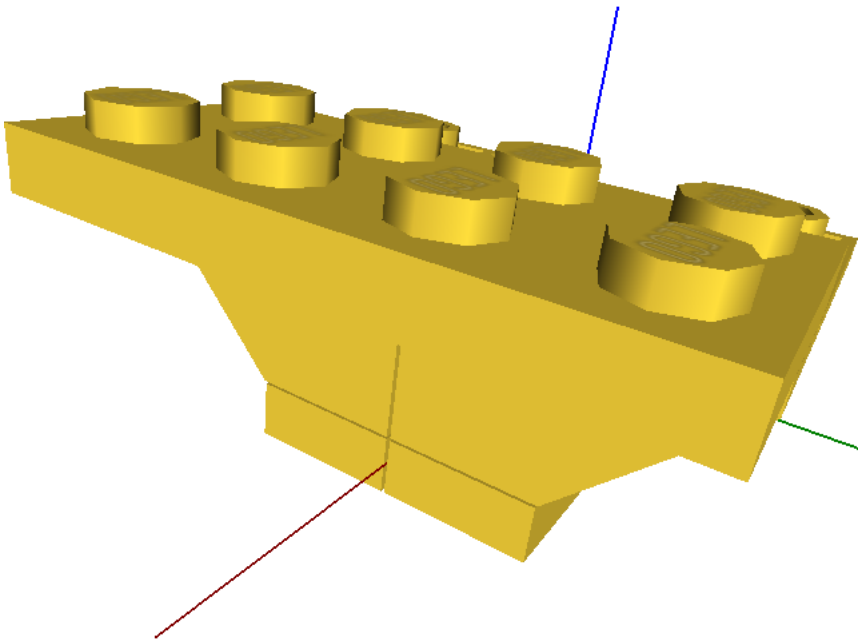
```
  (sb 0 -0.5 0.0 -90 0 0 "4275b" "Yellow")

  ; B. Slope Brick 45 2 x 1 Inverted.
  (sb -0.5 -0.5 1 -90 0 0 "3665" "Yellow")
  (sb -0.5 0.5 1 -90 180 0 "3665" "Yellow")

  ; C. Slope Brick 45 4 x 2 Double Inverted with Open Center.
  (sb 1 0 1 -90 0 0 "4871" "Yellow")

  ; D. Plate 2 x 4.
  (sb -1 0 1.33 -90 90 0 "3020" "Yellow")

  ))
```

Note: The **bh()** function converts brick heights to LDU (LDraw units).

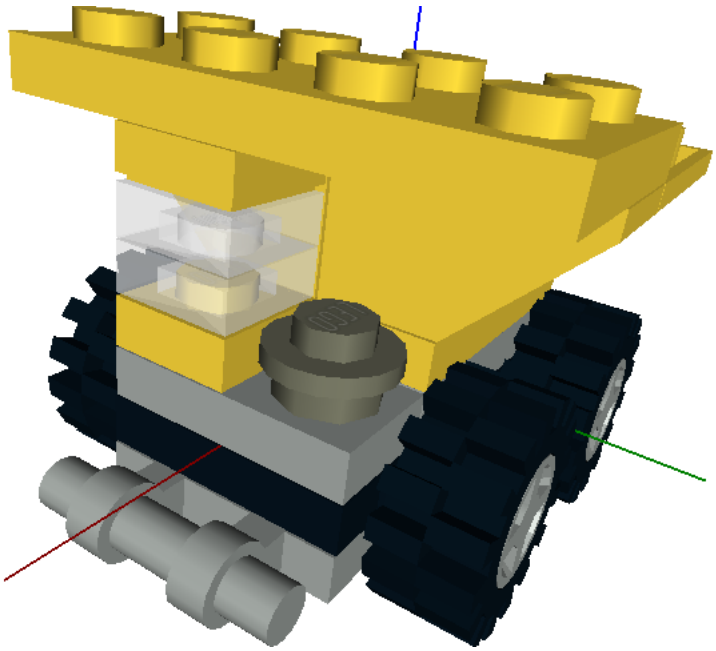
Step 5

Put everything together with a tilt option.

```
(def truck (tilt)
 (block

  ; A. Truck body.
  (truck-body)

  ; B. Dumper assembly.
  (translate (list (bw 1.5) 0 (bh 0.5))
   (rotate (list 0 (- tilt) 0)
    (dumper-assembly)))
```



```
  ))
```

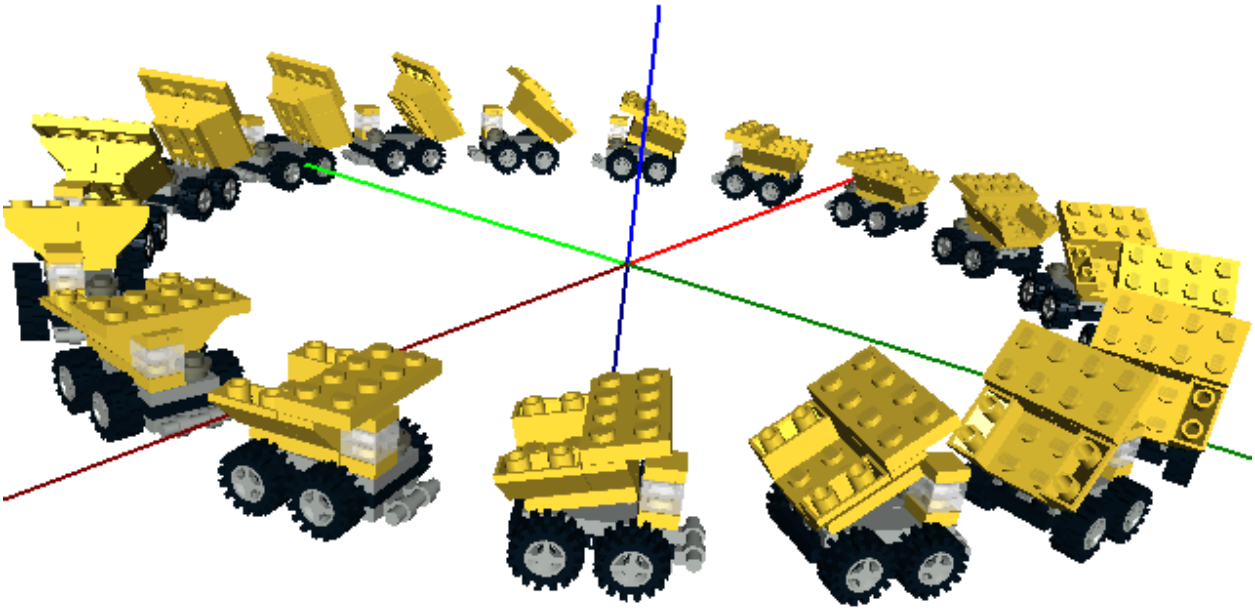


Step 6

Draw a ring of 18 trucks with different tilts.

```
(for (i 18)
 (rotate (list 0 0 (* i 20))
```

```
  (translate (list 0 (bw 20) 0)
   (truck (* 30.0 (+ 1 (cos (/ (* i 40 pi) 180))))))))
```



Note: The complete code is in the examples folder (dumper-truck.lcad).

Step 7

Convert the .lcad file to a .mpd file using *lcad_to_ldraw.py*.

```
cd opensdraw/opensdraw/examples
python ../scripts/lcad_to_ldraw.py dumper-truck.lcad
```

Build Steps Example

How to add build step information to your model using OpenSDraw.

Step 1

Import the *ldu* conversion library.

```
(import ldu :local)
```

Step 2

Create the .lcad file.

```
(for (i 20)
 (rotate (list 0 (* i (/ 360.0 20)) 0)
  (translate (list (bw 6.3) (if (= (% i 2) 0) 0 (bw 1)) 0)
   (part "32523" 14 i)))

 (rotate (list 0 (+ (* i (/ 360.0 20)) (* 0.5 (/ 360.0 20))) 0)
  (translate (list (bw 6.4) (bw 0.5) 0)
   (rotate (list 0 0 90)
    (part "3673" "black" i)))))
```

Note: The **part()** function takes an optional third argument which is the build step number.

Note: The step number does not have to be an integer, floating point numbers are also ok. Steps are ordered using the Python **sorted()** function.

Note: We don't use the *locate* library functions **sbs()** or **tbs()** because we want to translate first, then rotate.

Step 3

Convert the .lcad file to a .mpd file using *lcad_to_ldraw.py*.

```
cd opensdraw/opensdraw/examples
python ../scripts/lcad_to_ldraw.py steps.lcad
```

Step 4

Load the .mpd file with your favorite viewer (LDView renderings shown here).

Note: The complete code is in the examples folder (steps.lcad).

Note: There is also a global *step-offset* symbol, see examples/auto-step.lcad for an example of how to use this.

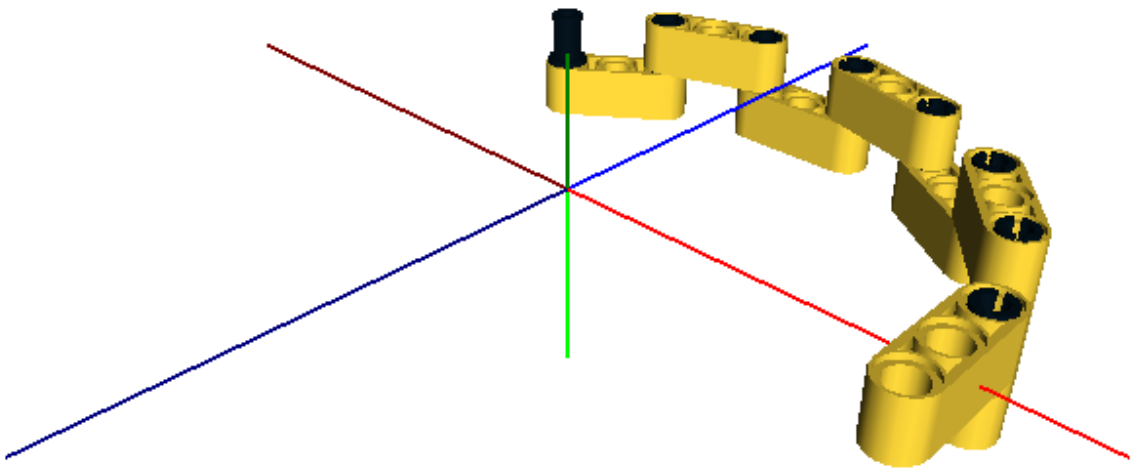


Fig. 2.1: Step 8

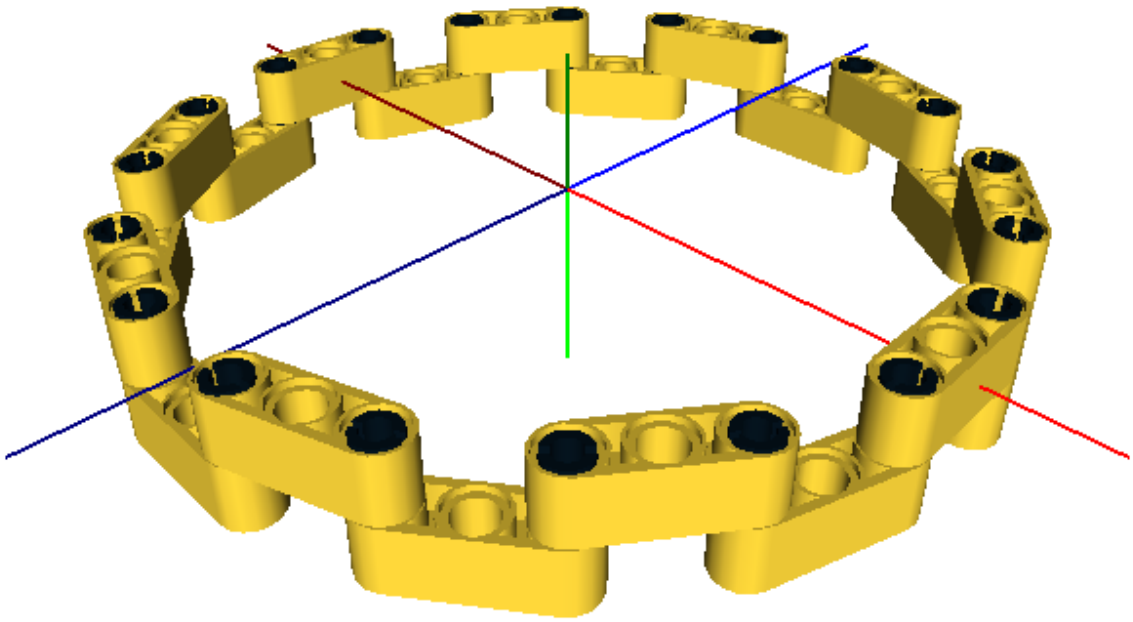


Fig. 2.2: Step 20

Animation Example

How to create animations using OpenSDraw. Useful ideas about how to configure LDView for this purpose can be found here.

Step 1

Create the lcad file.

```
(import locate :local)

(def axle1 ()
 (block

  ; Axle 4
  (tb 0 0 -0.5 0 90 0 "3705" "Black")

  ; Gear 8 Tooth
  (tb 0 0 0 0 0 0 "3647" "Dark_Gray")

  ))

(def axle2 ()
 (block

  ; Axle 5
  (tb 0 0 0 0 90 0 "32073" "Light_Gray")

  ; Gear 8 Tooth
  (tb 0 0 1 0 0 0 "3647" "Dark_Gray")

  ; Gear 24 Tooth with Single Axle Hole
  (tb 0 0 -1 0 0 0 "3648b" "Dark_Gray")

  ))

(def axle3 ()
 (block

  ; Axle 5
  (tb 0 0 0 0 90 0 "32073" "Light_Gray")

  ; Gear 24 Tooth with Single Axle Hole
  (tb 0 0 1 0 0 0 "3648b" "Dark_Gray")

  ))

(def angle1 (* time-index 5))
(def angle2 (+ 7.5 (/ angle1 3)))
(def angle3 (+ 7.5 (/ angle2 3)))

(translate (list 0 0 (bw -1))
 (rotate (list 0 0 angle1)
  (axle1)))
```

```
(translate (list 0 (bw 2) 0)
 (rotate (list 0 0 (- angle2))
  (axle2)))

(translate (list (bw 2) (bw 2) 0)
 (rotate (list 0 0 angle3)
  (axle3)))
```

Note: **time-index** is the animation variable. It will count up from 0 in increments of 1.

Note: This is the *gears.lcad* file in the examples folder.

Step 2

Create a directory to save the .dat files in, change to this directory and generate the dat files.

```
cd opensdraw/examples
mkdir animate
cd animate
python ../../scripts/lcad_to_ldraw.py ../gears.lcad gears.mpd 100
```

Note: This will make 100 different versions of the gears.dat file (time-index = 0..99).

Step 3

Generate the png files in the same directory.

```
python ../../scripts/ldview_render.py ./
```

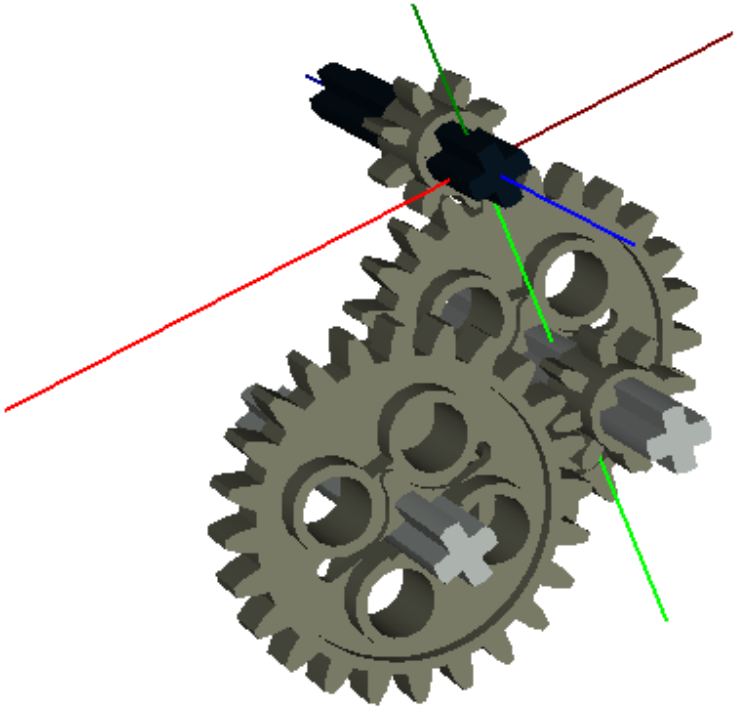
Note: Edit the options in *ldview_render.py* depending on the desired results (point of view, background color, etc..)

Step 4

Create a movie. I find ImageJ be a handy tool for this. You can import the series of .png files as an image sequence (File -> Import -> Image Sequence).

See Also

The belt.lcad example, which demonstrates animating a 3D chain and sprocket system. The chain.lcad example, which demonstrates animating a 2D chain and sprocket system.



Interfacing with Python

Writing your own Python modules that you can use with OpenSDraw.

Step 1

Create the Python file.

```
#!/usr/bin/env python

import numbers
import os
from PIL import Image

# Define the basestring type for Python 3.
try:
    basestring
except NameError:
    basestring = str

# These OpenSDraw modules have some classes that we will use.
import opensdraw.lcad_language.interpreter as interpreter
import opensdraw.lcad_language.typeFunctions as typeFunctions

# This OpenSDraw module defines some types that we will use.
import opensdraw.lcad_language.lcadTypes as lcadTypes

# This OpenSDraw module defines some exceptions that we will use.
import opensdraw.lcad_language.lcadExceptions as lcadExceptions

# OpenSDraw will look for this dictionary to figure out what functions to import.
lcad_functions = {}

#
# Your function(s) (like all functions in OpenSDraw) should be a sub-class of
# the interpreter.LCadFunction class.
#
# This class will open a user requested picture and return another class that
# the user can use to access various properties of the picture.
#
class OpenPicture(interpreter.LCadFunction):

    def __init__(self):

        # interpreter.LCadFunction.__init__ takes one argument, the name of the␣
↪function.
        interpreter.LCadFunction.__init__(self, "picture")

        # Set the function signature so that the interpreter will type check for a␣
↪single
        # argument of type string (the name of the picture file). Use basestring since
        # this will also work with unicode strings.
        self.setSignature([[basestring]])

    #
    # model is an instance of interpreter.Model. This stores the parts, groups,
```

```
    #    primitives and etc. This is the first argument to every function.
    #
    # filename is the name of the file.
    #
    # When you call this function the interpreter will check the arguments based on
    # the function signature provided in __init__(). The arguments to call should
    # match those in the signature.
    #
    def call(self, model, filename):

        # Check that the requested picture exists.
        if os.path.exists(filename):

            # Return an instance of the Picture class.
            return Picture(Image.open(filename))

        # If not, throw an exception.
        else:
            raise lcadExceptions.LCadException("picture " + filename + " not found.")

# Make sure to add an instance of your function to the functions dictionary.
lcad_functions["open-picture"] = OpenPicture()


#
# This class will return either the picture size, or the color at a particular
# pixel depending on the arguments that the user supplies
#
class Picture(interpreter.LCadFunction):

    def __init__(self, im):
        interpreter.LCadFunction.__init__(self, "user created picture function")

        # Store the PIL Image object.
        self.im = im

        # Set signature to be exactly two arguments both of which are numbers
        # or the symbols t/nil.
        self.setSignature([[numbers.Number, lcadTypes.LCadBoolean], [numbers.Number,␣
↪lcadTypes.LCadBoolean]])

    def call(self, model, x, y):

        # If we got t/nil return the size of the picture.
        # (Note: To check for Truth use 'functions.isTrue(val)').
        if typeFunctions.isBoolean(x) or typeFunctions.isBoolean(y):
            return list(self.im.size)

        # Otherwise return the color of the pixel as a LDraw "direct" color. Best
        # practice might be to do some range checking. This will also fail for
        # certain types of images (such as .gif).
        [r, g, b] = self.im.getpixel((x, y))

        # Convert colors (0-255) to upper case hex & concatenate.
        return "0x2" + "".join(map(lambda x: "{0:#0{1}x}".format(x,4).upper()[2:], [r,
↪ g, b]))
```

Note: This is the *picture.py* file in the examples folder.

Step 2

Create the lcad file.

```
(import locate :local)

; Import of picture.py module.
(pyimport picture)

; Load the example picture.
(def pic (open-picture "9393.png"))

; Get the picture size.
(def size (pic t t))
(def size-x (aref size 0))
(def size-y (aref size 1))

; Create picture backing. This works best for pictures whose size
; is a multiple of 2. Also, some of the corners can end up as 2 x 2
; on top of 2 x 2 which would not actually work.
(def stripe (y offset length)
 (block
  (def pos offset)
  (if (= (% offset 2) 0)
    (block
     (sb pos y 0 90 0 0 "3022" 7)
     (set pos (+ pos 3)))
   (block
    (sb pos y 0 90 0 0 "3020" 7)
    (set pos (+ pos 4))))
  (while (< pos (- length 2))
   (sb pos y 0 90 0 0 "3020" 7)
   (set pos (+ pos 4)))
  (if (< pos length)
    (sb (- pos 1) y 0 90 0 0 "3022" 7))))

; Backing upper layer.
(for (y (/ (+ size-y 2) 2))
 (translate (list (bw -0.5) (bw -0.5) (bh 0.3))
  (stripe (* 2 y) (% (+ y 1) 2) (+ size-x 2))))

; Backing lower layer.
(for (x (/ (+ size-x 2) 2))
 (translate (list (bw (- size-x 0.5)) (bw -0.5) (bh 0.6))
  (rotate (list 0 0 90)
   (stripe (* 2 x) (% x 2) (+ size-y 2)))))

; re-create the picture using Plate 1 x 1 Round.
(for (i size-x)
 (for (j size-y)
  (sb i j 0 90 0 0 "4073" (pic i j))))
```

Note: This is the *picture.lcad* file in the examples folder.

Step 3

Convert the .lcad file to a .mpd file using *lcad_to_ldraw.py*.

```
cd opensdraw/opensdraw/examples
python ../scripts/lcad_to_ldraw.py picture.lcad
```

Note: Large pictures can easily overwhelm both OpenSDraw and LDView. Picture size less than 100 x 100 or so are probably the best.

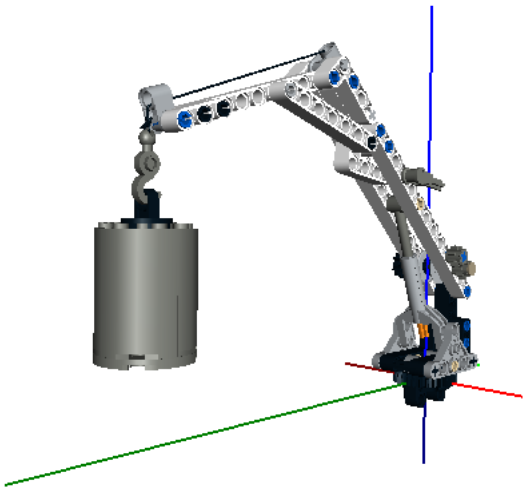
Also

If you want your module to always be available you can add it to the *lcad_language/modules.xml* file.

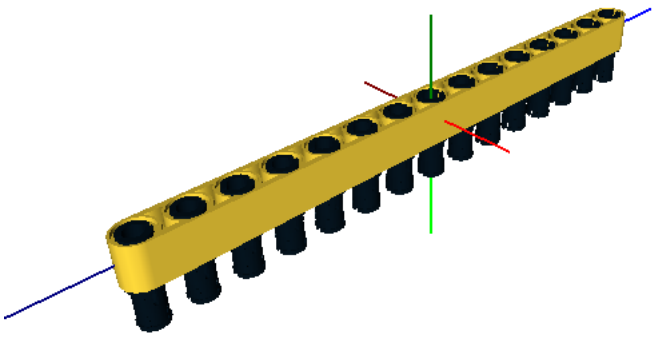
Summary of the Examples

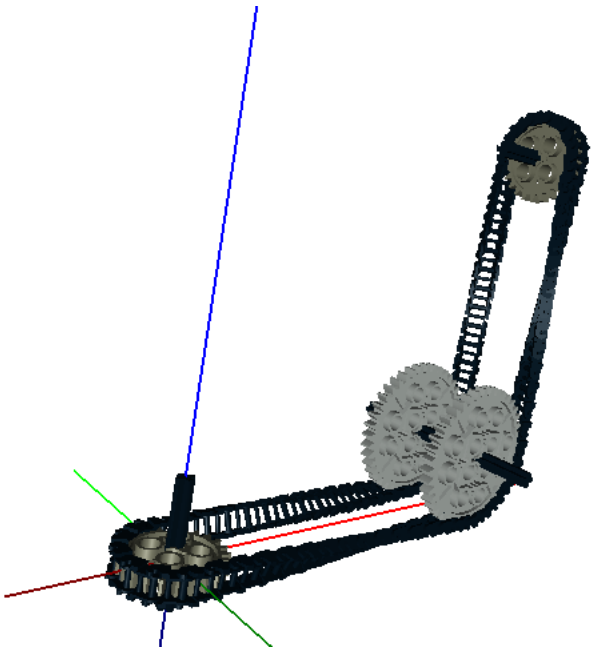
All of the examples can be found in the *examples* directory.

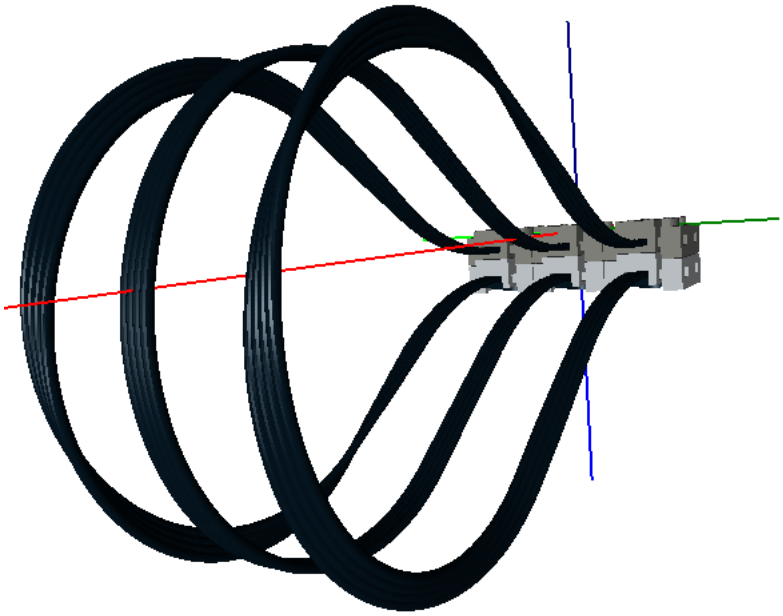
1. *artic-truck-crane.lcad* - Pulley system creation and animation.



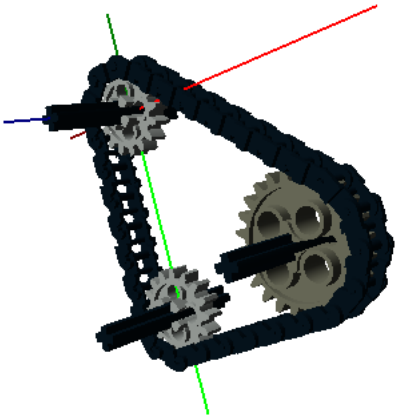
2. *auto-step.lcad* - An automatic build steps example.
3. *belt.lcad* - 3D chain and sprocket system creation and animation.
4. *callback.lcad* - Using callbacks to add twist to a curve function.



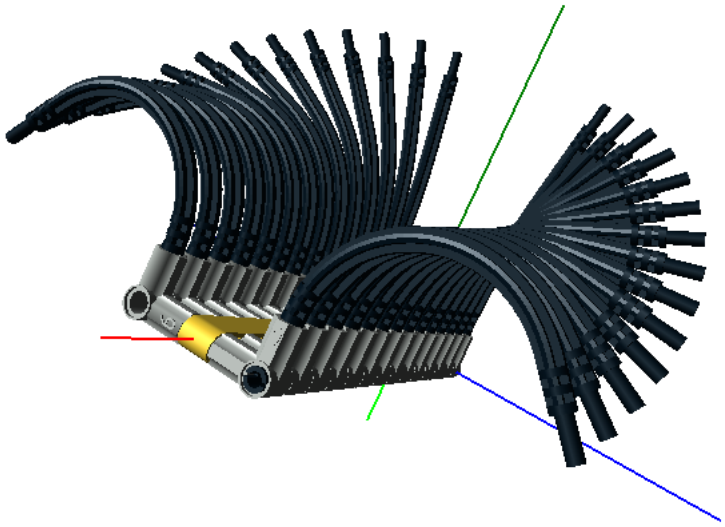


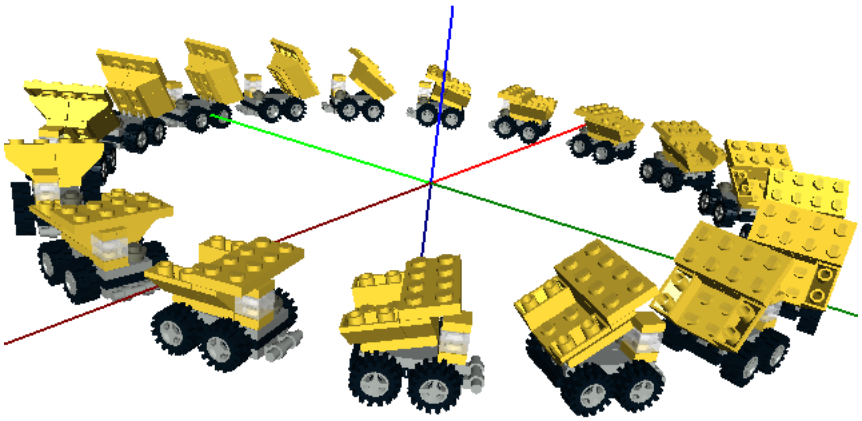


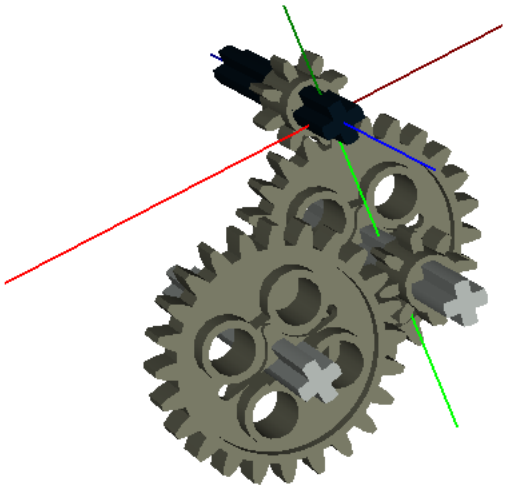
5. *chain.lcad* - 2D chain and sprocket system creation and animation.

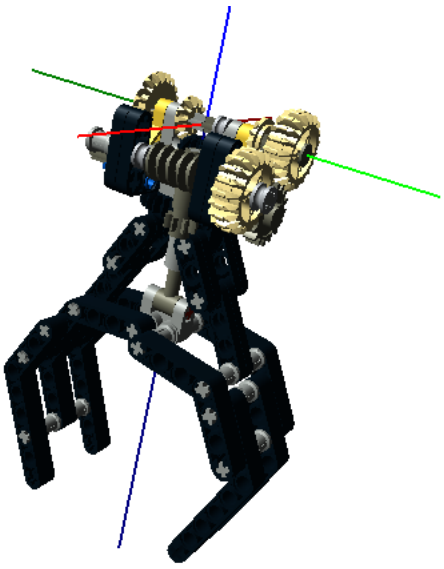


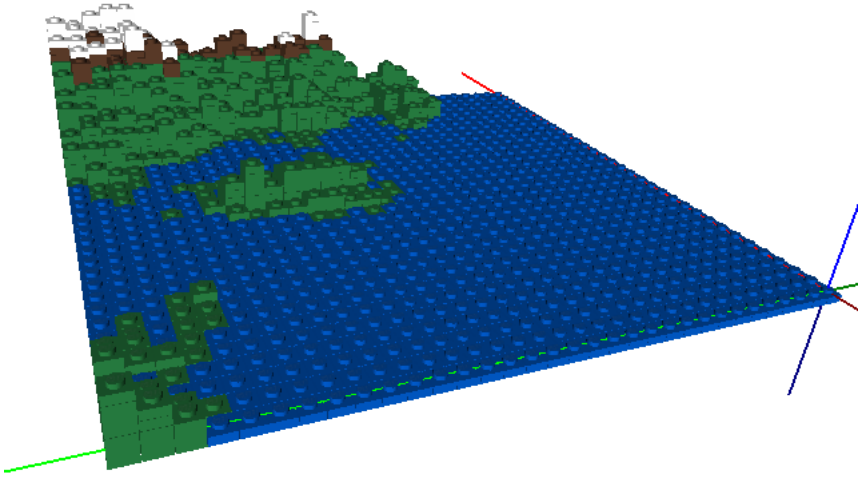
6. *curve.lcad* - Curve creation example.
7. *dumper-truck.lcad* - The Dumper Truck example from here.
8. *gears.lcad* - An animation example.
9. *gripper.lcad* - A gripper, original design by Efferman from here.
10. *landscape.lcad* - Create a model from a simple space delimited text file.
11. *overlay.lcad* - Overlay pictures / blueprints for model scaling.

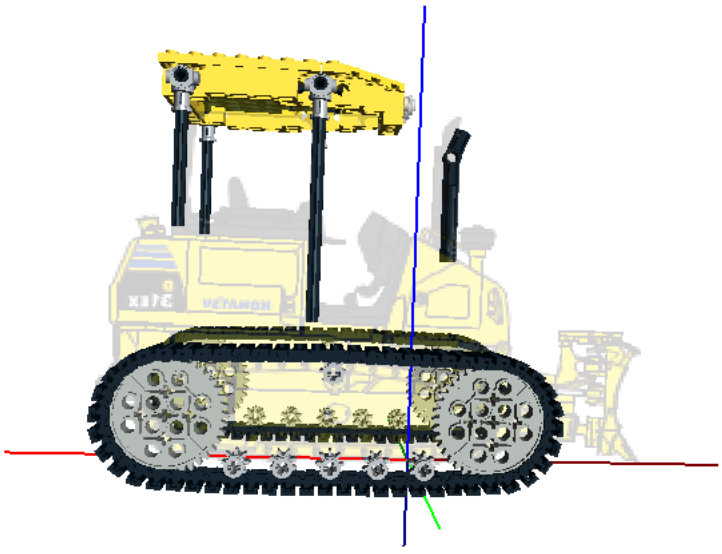






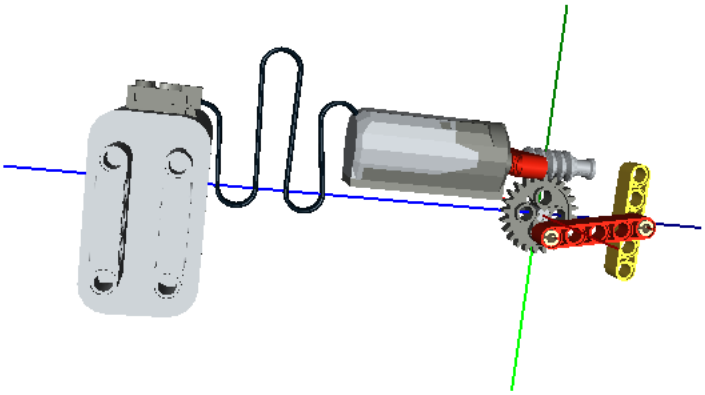




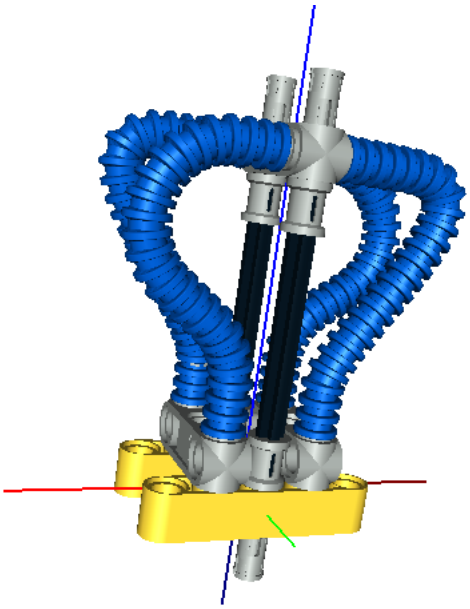


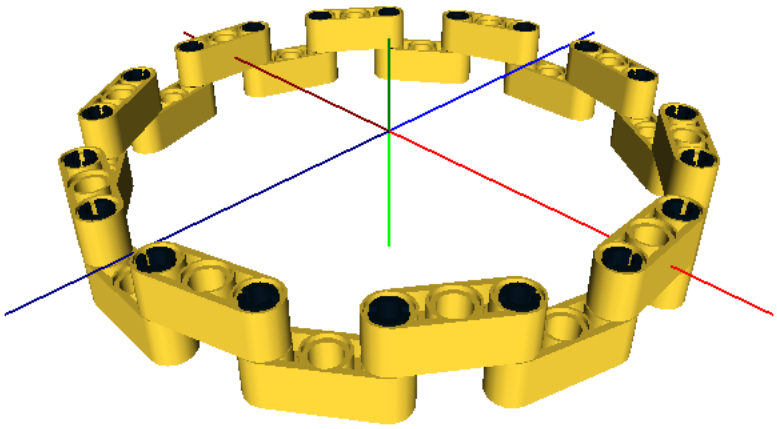
12. *picture.lcad* - Interfacing with Python example.

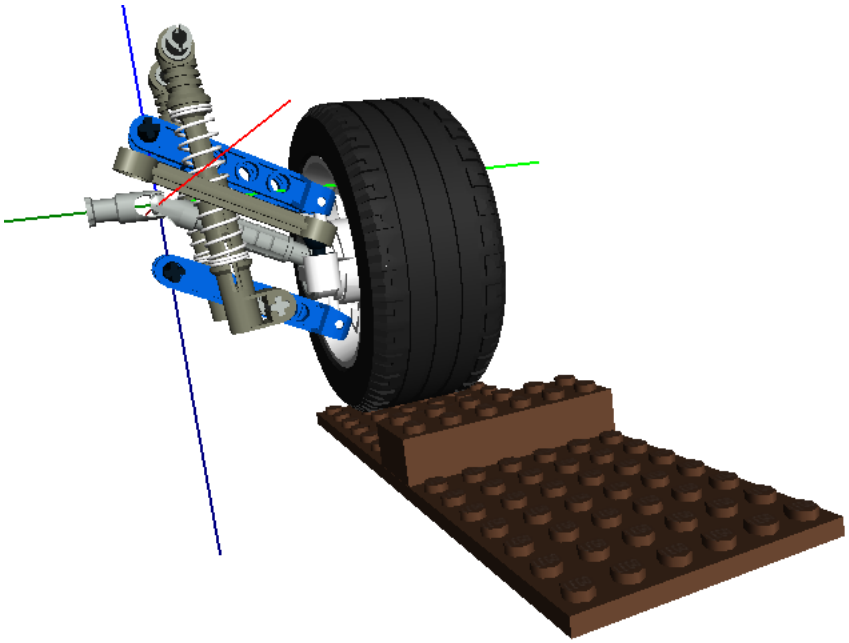
13. *power-functions-cable.lcad* - 4 Wire power functions cable creation and animation.

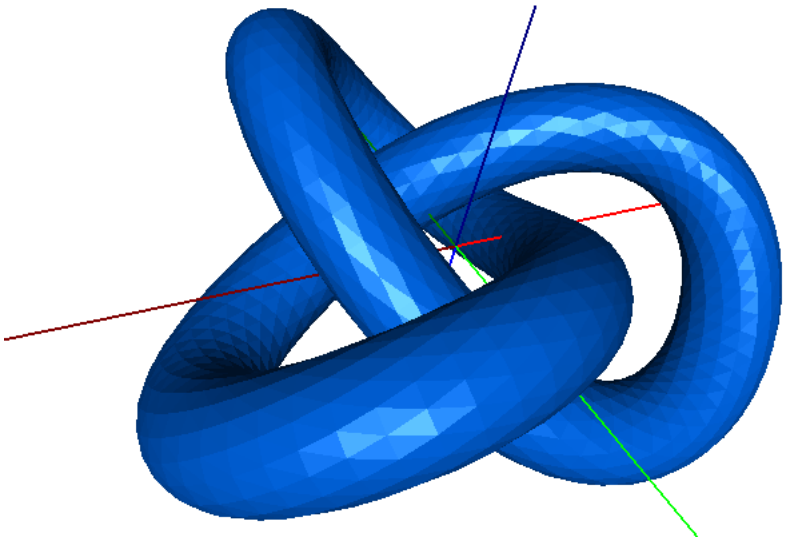


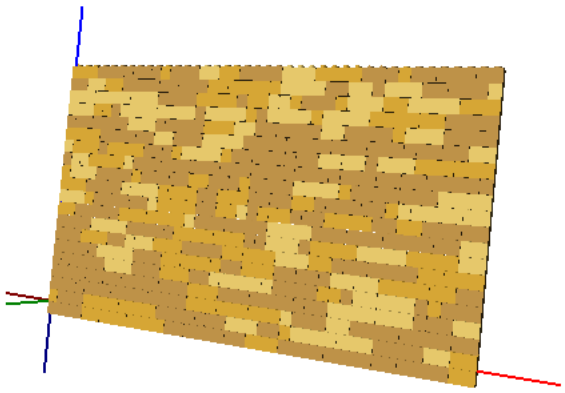
14. *rib-hose.lcad* - Curves and ribbed-hose example.
15. *steps.lcad* - A build steps example.
16. *suspension.lcad* - A spring creation and animation example.
17. *trefoil.lcad* - A LDraw primitives example.
18. *wall.lcad* - A random number generator example.











CHAPTER 3

Language Documentation

Language

Symbols

- **t** - True
- **nil** - False
- **pi** - 3.14159
- **e** - 2.7182
- **step-offset** - User settable to a number, or a function with no arguments that returns a number. This number is added to the *part_step* parameter of the **part()** function. The default value is 0.
- **time-index** - 0..N, for creating animations.

Functions

class opensdraw.lcad_language.coreFunctions.**Append**
append - Add one or more elements to a list.

Usage:

```
(def l (list 1))  ; Create the list (1).
(append l 2)      ; The list is now (1, 2).
```

class opensdraw.lcad_language.coreFunctions.**Aref**
aref - Return an element of a list, vector or matrix.

Usage:

```
(aref (list 1 2 3) 1)            ; returns 2
(set (aref (list 1 2 3) 1) 4)    ; list is now 1, 4, 3
(aref (vector 1 2 3) 1)          ; returns 2
(set (aref (vector 1 2 3) 1) 4) ; vector is now 1, 4, 3
```

class opensdraw.lcad_language.coreFunctions.**Block**

block - A block of code, similar to *progn* in Lisp.

Usage:

```
(block
  (def x 15)     ; local variable x
  (def inc-x () ; function to modify x (and return the current value).
    (+ x 1))
  inc-x)         ; return the inc-x function
```

class opensdraw.lcad_language.coreFunctions.**Concatenate**

concatenate - Concatenate 1 or more strings.

Usage:

```
(concatenate "as" "df")  ; Returns "asdf".
(concatenate "as" 1)     ; Returns "as1".
```

class opensdraw.lcad_language.coreFunctions.**Cond**

cond - Switch statement.

Usage:

```
(cond
  ((= x 1) ..) ; do this if x = 1
  ((= x 2) ..) ; do this if x = 2
  ((= x 3) ..) ; do this if x = 3
  (t ..))      ; otherwise do this
```

class opensdraw.lcad_language.coreFunctions.**Copy**

copy - Make a copy.

Usage:

```
(def a (list 1 2 3) b (copy a)) ; Make two independent lists with elements 1, 2,␣
↪3.
```

class opensdraw.lcad_language.coreFunctions.**Def**

def - Create a variable or function.

Usage:

```
(def x 15) - Create the variable x with the value 15.
(def x 15 y 20) - Create x with value 15, y with value 20.
(def incf (x) (+ x 1)) - Create the function incf.
```

Note: You cannot create multiple functions at the same time.:

```
(def fn (x y)  ; Wrong. def() will think you are trying to create
 (+ x 1)       ; two symbols, the first called 'fn' and the second
 (+ y 2))      ; called '(+ x 1)' and throw a likely confusing error
               ; message.
```

```
(def fn (x y)  ; Correct.
 (block
  (+ x 1)
  (+ y 2)))
```

class opensdraw.lcad_language.coreFunctions.**For**

for - For statement.

Usage:

```
(for (i 10) ..)           ; increment i from 0 to 9.
(for (i 1 11) ..)         ; increment i from 1 to 11.
(for (i 1 0.1 5) ..)      ; increment i from 1 to 5 in steps of 0.1.
(for (i (list 1 3 4)) ..) ; increment i over the values in the list.
```

class opensdraw.lcad_language.coreFunctions.**If**

if - If statement.

The first argument must be t or nil.

Usage:

```
(if t 1 2)        ; returns 1
(if 1 2 3)        ; error
(if (= 1 1) 1 2) ; returns 1
(if x 1 0)        ; error if x is not t or nil
(if (= x 2) 3)
(if (= (fn) 0)    ; if / else
  (block
     (fn1 1)
     (fn2))
  (fn3 1 2))
```

class opensdraw.lcad_language.coreFunctions.**Import**

import - Import a module.

Module are searched for in the current working directory first, then in the library folder of the opensdraw project. The modules are assumed to be in files that end with the ".lcad" extension.

Usage:

```
(import mod1)       ; import mod1.lcad
(print mod1:x)      ; print the value of x in the mod1.lcad module.
(mod1:fn 1)         ; call the function fn in the mod1.lcad module.

(import mod1 mod2) ; import mod1.lcad and mod2.lcad.
(def mx mod1:x)     ; use m1 as an alternate name for mod1:x.
(print mx)          ; print the value of x in the mod1.lcad module.

(import mod1 mod2 :local) ; import mod1.lcad and mod2.lcad into the name space of␣
↪current module.
```

class opensdraw.lcad_language.coreFunctions.**Lambda**

lambda - Create an anonymous function.

Usage:

```
(def fn (lambda (x) (+ x 1)))           ; Create a function and assign to symbol␣
↪fn.
(fn 1)                                  ; -> 2
(def myp (lambda () (part "32524" 15))) ; Create function that creates a␣
↪particular part.
```

class opensdraw.lcad_language.coreFunctions.**Len**
len - Return the length of a list.

Usage:

```
(len (list 1 2 3)) ; returns 3
```

class opensdraw.lcad_language.coreFunctions.**List**
list - Create a list.

Usage:

```
(list 1 2 3)  ; returns the list 1,2,3
(list)        ; an empty list
```

class opensdraw.lcad_language.coreFunctions.**Print**
print - Print to the console.

Usage:

```
(print x "=" 10)
```

class opensdraw.lcad_language.coreFunctions.**PyImport**
pyimport - Import a Python module and add any functions in the modules lcad_functions{} dictionary (that are instances of LCadFunction) to the current lexical environment. The module must be on the Python path (which includes the current working directory).

Usage:

```
(pyimport mymodule)  ; Import the Python module mymodule.py.
```

class opensdraw.lcad_language.coreFunctions.**Set**
set - Set the value of an existing symbol.

Usage:

```
(set x 15) - Set the value of x to 15.
(set x 15 y 20) - Set x to 15 and y to 20.
(set x fn) - Set x to be the function fn.
```

class opensdraw.lcad_language.coreFunctions.**While**
while - While loop.

Usage:

```
(def x 1)
(while (< x 10) .. )
```

Part Functions

class opensdraw.lcad_language.partFunctions.**Comment**
comment - Adds comments into the current group.

This different from header() in that these will appear in the body of the file along with parts, etc.. And if group has comments then the step parameter of the part() function will be ignored, so if steps are desired they'll need to be manually created with this function.

This will add a line of text, prepended with "0 ", to the current model. Multiple calls will add multiple lines, in the same order as the calls.

Usage:

```
(comment "BFC INVERTNEXT")
(comment "STEP")
```

class opensdraw.lcad_language.partFunctions.**Group**

group - Creates a group (or sub-file) in the model.

This allows the grouping of parts to create a multi-part output file. Normally you probably want to group parts into a function, but sometimes you might want to generate output that is better formatted for tools that work on mpd files.

Note:

1.Every model will have at least one group, *main*.

2.When created groups always have the identity transformation matrix, not the current transformation matrix.

3.Group names must be unique (and also not overlap with the names of any LDraw part files).

Usage:

```
(group "assembly1"   ; Create a group called "assembly1"
  (part ..))         ; containing a single part.
```

class opensdraw.lcad_language.partFunctions.**Header**

header - Adds header information to the current group.

This will add a line of text, prepended with "0 ", to the current model. Multiple calls will add multiple lines, in the same order as the calls.

Usage:

```
(header "FILE mymoc")
(header "Name: mymoc")
(header "Author: My Name")
```

class opensdraw.lcad_language.partFunctions.**Line**

line - Add a line primitive to the current group.

The arguments are (list x1 yz 1) (list x2 y2 z2).

Parameters

- **x1** – x position of vertex 1.
- **y1** – y position of vertex 1.
- **z1** – z position of vertex 1.
- **x2** – x position of vertex 2.
- **y2** – y position of vertex 2.
- **z2** – z position of vertex 2.

- **color** – (optional) line color, defaults to 16, the "main color".

Usage:

```
(line (list x1 y1 z1) (list x2 y2 z2) color)    ; A line from (x1, y1, z1) to (x2,
↪ y2, z2) with color color.
(line (list x1 y1 z1) (list x2 y2 z2))          ; Same as above, but now color␣
↪will be the "main color", i.e. 16.
```

class opensdraw.lcad_language.partFunctions.**OptionalLine**

optional-line - Add a optional line primitive to the current group.

The arguments are (list x1 yz 1) (list x2 y2 z2) (list x3 y3 z3) (list x4 y4 z4).

Parameters

- **x1** – x position of line vertex 1.
- **y1** – y position of line vertex 1.
- **z1** – z position of line vertex 1.
- **x2** – x position of line vertex 2.
- **y2** – y position of line vertex 2.
- **z2** – z position of line vertex 2.
- **x1** – x position of control vertex 1.
- **y1** – y position of control vertex 1.
- **z1** – z position of control vertex 1.
- **x2** – x position of control vertex 2.
- **y2** – y position of control vertex 2.
- **z2** – z position of control vertex 2.
- **color** – (optional) line color, defaults to 16, the "main color".

Usage:

```
(optional-line (list x1 y1 z1) (list x2 y2 z2) (list x3 y3 z3) (list x4 y4 z4)␣
↪color) ; A optional line with vertices (x1, y1, z1), (x2, y2, z2)
                                                                              ␣
↪    ; and control point vertices (x3, y3, z3), (x4, y4, z4).
(optional-line (list x1 y1 z1) (list x2 y2 z2) (list x3 y3 z3) (list x4 y4 z4))␣
↪    ; Same as above, but now color will be the "main color", i.e. 16.
```

class opensdraw.lcad_language.partFunctions.**Part**

part - Add a part to the current group.

Parameters

- **part_id** – The name of the LDraw .dat file for this part.
- **part_color** – The LDraw name or id of the color.
- **part_step** – (Optional) Which building step to add the part (default = first step).

Usage:

```
(part "32524" 13)           ; Technic Beam 7, LDraw color 13.
(part '32524' "yellow")     ; Technic Beam 7, LDraw color yellow.
(part "32524" "yellow" 10)  ; Technic Beam 7, LDraw color yellow, step 10.
(part "32524" "0x2808080")  ; Technic Beam 7, direct color (gray).
```

class opensdraw.lcad_language.partFunctions.**Quadrilateral**

quadrilateral - Add a quadrilateral primitive to the current group.

The arguments are (list x1 yz 1) (list x2 y2 z2) (list x3 y3 z3) (list x4 y4 z4).

Parameters

- **x1** – x position of vertex 1.
- **y1** – y position of vertex 1.
- **z1** – z position of vertex 1.
- **x2** – x position of vertex 2.
- **y2** – y position of vertex 2.
- **z2** – z position of vertex 2.
- **x3** – x position of vertex 3.
- **y3** – y position of vertex 3.
- **z3** – z position of vertex 3.
- **x4** – x position of vertex 4.
- **y4** – y position of vertex 4.
- **z4** – z position of vertex 4.
- **color** – (optional) fill color, defaults to 16, the "main color".

You should probably also specify a winding order using the header() function.

Example:

```
(header "BFC CERTIFY CCW")
```

Usage:

```
(quadrilateral (list x1 y1 z1) (list x2 y2 z2) (list x3 y3 z3) (list x4 y4 z4)␣
↪color) ; A quadrilateral with vertices (x1, y1, z1), (x2, y2, z2), (x3, y3, z3),
↪ (x4, y4, z4).
(quadrilateral (list x1 y1 z1) (list x2 y2 z2) (list x3 y3 z3) (list x4 y4 z4))  ␣
↪    ; Same as above, but now color will be the "main color", i.e. 16.
```

class opensdraw.lcad_language.partFunctions.**Triangle**

triangle - Add a triangle primitive to the current group.

The arguments are (list x1 yz 1) (list x2 y2 z2) (list x3 y3 z3).

Parameters

- **x1** – x position of vertex 1.
- **y1** – y position of vertex 1.
- **z1** – z position of vertex 1.
- **x2** – x position of vertex 2.

- **y2** – y position of vertex 2.
- **z2** – z position of vertex 2.
- **x3** – x position of vertex 3.
- **y3** – y position of vertex 3.
- **z3** – z position of vertex 3.
- **color** – (optional) fill color, defaults to 16, the "main color".

You should probably also specify a winding order using the header() function.

Example:

```
(header "BFC CERTIFY CCW")
```

Usage:

```
(triangle (list x1 y1 z1) (list x2 y2 z2) (list x3 y3 z3) color)  ; A triangle␣
↪with vertices (x1, y1, z1), (x2, y2, z2), (x3, y3, z3).
(triangle (list x1 y1 z1) (list x2 y2 z2) (list x3 y3 z3)         ; Same as above,
↪ but now color will be the "main color", i.e. 16.
```

Comparison Functions

class opensdraw.lcad_language.comparisonFunctions.**Equal**(*name*)

=

Usage:

```
(= 1 1)     ; t
(= 1 2)     ; nil
(= 2 2 2 2) ; t
(= "a" "a") ; t
```

class opensdraw.lcad_language.comparisonFunctions.**Ne**(*name*)

!=

Usage:

```
(!= 2 1) ; t
```

class opensdraw.lcad_language.comparisonFunctions.**Gt**(*name*)

>

Usage:

```
(> 2 1) ; t
```

class opensdraw.lcad_language.comparisonFunctions.**Ge**(*name*)

>=

Usage:

```
(>= 2 1) ; t
```

class opensdraw.lcad_language.comparisonFunctions.**Lt**(*name*)

<

Usage:

```
(< 2 1) ; nil
```

class opensdraw.lcad_language.comparisonFunctions.**Le**(*name*)

<=

Usage:

```
(<= 2 1) ; nil
```

Logical Functions

class opensdraw.lcad_language.logicFunctions.**And**(*name*)

and - And statement.

Usage:

```
(and (< 1 2) (< 2 3)) ; t
(and (fn x) nil)      ; nil
```

class opensdraw.lcad_language.logicFunctions.**Or**(*name*)

or - Or statement.

Usage:

```
(or (< 1 2) (> 1 3))  ; t
(or (fn x) t)         ; t
(or nil nil)          ; nil
```

class opensdraw.lcad_language.logicFunctions.**Not**(*name*)

not - Not statement.

Usage:

```
(not t)  ; nil
(not ()) ; t
```

Type Functions

class opensdraw.lcad_language.typeFunctions.**IsBoolean**(*name*)

boolean? - Returns t/nil if argument is a boolean.

Usage:

```
(boolean? nil) ; t
(boolean? 1)   ; nil
```

class opensdraw.lcad_language.typeFunctions.**IsMatrix**(*name*)

matrix? - Returns t/nil if argument is a matrix.

Usage:

```
(matrix? (matrix 0 0 0 0 0 0)) ; t
(matrix? (vector 0 0 0))       ; nil
```

class opensdraw.lcad_language.typeFunctions.**IsNumber**(*name*)
number? - Returns t/nil if argument is a number.

Usage:

```
(number? 1)   ; t
(number? "a") ; nil
```

class opensdraw.lcad_language.typeFunctions.**IsString**(*name*)
string? - Returns t/nil if argument is a string.

Usage:

```
(string? 1)   ; nil
(string? "a") ; t
```

class opensdraw.lcad_language.typeFunctions.**IsVector**(*name*)
vector? - Returns t/nil if argument is a vector.

Usage:

```
(vector? (matrix 0 0 0 0 0 0)) ; nil
(vector? (vector 0 0 0))       ; t
```

Geometry Functions

class opensdraw.lcad_language.geometryFunctions.**CrossProduct**
cross-product - Return the cross product of two vectors.

Parameters

- **v1** – The first vector.
- **v2** – The second vector.
- **normalize** – t/nil (Optional) Normalize the result vector, default is t.

Usage:

```
(cross-product (vector 1 0 0) (vector 0 1 0))     ; Returns (vector 0 0 1)
(cross-product (vector 1 0 0) (vector 0 2 0) nil) ; Returns (vector 0 0 2)
```

class opensdraw.lcad_language.geometryFunctions.**DotProduct**
dot-product - Return the dot product of two vectors.

Parameters

- **v1** – The first vector.
- **v2** – The second vector.
- **normalize** – t/nil (Optional) Normalize the result vector, default is t.

Usage:

```
(dot-product (vector 1 0 0) (vector 1 0 0))     ; Returns 1.
(dot-product (vector 1 0 0) (vector 0 1 0))     ; Returns 0.
(dot-product (vector 2 0 0) (vector 1 0 0) nil) ; Returns 2.
```

class opensdraw.lcad_language.geometryFunctions.**Matrix**

matrix - Return a 4 x 4 transform matrix.

The matrix is a numpy array. There are 4 different ways to call this function:

1.With *(list x y z a b c d e f g h i)* as defined here: http://www.ldraw.org/article/218.html#lt1

Parameters

- **x** – translation in x in LDU.
- **y** – translation in y in LDU.
- **z** – translation in z in LDU.
- **a** – m(0,0)
- **b** – m(0,1)
- **c** – m(0,2)
- **d** – m(1,0)
- **e** – m(1,1)
- **f** – m(1,2)
- **g** – m(2,0)
- **h** – m(2,1)
- **i** – m(2,2)

2.With *(list x y z rx ry rz)*

Parameters

- **x** – translation in x in LDU.
- **y** – translation in y in LDU.
- **z** – translation in z in LDU.
- **rx** – rotation around the x-axis in degrees.
- **ry** – rotation around the y-axis in degrees.
- **rz** – rotation around the z-axis in degrees.

3.With *(list p-vec x-vec y-vec z-vec)*

Parameters

- **p-vec** – translation vector in LDU.
- **x-vec** – x vector of the coordinate system.
- **y-vec** – y vector of the coordinate system.
- **z-vec** – z vector of the coordinate system.

The vectors x-vec, y-vec, z-vec should be orthogonal to each other. They will be normalized to length 1.

4.With another matrix, in which case a copy will be made.

Parameters **m** – 4 x 4 transformation matrix.

Usage:

```
(def m1 (matrix (list x y z a b c d e f g h i))) ; All the coefficients.
(def m2 (matrix (list x y z rx ry rz))           ; Translation & rotation values.
(def m3 (matrix (list (list 0 0 10)              ; 4 lists or vectors.
                      (list 1 0 0)
                      (list 0 1 0)
                      (list 0 0 1))))
(def m4 (matrix m1))                             ; m4 is a copy of m1.
```

class opensdraw.lcad_language.geometryFunctions.**Mirror**

mirror - Mirror child elements on a plane through the origin.

The arguments are *(list mx my mz)*, or *(vector mx my mz)*.

Parameters

- **mx** – mirror on x axis.
- **my** – mirror on y axis.
- **mz** – mirror on z axis.

Usage:

```
(mirror (list 1 0 0) ..) ; mirrors on the x axis.
```

class opensdraw.lcad_language.geometryFunctions.**Rotate**

rotate - Rotate child elements.

Add a rotation to the current transformation matrix, rotation is done first around z, then y and then x. Parts added inside a rotate block have this transformation applied to them.

The arguments are *(list ax ay az)*, or *(vector ax ay az)*.

Parameters

- **ax** – Amount to rotate around the x axis in degrees.
- **ay** – Amount to rotate around the y axis in degrees.
- **az** – Amount to rotate around the z axis in degrees.

Usage:

```
(rotate (list 0 0 90) .. )
(rotate (vector 0 0 90) .. )
```

class opensdraw.lcad_language.geometryFunctions.**Scale**

scale - Scale child elements.

Add scale terms to the current transformation matrix. Parts inside a scale block have this transformation applied to them. This is probably not a good idea for standard parts, but it seems to be used with some part primitives.

The arguments are *(list sx sy sz)*, or *(vector sx sy sz)*.

Parameters

- **sx** – Scale in x.
- **sy** – Scale in y.
- **sz** – Scale in z.

Usage:

```
(scale (list 2 1 1) .. )  ; stretch by a factor of two in the x dimension.
```

class opensdraw.lcad_language.geometryFunctions.**Transform**

transform - Transform child elements.

Transform child elements with a 4 x 4 transform matrix. This function is called in exactly the same way as the *matrix()* function.

Usage:

```
(transform (list x y z a b c d e f g h i) ; All the coefficients.
 ..)
(transform (list x y z rx ry rz)          ; Translate to x, y, z and rotate by rx,
↪ ry, rz.
 ..)
(transform m                              ; m is a 4x4 transform matrix.
 ..)
```

class opensdraw.lcad_language.geometryFunctions.**Translate**

translate - Translate child elements.

Add a translation to the current transformation matrix. Parts inside a translate block have this transformation applied to them.

The arguments are *(list dx dy dz)*, or *(vector dx dy dz)*.

Parameters

- **dx** – Displacement in x in LDU.
- **dy** – Displacement in y in LDU.
- **dz** – Displacement in z in LDU.

Usage:

```
(translate (list 0 0 5) .. )
```

class opensdraw.lcad_language.geometryFunctions.**Vector**

vector - Create a 4 element vector.

This is a numpy array. It can be used in place of a list for most of the geometry functions like rotate() and translate(). You can also multiply it with 4 x 4 transformation matrices.

Parameters

- **e1** – The first element in the vector.
- **e2** – The second element in the vector.
- **e3** – The third element in the vector.
- **e4** – (Optional) The fourth element in the vector, defaults to 1.0.

Usage:

```
(def v (vector 0 0 5))  ; Create the vector [0 0 5 1]
(translate v .. )       ; Translate by 5 LDU in z.
(* mat v)               ; Multiply the vector with the 4 x 4 matrix mat.
```

Math Functions

class opensdraw.lcad_language.mathFunctions.**Absolute**(*name*)

abs

Return the absolute value of a number.

Usage:

```
(abs 2)  ; 2
(abs -2) ; 2
```

class opensdraw.lcad_language.mathFunctions.**Plus**(*name*)

+

Add together two or more numbers, vectors or matrices.

Usage:

```
(+ 10 20 y) ; 30 + y
```

class opensdraw.lcad_language.mathFunctions.**Minus**(*name*)

-

Subtract one or more numbers, vectors or matrices from the first number, vector or matrix.

Usage:

```
(- 50 20 y) ; -30 - y
(- 50)      ; -50
```

class opensdraw.lcad_language.mathFunctions.**Multiply**(*name*)

*****

Multiply two or more numbers, vectors or matrices. If the first number is a matrix, then multiplication will be done using matrix multiplication, i.e. (* mat vec) will return a vector and (* mat mat) will return a matrix.

Usage:

```
(* 2 2 y) ; 4 * y
```

class opensdraw.lcad_language.mathFunctions.**Divide**(*name*)

/

Divide the first number, vector or matrix by one or more additional numbers, vectors or matrices. Vectors and matrices are divided pointwise.

Usage:

```
(/ 4 2) ; 2
```

class opensdraw.lcad_language.mathFunctions.**Modulo**(*name*)

%

Return remainder of the first number divided by the second number.

Usage:

```
(% 10 2) ; 0
```

All the functions in the python math library are also available:

Usage:

```
(cos x)
(sin x)
(degrees (atan2 2 3))
...
```

Random Number Functions

class opensdraw.lcad_language.randomNumberFunctions.**RandSeed**(*name*)
rand-seed - Initialize the random number generator.

Usage:

```
(rand-seed 10)
```

class opensdraw.lcad_language.randomNumberFunctions.**RandChoice**(*name*)
rand-choice - Return a random element from a list.

Usage:

```
(rand-choice (list 1 2 3)) ; return 1,2 or 3.
(rand-choice (list a b)) ; return a or b
```

class opensdraw.lcad_language.randomNumberFunctions.**RandGauss**(*name*)
rand-gauss - Return a gaussian distributed random number. This can be called with either 0 or 2 arguments.

Usage:

```
(rand-gauss)      ; mean = 0, standard deviation = 1.0
(rand-gauss 1 10) ; mean = 1, standard deviation = 10.0
```

class opensdraw.lcad_language.randomNumberFunctions.**RandInteger**(*name*)
rand-integer - Return a random integer

Usage:

```
(rand-integer 0 100) ; random integer between 0 and 100.
(rand-integer 2 30)  ; random integer between 2 and 30.
```

class opensdraw.lcad_language.randomNumberFunctions.**RandUniform**(*name*)
rand-uniform - Return a uniformly distributed random number. This can be called with either 0 or 2 arguments.

Usage:

```
(rand-uniform)      ; distributed on 0 - 1.
(rand-uniform 1 10) ; distributed on 1 - 10.
```

Libraries

Core Functions

class opensdraw.lcad_language.belt.**LCadBelt**
belt - Creates a belt function.

This function creates and returns a function that parametrizes a belt making it easier to add custom belts / chains to a MOC. Unlike the chain function this allows for (almost) arbitrary locations and orientations of the pulleys / sprockets. All units are in LDU.

Each pulley / sprocket is specified by a 4 member list consisting of *(position orientation radius winding-direction)*.

Parameters

- **position** – A 3 element list specifying the location of the pulley / sprocket.
- **orientation** – A 3 element list specifying the vector perpendicular to the plane of the pulley / sprocket.
- **radius** – The radius of the pulley / sprocket in LDU.
- **winding-direction** – Which way the belt goes around the pulley / sprocket (1 = counter-clockwise, -1 = clockwise).

The belt goes around the pulleys / sprockets in the order in which they are specified, and when *:continuous* is **t** returns from the last pulley / sprocket to the first to close the loop.

When you call the created belt function you will get a 4 x 4 transform matrix which will translate to the requested position on the belt and orient to a coordinate system where the z-axis is pointing along the belt, the y-axis is in the plane of the belt and the x-axis is perpendicular to the plane of the belt, pointing in the direction of the pulley / sprocket perpendicular vector.

If you call the created belt function with the argument **t** it will return the length of the belt.

Additionally belt has the keyword argument:

```
:continuous t/nil  ; The default is t, distances will be interpreted modulo the␣
↪belt length, and
                   ; the belt will go from that last pulley back to the first␣
↪pulley. If nil
                   ; then negative distances will wrap around the first pulley␣
↪and positive
                   ; distances will wrap around the last pulley.
```

Usage:

```
(def a-belt (belt (list (list (list 0 0 0) ; Create a belt with two pulleys.
                              (list 0 0 1) ; Pulley one is at 0,0,0 and is in the
                              1.0 1)       ; x-y plane with radius 1 and counter-
↪clockwise
                        (list (list 4 0 0) ; winding direction.
                              (list 0 0 1) ; Pulley two is at 4,0,0 with radius 1.
↪5.
                              1.5 1))))

(def m (a-belt 1))                         ; m is a 4 x 4 transform matrix.
(a-belt t)                                 ; Returns the length of the belt.
```

class opensdraw.lcad_language.chain.**LCadChain**

chain - Creates a chain function.

This function creates and returns a function that parametrizes a chain, making it easier to add chains, tracks, etc. to a MOC. This is a simplified version of the belt function where everything is confined to the XY plane. If you just want a simple chain this is probably the function that you want to use. All units are LDU.

A chain must have at least two sprockets. Each sprocket is specified by a 4 member list consisting of *(x y radius winding-direction)*.

Parameters

- **x** – The x location of the sprocket.
- **y** – The y location of the sprocket.
- **radius** – The radius of the sprocket.
- **winding-direction** – Which way the belt goes around the sprocket (1 = counter-clockwise, -1 = clockwise).

The chain goes around the sprockets in the order in which they are specified, and when *:continuous* is **t** returns from the last sprocket to the first sprocket to close the loop.

When you call the created chain function you will get a 4 x 4 transform matrix which will translate to the requested position on the chain and orient to a coordinate system where the z-axis is pointing along the chain, the y-axis is in the plane of the chain and the x-axis is perpendicular to the plane of the chain.

If you call the created chain function with the argument **t** it will return the length of the chain.

Additionally chain has the keyword argument:

```
:continuous t/nil  ; The default is t, distances will be interpreted modulo the␣
↪chain length, and
                   ; the chain will go from that last sprocket back to the first␣
↪sprocket. If nil
                   ; then negative distances will wrap around the first sprocket␣
↪and positive
                   ; distances will wrap around the last sprocket.
```

Usage:

```
(def a-chain (chain (list (list -4 0 1 1)     ; Create a chain with two sprockets,␣
↪the 1st at (-4,0) and
                          (list 4 0 1 1))))   ; the second at (4,0). Both␣
↪sprockets have radius 1 and a
                                              ; counter-clockwise winding␣
↪direction.
(def m (a-chain 1))                           ; m is a 4 x 4 transform matrix.
(a-chain t)                                   ; Returns the length of the chain.
```

class opensdraw.lcad_language.curve.**LCadCurve**

curve - Creates a curve function.

This function creates and returns a function that parametrizes a curve, specifically a cubic spline. All units are LDU.

Control points are specified by a 2 member list consisting of *(position derivative)*.

Parameters

- **position** – A 3 element list specifying the location of the control point.
- **derivative** – A 3 element list specifying the derivative (tangent) of the curve as it passes through the control point.

The first control point is specified by a 3 member list consisting of *(position derivative perpendicular)*

Parameters

- **position** – A 3 element list specifying the location of the control point.
- **derivative** – A 3 element list specifying the derivative (tangent) of the curve as it passes through the control point.

- **perpendicular** – A 3 element list specifying an approximately perpendicular vector to the derivative.

A curve must have at least 2 control points.

When you call the created curve function you will get a 4 x 4 transform matrix which will translate to the requested position on the curve and orient to a coordinate system where z is along the curve and x is perpendicular to the coordinate system as defined by the perpendicular vector provided for the 1st control point.

If you call the created curve function with the argument **t** it will return the length of the curve.

Additionally curve has several keyword arguments:

```
:auto-scale   t/nil        ; The default is t, automatically scale the derivative␣
↪to
                           ; minimize the maximum curvature of the curve.
:extrapolate  t/nil        ; The default is t, distances outside of the curve␣
↪will be
                           ; linearly extrapolated from the end of the curve. If␣
↪nil
                           ; then the distance will be modulo the curve length.
:scale        float > 0.0  ; The multiplier for auto-scale mode, defaults to 1.␣
↪This
                           ; sets the boundaries on the auto-scale optimal
                           ; derivative search range. If you change this you␣
↪probably
                           ; want a number greater than 1.0, which is the default␣
↪value.
:twist        angle        ; Additional twist along the curve, defaults to 0.
```

Unfortunately auto scaling is a bit slow and sometimes fails, so more work is needed..

Usage:

```
(def my-curve (curve (list (list (list 0 0 0) (list 1 1 0) (list 0 0 1)) ; Create␣
↪a curve going through (0,0,0), (5,0,0)
                           (list (list 5 0 0) (list 1 0 0))))            ; With␣
↪derivative (1,1,0) and (1,0,0). Since we did
                                                                         ; not␣
↪specify :auto-scale nil, the derivative will
                                                                         ; be␣
↪scaled to create a hopefully pleasing curve.

(def m (my-curve 1))                                                     ; m is a␣
↪4 x 4 transform matrix for the
                                                                         ; curve␣
↪at distance 1 along the curve.
(my-curve t)                                                             ;␣
↪Returns the length of the curve.
```

class opensdraw.lcad_language.pulleySystem.**LCadPulleySystem**

pulley-system - Creates a pulley-system function.

This function creates and returns a function that parametrizes a pulley-system making it easier to add pulleys and string to a MOC. The function is very similar to belt(), except that the first element specifies a drum on which the string is wound and the last specifies either the end point of the string, or the direction that string goes after the last sprocket. All units are LDU.

A pulley-system must have at least two elements (including the initial drum and the final end point). Like belt, the pulleys are specified by a 4 element list consisting of *(position orientation radius winding-direction)*.

Parameters

- **position** – A 3 element list specifying the location of the pulley.
- **orientation** – A 3 element list specifying the vector perpendicular to the plane of the pulley.
- **radius** – The radius of the pulley in LDU.
- **winding-direction** – Which way the string goes around the pulley (1 = counter-clockwise, -1 = clockwise).

The string goes around the pulleys in the order in which they are specified.

The initial drum is specified by the 7 element list *(position orientation radius winding-direction drum-width string-thickness string-length)*.

Parameters

- **position** – A 3 element list specifying the location of the drum.
- **orientation** – A 3 element list specifying the vector perpendicular to the plane of the drum.
- **radius** – The radius of the drum in LDU.
- **winding-direction** – Which way the string goes around the drum (1 = counter-clockwise, -1 = clockwise).
- **drum-width** – The width of the drum in LDU.
- **string-thickness** – The diameter of the string.
- **string-length** – The amount of string wound around the drum, **not** the total string length.

The final end point is specified by a 2 element position list, *(position/direction type)*.

Parameters

- **position/direction** – A 3 element list specifying the end point of the pulley system, or a tangent vector.
- **type** – This is either the string "point" or "tangent".

When you call the created pulley-system function you will get a 4 x 4 transform matrix which will translate to the requested position on the string and orient to a coordinate system where the z-axis is pointing along the string, the y-axis is in the plane of the string and the x-axis is perpendicular to the plane of the string, pointing in the direction of the pulley perpendicular vector.

If you call the created pulley-system function with the argument **t** it will return the length of the string.

Usage:

```
; Drum at 0,0,0 rotating counter-clockwise in the x-y plane with radius
; 5, width 50 and 50 LDU of 1 LDU diameter string. The string then goes
; around a pulley 20,0,1 rotating counter-clockwise in the x-y plane
; with radius 5. After passing the pulley the string continues in the
; -x direction.
(def ps1 (pulley-system (list (list (list 0 0 0) (list 0 0 1) 5.0 1 5.0 1 50)
                              (list (list 20 0 1) (list 0 0 1) 5.0 1)
                              (list (list -1 0 0) "tangent"))))

; Drum at 0,0,0 rotating counter-clockwise in the x-y plane with radius
; 5, width 50 and 50 LDU of 1 LDU diameter string. The string then passes
```

```
; through the point 20,0,0.
(def ps2 (pulley-system (list (list (list 0 0 0) (list 0 0 1) 5.0 1 5.0 1 50)
                              (list (list 20 0 0) "point"))))

(def m (ps1 1))    ; m is a 4 x 4 transform matrix.
(ps1 t)            ; Returns the length of pulley system ps1 including the
                   ;  string on the drum.
(ps2 t)            ; Returns the length of pulley system ps2 including the
                   ;  string on the drum.
```

class opensdraw.lcad_language.spring.**LCadSpring**

spring - Creates a spring function.

This function creates and returns a function that parametrizes a spring making it easier to add custom springs to a MOC. All units are LDU. The spring is oriented along the z-axis.

Parameters

- **length** – The length of the spring.
- **diameter** – The diameter of the spring.
- **gauge** – The thickness of the spring wire.
- **turns** – The number of turns in the center part of the spring.
- **end-turns** – (optional) The number of turns at the end of the spring, default is 2.

Usage:

```
(import flexible-rod :locale)             ; Import the flexible-rod module.
(def a-spring (spring 40 10 1 10))        ; Create a length 40 spring with␣
↪diameter 10, gauge 1 and 10 turns.
(flexible-rod a-spring 0 (a-spring t) 1) ; Draw a rod a long the path of a-spring␣
↪with diameter 1.

(def a-spring (spring 40 10 1 10 1))      ; Same as above, but with only 1 turn at␣
↪the end.
```

Extra Functions (Python)

These are functions that you will need to import in order to use. They can all be found in the *opensdraw/library* directory.

Knots

This module makes it easier to add a knot to your MOC.

```
(pyimport opensdraw.library.knots)
```

class opensdraw.library.knots.**SheetBendKnot**

sheet-bend-knot - Creates a sheet bend knot function.

This creates and returns a function that parametrizes a sheet bend knot. All units are LDU.

When you call the created knot function you will get a 4 x 4 transform matrix which will translate to the requested position on the knot and orient to a coordinate system where the z-axis is pointing along the knot, the x-axis is in the plane of the knot and the y-axis is perpendicular to the plane of the knot.

Parameters

- **diameter** – The diameter of the string.
- **loop_size** – The diameter of the loop.

Usage:

```
(def sbk (sheet-bend-knot 3 10)) ; A knot with 3 LDU diameter string and loop␣
↪diameter of 10.
(def p1 (sbk 1))                 ; p1 is the list (x y z rx ry rz) which defines␣
↪the
                                 ; knot at distance 1 along the knot.
(sbk t)                          ; Returns the length of the knot.
```

Overlay

This module make LDraw compatible images that can be overlaid on a MOC for scaling pictures. It can also be used for creating 2D (i.e. flat) stickers.

```
(pyimport opensdraw.library.overlay)
```

class opensdraw.library.overlay.**Overlay**

overlay - Create a semi-transparent LDraw compatible image from a normal image.

This function is useful for overlaying images for scaling purposes. It could also be used to create 2D stickers. The image will be in the XY plane with the upper left corner of the image at 0,0.

The function will return the highest color index that is used.

Parameters

- **image** – The name of the image file.
- **scale** – The conversion factor (LDU / pixel).
- **index** – The starting color index. LDraw uses color indices as high as 511, so 600+ is probably a good idea.
- **transparency** – Optional (0-255, lower is more transparent), default is 64.

Usage:

```
(overlay "blueprint.png" 2.0 600)
```

Parts Strings

This module lets you add parts using space delimited strings, instead of using functions like *tb()* and *sb()* from *locate.lcad*

```
(pyimport opensdraw.library.partsString)
```

class opensdraw.library.partsString.**PartsFile**

parts-file - Specify parts in a text file.

This lets you load parts from a text file that is formatted in the same fashion as for the *parts-string()* function.

Parameters **file** – A string containing the name of the file to load.

Usage:

```
(parts-file "parts_file.txt") ; Load parts from parts_file.txt
```

class opensdraw.library.partsString.**PartsString**

parts-string - Specify parts using a return delimited string.

This lets you specify parts using a return delimited string, instead of having to use a function like *tb()* or *sb()* from *locate.lcad*. When you have lots of parts with a relatively simple geometry this might be faster and easier. If the first line of the string contains the word "technic" then technic brick spacing will be used instead of standard brick spacing. Any line that contains 8 or 9 elements is assumed to specify a part as *(x, y, z, x rotation, y rotation, z rotation, part, color, {optional} step)*. Other lines are ignored, including all lines that start with ";".

Parameters **string** – The string of part locations and types.

Usage:

```
; 3 2x1 bricks using standard brick units.
(parts-string "0 0 0 -90 0 0 3004 Red
               0 0 1 -90 0 0 3004 Green
               0 0 2 -90 0 0 3004 Blue")

; 3 technic beam 2 using standard technic units.
(parts-string "technic
               0 2 0 90 0 0 43857 4
               0 2 1 90 0 0 43857 2
               0 2 2 90 0 0 43857 1")
```

Shapes

This module makes it easier (and faster) to create simple shapes from LDraw primitives.

```
(pyimport opensdraw.library.shapes)
```

class opensdraw.library.shapes.**Axle**

axle - Draw an axle using LDraw primitives.

Parameters

- **curve** – The curve that the axle should follow.
- **start** – The starting point on the curve.
- **stop** – The stopping point on the curve.
- **orientation** – (optional) Angle in degrees in the XY plane.

The axle will have the color 16.

Usage:

```
(axle curve 0 10) ; Draw an axle along curve from 0 to 10 (LDU).
```

class opensdraw.library.shapes.**FlatCable**

flat-cable - Draw a flat cable (i.e. EV3 or NXT style) using LDraw primitives.

Parameters

- **curve** – The curve that the cable should follow.
- **start** – The starting point on the curve.
- **stop** – The stopping point on the curve.

- **width** – The width of the cable.
- **radius** – The edge radius of the cable.
- **orientation** – (optional) Angle in degrees in the XY plane, default is 0 (the long axis of the cable is along the X axis).

The flat cable will have the color 16.

Usage:

```
(flat-cable curve 0 10 4 1) ; Draw a 4 LDU wide flat cable with 1 LDU radius␣
↪edges.
```

class opensdraw.library.shapes.**RibbonCable**

ribbon-cable - Draw a ribbon cable using LDraw primitives.

Parameters

- **curve** – The curve that the cable should follow.
- **start** – The starting point on the curve.
- **stop** – The stopping point on the curve.
- **strands** – The number of strands in the cable.
- **radius** – The radius of a single strand in the cable.
- **orientation** – (optional) Angle in degrees in the XY plane, default is 0 (the long axis of the cable is along the X axis).

The ribbon cable will have the color 16.

Usage:

```
(ribbon-cable curve 0 10 4 1) ; Draw a 4 stranded ribbon cable with each strand
                              ; having a radius of 1 LDU.
```

class opensdraw.library.shapes.**Ring**

ring - Draw a ring using LDraw primitives.

Parameters

- **m1** – Transform matrix for the first edge of the ring.
- **v1** – Vector for the first edge of the ring.
- **m2** – Transform matrix for the second edge of the ring.
- **v2** – Vector for the second edge of the ring.
- **ccw** – Counterclockwise winding (t/nil).

The ring will have the color 16.

Usage:

```
(ring m1 v1 m2 v2 t)  ; Draw a ring with edge 1 defined by m1, v1
                      ; and edge 2 defined by m2, v2, with ccw winding.
```

class opensdraw.library.shapes.**Rod**

rod - Draw a rod using LDraw primitives.

Parameters

- **curve** – The curve that the rod should follow.

- **start** – The starting point on the curve.
- **stop** – The stopping point on the curve.
- **radius** – The radius of the rod.

The rod will have the color 16.

Usage:

```
(rod curve 0 10 2) ; Draw a 2 LDU diameter rod from 0 to 10 along curve.
```

class opensdraw.library.shapes.**Tube**

tube - Draw a tube using LDraw primitives.

Parameters

- **curve** – The curve that the tube should follow.
- **start** – The starting point on the curve.
- **stop** – The stopping point on the curve.
- **inner_radius** – The inner radius of the tube.
- **outer_radius** – The outer radius of the tube.

The tube will have the color 16.

Usage:

```
(tube curve 0 10 2 3) ; Draw a 2 LDU inner diameter, 3 LDU outer diameter
                      ; tube from 0 to 10 along curve.
```

Extra Functions (LCad)

These are functions that you will need to import in order to use. They can all be found in the *opensdraw/library* directory.

- cables.lcad - Rendering cables.
- flexible-axle.lcad - Rendering flexible axle.
- flexible-hose.lcad - Rendering flexible hoses.
- flexible-rod.lcad - Rendering flexible rods.
- ldu.lcad - Converting from bricks to LDU.
- locate.lcad - Functions to make placing parts easier.
- ribbed-hose.lcad - Rendering ribbed hoses.
- triangles.lcad - Functions for calculating sides and angles of triangles.

TODO: Figure out how to import reST from non-Python code.

Python Module Index

Index

L

M

N

O

P

Q

R