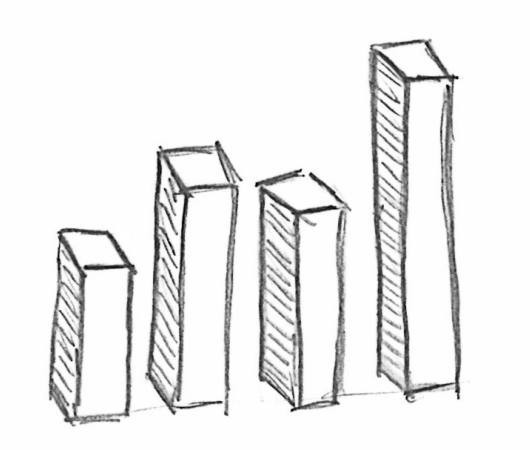
TANGIBLE

DANILO BARGEN



A Python library to convert data into tangible 3D models.
Student Research Project Thesis, Fall 2013.

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

SUPERVISORS:
Prof. Dr. Josef M. Joller

UNIVERSITY:
HSR University of Applied Science Rapperswil

DEPARTMENT:
Department of Computer Science

INSTITUTE:
ITA Institute for Internet Technologies and Applications

LOCATION:
Rapperswil

TIME FRAME:
Fall 2013

ABSTRACT

In the past, making data tangible was a complicated, manual process. Digital 3D representations of complex data have been around for quite a while, but they were always confined to the digital world. Mostly because it was impractical to convert a digital model to a physical representation.

With the advent of cheap, affordable 3D printers, this changed. It is now easy to convert a purely digital model to a tangible, physical object. The missing piece in the process of making data tangible is the conversion of data to a digital 3D model.

This thesis wants to solve that problem by providing an easy to use software library with "batteries included" that can convert arbitrary numeric data to 3D models. The library – named *Tangible* – is written in Python and provides a set of predefined but customizable shapes, a few tools to preprocess data and a backend implementation for OpenSCAD, an open source programmatic CAD software.

Tangible is implemented as a cross-compiler with a simple abstract syntax tree (AST), a set of predefined shapes that build on top of the AST and an interface that allows the creation of different code generation backends.

The library is ready to use, well tested and thoroughly documented. It has been released under an open source license and is available online at `https://github.com/dbrgn/tangible`.

KEYWORDS:
3D Printing, CAD, Cross Compilers, Data Analysis, Data Visualization, OpenSCAD, Python, Software Libraries, Statistics, Tangible

ACKNOWLEDGEMENTS

First of all I would like to thank my supervisor Prof. Dr. Josef M. Joller for the support and the freedom he gave me during the planning and implementation of this project.

Second I would like to thank Tobias Zehnder, who brought up the idea of tangible printable statistics at an afterwork beer-and-burger meeting.

Further thanks go to Prof. Oliver Augenstein, who helped me with questions related to curve interpolation using splines.

And last but not least, I would like to thank Prof. Dr.-Ing. André Miede for providing the great *classicthesis* LaTeX-Template which I used for this thesis.

CONTENTS

LIST OF FIGURES

ACRONYMS

ABS	Acrylonitrile Butadiene Styrene, a "Lego-like" filament used by FDM 3D printers
AST	Abstract Syntax Tree, a logical representation of a computer program used by compilers
CAD	Computer-aided Design, the use of computer systems to assist in the creation, modification, analysis, or optimization of a design
CSV	Comma-Separated Values, a file format to store tabular data
FDM	Fused Deposition Modeling, a 3D printing technology
JSON	JavaScript Object Notation, a lightweight data-interchange format
PEP	Python Enhancement Proposal
PLA	Polylactic Acid, a biodegradeable filament used by FDM 3D printers
STL	STereoLithography, a file format for 3D models widely used in the field of 3D printing
TDD	Test Driven Development, a software development methodology

Part I

INTRODUCTION

1 MOTIVATION

Data visualization is definitely nothing new. Neither is software to do statistical analysis or 3D model generation. And 3D printers have been around since 1984. But what happens if all these things are combined? By doing that, data visualization is taken to a physical level.

1.1 DATA VISUALIZATION

> *The main goal of data visualization is its ability to **visualize data**, communicating information clearly and effectively.*
> — Vitaly Friedman [7]

Data visualization tries to make raw data more easily accessible. Changes in datapoints over time should be visible, relations between different datasets should become apparent, and at the same time the visualization should be easy to understand and pleasant to look at.

The traditional means to visualize data were mostly two dimensional: Maps visualize geographical and topological relations between objects and landmarks, charts show a dataset in an easy to understand graphical way and infographics present complex information about a specific topic quickly and clearly.

With the advent of computers, data visualization became interactive. Data could be visualized in two- and three dimensional ways and a user could interact with the data and learn more about it.

But digital 3D visualizations are still only two dimensional projections of three dimensional objects. Data and its visualization can be taken to a new level by making the visualizations tangible.

In the past, creating physical objects to convey information was not something very common. It was mostly done by artists as a creative way to display information in the form of a sculpture or another type of object [4][10]. But it was a manual, slow and tedious process.

1.2 THE RISE OF AFFORDABLE 3D PRINTING

Digital 3D representations of complex data have been around for quite a while [8], but they were always confined to the digital world. Mostly because it was impractical to convert a digital model to a physical representation.

Industrial 3D printing and CNC milling have been available for about 3 decades, but just until recently these machines were pro-

hibitively expensive for regular people that just wanted to visualize data. The only alternative was manual work.

The patent "US5121329: Apparatus and method for creating three-dimensional objects" was granted to S. Scott Crump in 1992 and expired in 2009.

During the last few years this changed. In 2009, US patent 5121329 [3] expired, and with that prices for consumer-ready 3D printers plummeted.

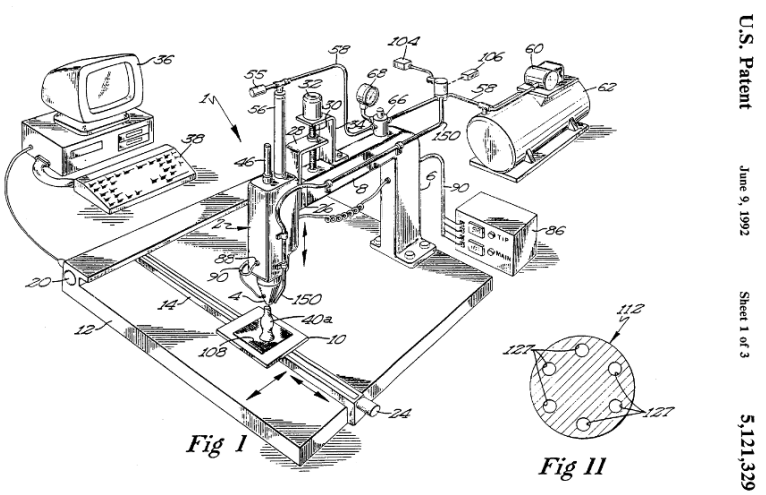


Figure 1: US Patent 5121329

The RepRap project works on creating general-purpose self-replicating machines capable of printing plastic objects, and making them freely available for the benefit of everyone.

The surge of new cheap 3D printers – as an affordable alternative to the large, expensive industrial-grade machines – was largely due to an increasing number of enthusiasts from the *Hacker-* and *Maker-*communities that worked on projects like the RepRap[1], a very successful and influential project.

As Erik De Bruijn, founder of the successful Ultimaker[2] company, discusses in his Master's thesis *"On the viability of the Open Source Development model for the design of physical objects – Lessons learned from the RepRap project"* [2], the Open Source development model has proven to be well suited in the field of physical fabrication.

At the same time, while many of these projects were started by communities as non commercial Open Source[3] and Open Hardware[4] projects, crowdfunding platforms like Kickstarter[5] and Indiegogo[6] made raising capital for new 3D printers quick and easy, something that would not have been possible 10 years ago. This resulted in a flood of readily assembled, affordable 3D printers even for people lacking the skills to build their own machine from a list of parts.

1 http://reprap.org/wiki/RepRap
2 https://www.ultimaker.com/
3 http://opensource.org/
4 http://www.oshwa.org/
5 http://www.kickstarter.com/
6 http://www.indiegogo.com/

1.3 FROM DATA TO TANGIBLE OBJECTS

With the rapid and constantly accelerating developments in the field of 3D printing, visualizing data as real, tangible objects has become feasible. There are still some hurdles though. The majority of people – even those owning a 3D printer – have no experience with 3D modeling software. Most freely available software tools to create 3D models – like OpenSCAD[7] or Blender[8] – require a steep learning curve and demand a nontrivial amount of learning time to create the desired models. And platforms like Thingiverse[9] or YouMagine[10] provide so many freely available models that creating own objects is often not even a necessity.

Another aspect is that these general-purpose modeling tools are not optimized for data visualization. Each type of visualization has to be created manually, and when the data changes it's a lot of manual work to update the model.

Tangible was created to fill that void. It makes creation of customized, printable data visualizations as 3D models easy, while at the same time retaining a lot of flexibility for customization.

7 http://www.openscad.org/
8 http://www.blender.org/
9 http://www.thingiverse.com/
10 https://www.youmagine.com/

2 GOALS

The goals of this thesis can be summarized as follows:

2.1 FEATURES

- The result of the thesis is a Python library to visualize data as printable 3D objects. It should handle single- and multi-dimensional data.
- The library should provide a set of basic predefined shapes.
- It should be possible to create custom shapes using the provided primitives.
- The input and output should be decoupled. The library should act as a cross compiler. It should be possible to generate code for different backends.
- During the time of this thesis, the main targeted backend is OpenSCAD[1].
- The library should run on Python 2.7.

2.2 USABILITY

- The library should be pythonic[2] and easy to use.
- Comprehensive documentation should be available.

2.3 QUALITY

- The library should be well tested (at least 80% test coverage).
- Tests should run automatically every time code is pushed to the repository.
- Change in test coverage should be measured each time the tests are run.

1 http://www.openscad.org/
2 http://stackoverflow.com/q/58968

Part II

THE LIBRARY

3 FEATURES & USAGE

3.1 INTRODUCTION

Tangible is a Python library to convert data into tangible 3D models. It generates code for different backends. It is inspired by projects like OpenSCAD and d3.js[1].

The Python programming language has proven to be an easy, accessible and at the same time very powerful language for data analysis and many other tasks. Due to its readable syntax, it's very easy to get started with it, even for people without any prior programming knowledge. *Tangible* tries to built upon this foundation, by providing an easy to use software library that has "batteries included". It should be easy to process a dataset, normalize the values and generate the desired visualization.

The difference from Projects like SolidPython[2] is that *Tangible* is a modular system with an intermediate representation of objects that is capable of generating code for different backends, it's not tied to a single representation. Additionally, its main focus is not general CAD, but printable 3D visualization of data.

Besides the support for model generation and different backends, *Tangible* also provides utilities to preprocess data, e.g. for normalization of data or for grouping and aggregation of data.

The library runs on Python 2.6 and 2.7. Support for Python 3.3+ is planned.

3.2 USAGE

Tangible was designed to be very straightforward to use. Common data visualizations should be possible with just a few lines of code.

Visualizing data with *Tangible* consists of three steps: Preprocessing the data, creating a shape instance and finally rendering the code using the desired backend.

3.2.1 *Preprocessing Data*

Many times the data is not yet in the right form for visualization. Let's say that the user has air temperature measurements for every hour during an entire day. The temperature range is between 8°C during the night and 22°C during the day.

1 http://d3js.org/

2 https://github.com/SolidCode/SolidPython

```
>>> temperatures = [
>>>     10, 9, 8, 8, 9, 8, 9, 10, 12, 15, 17, 19,
>>>     20, 22, 22, 21, 20, 17, 14, 12, 11, 10, 9, 10
>>> ]
```

To visualize the data, the user wants to create a round tower where the radius of a slice corresponds to a temperature measurement. But the temperatures are not well suited to be used directly as millimeter values. Therefore the user wants to linearly transform the range 8–22 (°C) to the range 10–40 (mm).

Tangible provides helper functions for this called *scales*. First a linear scale needs to be constructed:

```
>>> from tangible import scales
>>> scale = scales.linear(domain=[8, 22], codomain=[10, 40])
```

The returned object is the actual scaling function. It can be used directly:

```
>>> scale(8)
10.0
>>> scale(15)
25.0
>>> scale(22)
40.0
```

...or it can be used in combination with Python's `map()` function:

```
>>> radii = map(scale, temperatures)
>>> radii
[14.285714285714285, 12.142857142857142, 10.0, 10.0, ...]
```

Now the data is ready to be visualized. There are also several other functions to preprocess data, for example to group or aggregate datapoints. For more information, take a look at section 4.5 Utils.

3.2.2 *Creating a Shape Instance*

Tangible provides many predefined shapes that can be used directly. Currently there are three types of shapes: Vertical shapes, bar shapes and pie shapes.

For the temperature tower, the user needs the `CircleTower1D` shape from the `tangible.shapes.vertical` module. The class requires two arguments: The data list as well as the height of each layer.

```
>>> from tangible.shapes.vertical import CircleTower1D
>>> tower = CircleTower1D(data=radii, layer_height=2)
```

An overview over all shape classes can be found in section 4.4.3.

3.2.3 *Rendering the Code*

Now the shape is ready to be rendered. First, choose the desired backend from the `tangible.backends` package. At the time of this writing, the only available backend is the OpenSCAD backend.

```
>>> from tangible.backends.openscad import OpenScadBackend
```

Next, render the shape using this backend. For convenience, we write the resulting code directly into a file.

```
>>> with open('tower.scad', 'w') as f:
...     code = tower.render(backend=openscad.OpenScadBackend)
...     f.write(code)
```

The OpenSCAD code can now be rendered on the command line (or alternatively from the GUI tool) into an image for previewing or into an STL file for printing:

```
$ openscad -o tower.png --render --imgsize=512,512 tower.scad
CGAL Cache insert: cylinder($fn=0,$fa=12,$fs=2,h=5,r1=14.28)
CGAL Cache insert: cylinder($fn=0,$fa=12,$fs=2,h=5,r1=12.14)
...
$ openscad -o tower.stl --render tower.scad
CGAL Cache insert: cylinder($fn=0,$fa=12,$fs=2,h=5,r1=14.28)
CGAL Cache insert: cylinder($fn=0,$fa=12,$fs=2,h=5,r1=12.14)
...
```

The result:

Figure 2: 3D visualization of a temperature range

A few more usage examples are available in the *Examples* chapter on page 25.

3.3 FUTURE DEVELOPMENTS

There are many areas where *Tangible* could still be improved, for example by adding more backends and shapes, by improving Python version support or by adding more utils and helper functions.

3.3.1 *More Backends*

OpenSCAD was chosen as the initial backend because of its popularity. Moreover, it's freely available and open source. Due to its programmatic nature, it's easy to implement as a backend and it also allows the resulting code to be modified directly to make some final adjustments before printing.

Another possible backend would have been ImplicitCAD[3]. It's a project strongly inspired by OpenSCAD but with additional features on top of it. ImplicitCAD is is written in Haskell and supports both an OpenSCAD compatible "legacy" syntax as well as the traditional Haskell notation.

Both of these examples are programmatic CAD tools. A step further would be to support the widely used STL format directly as a backend. But implementing direct STL generation would have gone beyond the scope of this thesis and was therefore not attempted.

3.3.2 *More Shapes*

Right now *Tangible* provides three base shape types with different variants. But the shape library could be further extended, in order to provide even more visualizations that can be used by users without any additional modeling efforts.

3.3.3 *Interpolation / Smoothing*

Tangible provides no tools for interpolation / smoothing of surfaces. This means that shapes with a lot of datapoints may look very ragged and printing them may cause problems because of the overhanging surfaces.

Although libraries like Numpy[4] and SciPy[5] provide interpolation functionality, this is something that *Tangible* should provide out of the box. A possible way of implementing curve smoothing would be by using spline interpolation. This feature is planned for a future release.

3 http://www.implicitcad.org/
4 http://www.numpy.org/
5 http://www.scipy.org/

3.3.4 *Data Preprocessing Tools*

The current selection of data preprocessing tools is already very useful, but still doesn't cover many use cases. Therefore these utils should be expanded, for example by adding logarithmic and exponential scales and by adding more grouping functions. Furthermore the scales could be improved to allow changes in the domain / codomain as well as rescaling of the data at any point in time.

3.3.5 *Python 3 Support*

Right now *Tangible* is written for Python 2.6/2.7. But it would be quite easy to add support for Python 3.3+, especially because the extensive use of future imports (see section 6.2). Python 3 support is already planned and will be implemented soon.

4 ARCHITECTURE

4.1 OVERVIEW

Tangible is implemented as a single Python package, without any external dependencies.

The architecture of *Tangible* can be categorized into four different components: The abstract syntax tree (4.2), code generation backends (4.3), shapes (4.4) and utils (4.5).

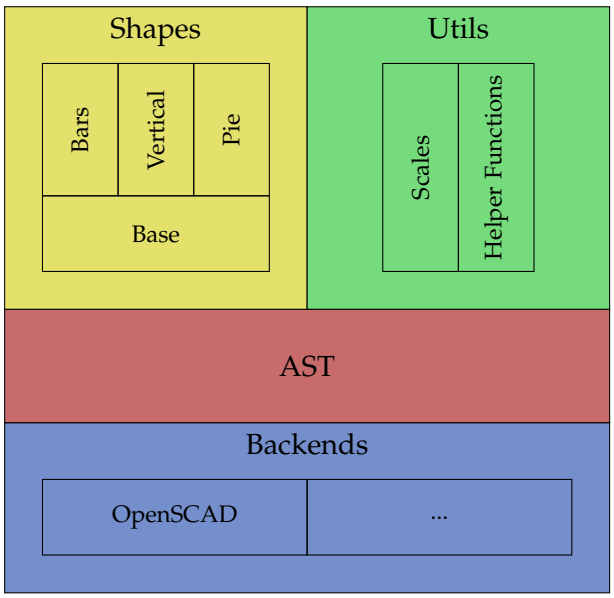


Figure 3: Architecture Diagram

4.2 AST

The `ast.py` module provides the objects for the abstract syntax tree (AST) for *Tangible*. All AST objects extend a single base class called `AST`. This base class is responsible for three things:

- It provides a single base type that can be used for type checking, e.g. `isinstance(mysubclass, ast.AST)`.
- It overrides the `__eq__` and `__ne__` methods, with the result that all subclasses are compared by value (using `self.__dict__`), and not by identity.

- It provides a `__repr__` implementation that displays both the class name as well as the object memory address, which simplifies debugging.

The module contains the following classes:

4.2.1 *Base Class*

- `AST`: The base shape for all AST elements, as described above.

4.2.2 *2D Shapes*

- `Circle`: A circle shape with a specified radius.
- `CircleSector`: A circle sector shape (pizza slice) with a specified radius and angle.
- `Rectangle`: A rectangular shape with a specified width and height.
- `Polygon`: A polygon shape made from a list of 2D coordinates.

4.2.3 *3D Shapes*

- `Cube`: A cube with a specified width, height and depth.
- `Sphere`: A sphere with a specified radius.
- `Cylinder`: A cylinder with a height and top/bottom radii.
- `Polyhedron`: An arbitrary 3D shape made from connected triangles or quads.

4.2.4 *Transformations*

- `Translate`: Used to translate an object.
- `Rotate`: Used to rotate an object.
- `Scale`: Used to scale an object.
- `Mirror`: Used to mirror an object.

4.2.5 *Boolean Operations*

- `Union`: A combination of multiple shapes into a single shape.
- `Difference`: A boolean difference of two or more shapes.
- `Intersection`: A boolean intersection of two or more shapes.

4.2.6 *Extrusions*

- `LinearExtrusion`: An extrusion of a 2D object linearly along the z axis.
- `RotateExtrusion`: An extrusion of a 2D object around the z axis.

4.3 BACKENDS

The backends are responsible for code generation. They receive an AST instance, traverse it and emit backend specific code.

At the time of this writing, only one backend has been implemented: The OpenSCAD backend. But it would be very easy to add additional backends in the future.

4.3.1 *Creating Custom Backends*

To be valid, a custom backend simply needs to implement the following interface:

```
class CustomBackend(object):
    def __init__(self, ast):
        """Initialize backend using the provided AST."""
    def generate(self):
        """Generate code from AST and return it
        as a unicode string."""
```

The code generated by a backend is returned as a unicode string. It can then be printed to the terminal or used for further processing.

Implementation details regarding the already existing backend can be found in section 7.2.

In duck typed programming languages like Python, interfaces are usually not declared explicitly. The adherence to the interface is only judged by the behavior of the implementation: If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

4.4 SHAPES

The `shapes` package is a key component of *Tangible*. It provides a hierarchical collection of predefined shapes that can be used directly to generate three dimensional data visualizations.

The package is organized into different files:

- `base.py`: Base classes for all shape objects.
- `mixins.py`: Mixins used in the shape classes, mostly used for data validation.
- `bars.py`: Bar like shapes.
- `vertical.py`: Vertical shapes, e.g. towers.
- `pie.py`: Circular "pie" shapes.

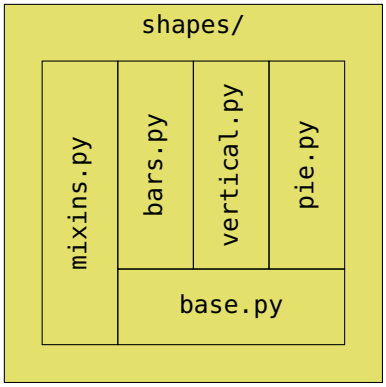


Figure 4: Shapes Architecture

4.4.1 *Base Class*

The class `Shape` in `shapes/base.py` is the base class for all predefined shapes in *Tangible*:

```
class BaseShape(object):
    def _build_ast(self):
        raise NotImplementedError("_build_ast method not implemented.")

    def render(self, backend):
        ast = self._build_ast()
        return backend(ast).generate()

class Shape(BaseShape):
    def __init__(self, data):
        self.data = utils.ensure_list_of_lists(data)
        if len(self.data[0]) == 0:
            raise ValueError("Data may not be empty.")
```

Each shape is initialized with the data as the first positional argument. Using a helper function, single lists with one dimensional data are converted to nested lists, to simplify the rendering code. Empty data is not allowed.

The `_build_ast()` method is not implemented in the base class. An inheriting class needs to override the method and return an AST.

Finally, the `render(backend)` method renders the AST using the specified backend class and returns the resulting source code as a unicode string, as described in section 4.3.

4.4.2 *Mixins*

The mixin classes are used in combination with Python's multiple inheritance system to provide "pluggable" generic data validation. At the time of this writing, the following mixins are available:

- `Data1DMixin`: Ensures that data contains exactly 1 dataset.

- `Data2DMixin`: Ensures that data contains exactly 2 datasets.
- `Data3DMixin`: Ensures that data contains exactly 3 datasets.
- `Data4DMixin`: Ensures that data contains exactly 4 datasets.
- `DataNDMixin`: Used for shapes where the number of data dimensions is not relevant. But it asserts that the data is not empty and that all data items are of a sequence type.
- `SameLengthDatasetMixin`: Ensures that all datasets have the same length.

The mixins are properly implemented using Python's argument list unpacking (`*args, **kwargs`) and `super()` calls, so that a class can use multiple mixins without breaking the inheritance chain. A good example where this is used is the `RectangleTower2D` shape:

```
class RectangleTower2D(Data2DMixin,
    SameLengthDatasetMixin, VerticalShape):
    # ...
```

4.4.3 *Shape Classes*

The final shape classes are grouped into three categories: Bar shapes, vertical shapes and pie shapes.

- Bar shapes consist of several bars that start on `z=0` and have a height depending on the corresponding datapoint. They can be aligned in rows, and rows can be combined to create 3D bar graphs.
- A vertical shape is a shape with layers stacked on top of each other, with a fixed layer height, for example a round tower where the radius of each layer corresponds to the datapoint.
- A pie shape can represent data as angle, height or radius of the corresponding slice. It is also possible to define an inner radius (→donut) and to explode the slices.

The naming of the shape classes follows a consistent pattern: First a descriptive name of the shape (e.g. `RhombusTower` or `RadiusHeightPie`), then the dimensionality of the data (e.g. `1D`, `4D` or `ND`). A way to describe data dimensionality in Python terms would be: *n-dimensional data is a list containing `n` lists*.

4.5 UTILS

4.5.1 *Scales*

The module `scales.py` contains functions for mapping an input domain to an output range (the codomain). For example it could be used to normalize temperatures between 0°C and 100°C to values between 1 and 10. The scales are inspired by the quantitative scales in `d3.js`[1].

At the time of this writing, only a linear scale has been implemented. It accepts three parameters: The domain, the codomain (output range), and whether to clamp the values to the output range or not.

```
>>> from tangible import scales
>>> temperatures = [32, 60, 100, 0, 120, -50]
>>> domain = [0, 100]  # Input range
>>> codomain = [1, 10]  # Output range
>>> scale = scales.linear(domain, codomain, clamp=False)
>>> scale_clamp = scales.linear(domain, codomain=, clamp=True)
```

The returned object is the actual scaling function:

```
>>> scale(0)
1.0
>>> scale(50)
5.5
>>> scale(100)
10.0
```

A very convenient way to use scales is by applying them using the `map()` function:

```
>>> map(scale, temperatures)
[3.88, 6.3999999999999995, 10.0, 1.0, 11.799999999999999, -3.5]
>>> map(scale_clamp, temperatures)
[3.88, 6.3999999999999995, 10.0, 1.0, 10, 1]
```

Logarithmic and exponential scales as well as dynamic resizing of domains / codomains are currently not implemented, but will follow in the future.

4.5.2 *Helper Functions*

The module `utils.py` contains different helper functions to simplify common tasks.

1 https://github.com/mbostock/d3/wiki/Quantitative-Scales

4.5.2.1 *pairwise(iterable)*

This function returns a generator acting as a sliding window over an iterable. Each item returned by the generator is a 2-tuple containing two consecutive items from the original iterable.

Example:

```
>>> from tangible import utils
>>> pairs = utils.pairwise([1, 2, 3, 4, 'a'])
>>> pairs
<itertools.izip object at 0xeea098>
>>> list(pairs)
[(1, 2), (2, 3), (3, 4), (4, 'a')]
```

4.5.2.2 *reduceby(iterable, keyfunc, reducefunc, init)*

This function combines the functionality of `itertools.groupby()` and `reduce()`. It iterates over the iterable and aggregates the values using the specified reduce function and init value as long as `keyfunc(item)` returns the same value. Each time the key changes, the aggregated value is yielded.

A possible use case could be the aggregation of website visits, grouped by month. Example:

```
>>> from datetime import date
>>> from tangible import utils
>>> visits = [(date(2013, 1, 24), 27),
...           (date(2013, 1, 26), 4),
...           (date(2013, 2, 17), 19),
...           (date(2013, 3, 11), 23),
...           (date(2013, 3, 14), 42)]
>>> keyfunc = lambda x: x[0].month
>>> reducefunc = lambda x, y: x + y[1]
>>> groups = utils.reduceby(visits, keyfunc, reducefunc, 0)
>>> groups
<generator object reduceby at 0xedc5a0>
>>> list(groups)
[31, 19, 65]
```

The corresponding SQL statement would be:

```
SELECT SUM(visit_count) FROM visits GROUP BY MONTH(visit_date);
```

4.5.2.3 *connect_2d_shapes(shapes,layer_distance, orientation)*

This is quite a complex function. It connects a list of 2D shapes into a 3D shape using the desired layer distance. The `orientation` argument specifies, whether the shapes should be joined horizontally or vertically.

The main part of the function has separate implementations depending on the 2D object. Circles are connected by joining cylinders, while rectangles and polygons are connected by joining polyhedra.

Example:

```
>>> from tangible import utils, ast
>>> shapes = [ast.Circle(3), ast.Circle(8), ast.Circle(5)]
>>> utils.connect_2d_shapes(shapes, layer_distance=10,
...     orientation='vertical')
<AST/Union: 21376464>
```

4.5.2.4 *_quads_to_triangles(quads)*

This function converts a list of quads (4-tuples) to a list of triangles (3-tuples). This is mostly because many backends require surface meshes to consist of triangles, without support for quads.

Example:

```
>>> from tangible import utils
>>> quads = [(0, 1, 2, 3)]
>>> utils._quads_to_triangles(quads)
[(0, 1, 2), (0, 2, 3)]
```

The function is mostly used internally. A stable API is not guaranteed.

4.5.2.5 *_ensure_list_of_lists(data)*

This function ensures that the argument is a list of lists. If it doesn't contain lists or tuples, it is wrapped in a list and returned.

Example:

```
>>> utils._ensure_list_of_lists([1, 2, 3])
[[1, 2, 3]]
>>> utils._ensure_list_of_lists([[1, 2], [3]])
[[1, 2], [3]]
```

The function is mostly used internally. A stable API is not guaranteed.

5 EXAMPLES

The following pages demonstrate a few code examples of how to use the *Tangible* library.

5.1 A SIMPLE TOWER

This example describes a simple round tower where the radius of the layers corresponds to the datapoint. The dataset describes the number of web site visits on `http://blog.dbrgn.ch/` during the month of September 2013. The value range is normalized to a range between 10 and 50 using a linear scale.

```
from tangible import scales
from tangible.shapes.vertical import CircleTower1D
from tangible.backends.openscad import OpenScadBackend

# Normalize raw data
visits = [53, 69, 86, 92, 81, 76, 37, 36, 62, 76, 72, 67, 55, 61, 54,
          72, 92, 84, 78, 75, 45, 48, 85, 81, 83, 69, 68, 66, 62, 115]
scale = scales.linear([min(visits), max(visits)], [10, 50])
datapoints = map(scale, visits)

# Create shape
tower = CircleTower1D(datapoints, layer_height=10)

# Render OpenSCAD code
code = tower.render(backend=OpenScadBackend)
print code
```

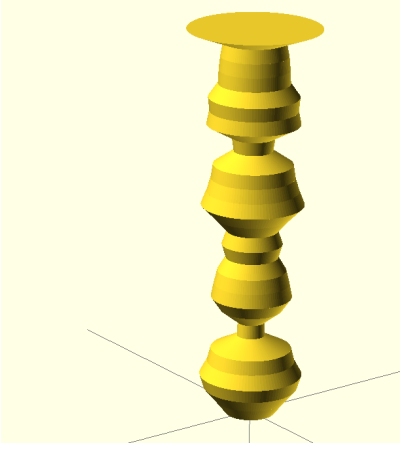


Figure 5: A CircleTower1D shape

5.2 MULTI DIMENSIONAL DATA

Here we have two dimensional data, represented as two lists of integers. The first list should be mapped to the angle of the "pie slices", while the second list should be mapped to the height of each slice. Additionally, we'll add a center radius to make the model look like a donut, and we'll explode the slices.

```
from tangible.shapes.pie import AngleHeightPie2D
from tangible.backends.openscad import OpenScadBackend

# Data
datapoints = [
    [30, 30, 5, 5, 20], # Angle
    [18, 23, 20, 15, 10], # Height
]

# Create shape
pie = AngleHeightPie2D(datapoints, inner_radius=4, explode=1)

# Render OpenSCAD code
code = pie.render(backend=OpenScadBackend)
print code
```

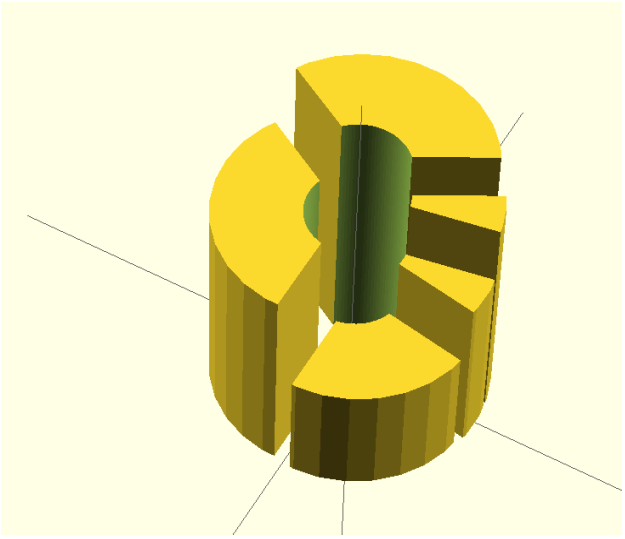


Figure 6: An AnglePie2D shape

5.3 READING DATA FROM CSV

Often the data that you want to visualize is not already available as a Python datastructure, but in formats like JSON or CSV. Here's a small example where website visitor data is read from the CSV exported by Google Analytics. Then the number of visits and the average visit duration are mapped to the distance between opposing corners of a rhombus tower.

```
import csv
from datetime import timedelta
from tangible.shapes.vertical import RhombusTower2D
from tangible.backends.openscad import OpenScadBackend

# Read data into list
datapoints = [[], []]
with open('analytics-sep-13.csv', 'r') as datafile:
    reader = csv.DictReader(datafile)
    for row in reader:
        datapoints[0].append(int(row['Visits']))
        h, m, s = map(int, row['AvgDuration'].split(':'))
        duration = timedelta(hours=h, minutes=m, seconds=s)
        datapoints[1].append(duration.total_seconds())

# Create shape
tower = RhombusTower2D(datapoints, layer_height=10)

# Render OpenSCAD code
code = tower.render(backend=OpenScadBackend); print code
```

Here are the CSV contents:

```
Day,Visits,AvgDuration
9/1/13,53,00:00:51
9/2/13,69,00:01:01
9/3/13,86,00:01:24
...
```

And the resulting shape:

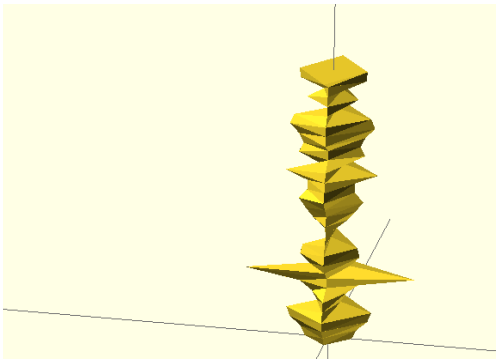


Figure 7: A RhombusTower2D shape from CSV data

5.4 GROUPED DATA

Some one dimensional datasets does not work well when visualized directly. An example would be website visitor statistics during a full year, a single bar graph would be much too wide. But by grouping the data from example 5.3 into months, a `BarsND` graph can be constructed:

```
import csv
from itertools import chain
from tangible import scales
from tangible.shapes.bars import BarsND
from tangible.backends.openscad import OpenScadBackend

# Read data into list
datapoints = [list() for i in xrange(9)]
with open('analytics-full-13.csv', 'r') as datafile:
    reader = csv.DictReader(datafile)
    for row in reader:
        date = row['Day']
        month = int(date.split('/', 1)[0])
        visits = int(row['Visits'])
        datapoints[month - 1].append(visits)

# Normalize data
all_datapoints = list(chain.from_iterable(datapoints))
scale = scales.linear([min(all_datapoints), max(all_datapoints)],
                      [10, 150])
datapoints = map(lambda x: map(scale, x), datapoints)

# Create shape
bars = BarsND(datapoints, bar_width=7, bar_depth=7)

# Render OpenSCAD code
code = bars.render(backend=OpenScadBackend); print code
```

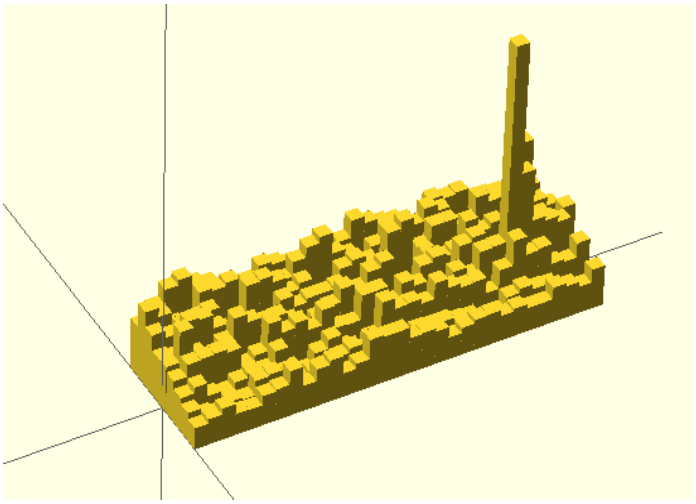


Figure 8: A BarsND shape from CSV data grouped by month

5.5 CREATING CUSTOM SHAPES FROM AST

It's not necessary to rely on the provided shape classes only, you can also create your own shapes by using the AST objects directly.

The easiest and cleanest way to do this, is to create a subclass of the `BaseShape` class and to override its `_build_ast` method:

```
from tangible.shapes.base import BaseShape
from tangible import ast
from tangible.backends.openscad import OpenScadBackend

# Create custom shape
class Cogwheel(BaseShape):
    def _build_ast(self):
        cogs = []
        for i in xrange(18):
            cog = ast.Rectangle(2, 2)
            translated = ast.Translate(9.5, -1, 0, cog)
            rotated = ast.Rotate(i * 30, (0, 0, 1), translated)
            cogs.append(rotated)
        return ast.Union([ast.Circle(radius=10)] + cogs)

# Render shape
f = Cogwheel()
code = f.render(backend=OpenScadBackend)
print code
```

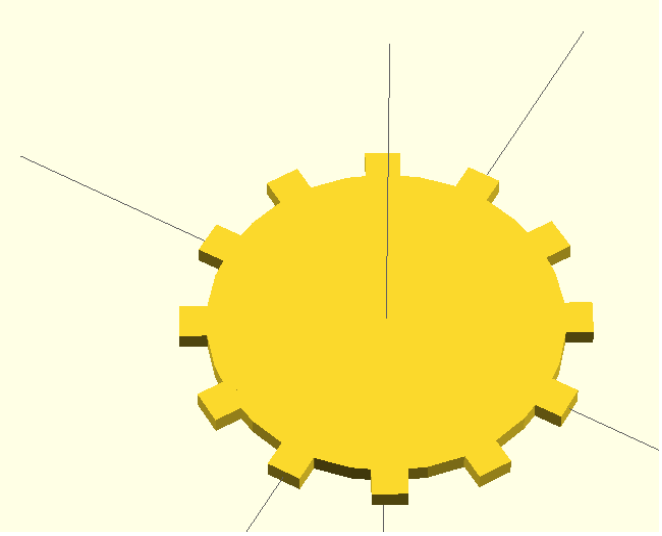


Figure 9: A custom cogwheel shape

Part III

THE DEVELOPMENT PROCESS

6 DEVELOPMENT PROCESS & TOOLS USED

6.1 CODING GUIDELINES

The PEPs (Python Enhancement Proposals) are Python's way of continuously improving the language in a community driven process. Any community member can submit a proposal for a language change, which is then discussed and accepted or rejected.

Coding guidelines are important in order to achieve a consistent style throughout the codebase. In the Python world, the PEP8 style guide [11] has seen near ubiquitous adaptation and should be used for all projects in order to aid the legibility of the source code.

> *One of Guido's key insights is that code is read much more often than it is written. The guidelines provided here are intended to improve the readability of code and make it consistent across the wide spectrum of Python code. As PEP 20 says, "Readability counts".*
>
> — *Style Guide for Python Code [11]*

Tangible follows most parts of PEP8, with two exceptions:

- While line lengths below 80 characters are the ideal case, lines with up to 99 characters are still acceptable, if it makes the code more readable.
- Errors `E126-E128`, which specify indentation rules for multi-line statements, can be ignored.

The adherence to these coding guidelines is tested automatically by the test suite (see section 6.3). Violations are counted as test failures.

6.2 FUTURE IMPORTS

While the first version of Python 3 has been released back in 2008, it has still not completely managed to replace the older 2.x versions. As a result, the *Tangible* codebase currently targets Python 2.

But in the past year several things have happened that accelerated the adoption rate of Python 3. Most importantly, several big Linux distributions like Arch Linux[1] and Fedora[2] have decided to move to Python 3 as the default Python implementation. Another important factor was the newly added Python 3 support in big Python frameworks like Django[3].

In view of these facts, the *Tangible* source code is written in a forward compatible way by adding "future-imports" to the top of every

1 https://www.archlinux.org/
2 http://fedoraproject.org/
3 https://www.djangoproject.com/

code file. Python provides a module called `future` which contains backports of newer language features to older Python versions. By using these imports, the migration process to newer language versions can be substantially simplified.

In the *Tangible* Project, the following preamble should be added to every source code file:

```
# -*- coding: utf-8 -*-
from __future__ import print_function, division
from __future__ import absolute_import, unicode_literals
```

This results in the following effects:

- The `# -*- coding: utf-8 -*-` line tells the Python interpreter that this file source is UTF8-encoded. In Python 3 UTF8 encoding will become the default.
- The `print_function` import removes the `print` statement and adds a `print()` function.
- When activating the `division` import, division of two integers results in a `float` value instead of the old, lossy way of returning a floored integer. When floor division is explicitly desired, the `//` operator should be used instead.
- By importing `absolute_import`, Python prioritizes absolute imports over relative imports. This fixes a few issues with import name clashes.
- The `unicode_literals` import is the one with the most consequences of all the future imports listed above. It changes the default type of strings from bytestrings to unicode objects. By eliminating this Python 2/3 inconsistency from the beginning, many hard to spot migration bugs can be prevented.

The choice of future imports is based on the article *Quick Tips on Making Your Code Python 3 Ready* by Hristo Deshev [5].

6.3 TESTING

> *More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded — indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest.*
> — *Boris Beizer [1]*

Testing is not a question of "whether", but rather a question of "how". By having a high test coverage of your code base, code can be changed without any fear of overlooking newly introduced bugs. Additionally, the TDD methodology greatly aids both the process of writing tests, as well as the process of writing software. Writing tests before actually implementing the code leads to better structured code with less bugs than if testing features after their implementation.

Test Driven Development (TDD) is a software development methodology where tests are written for every feature before it is implemented.

This section can be divided into three subsections: Writing the tests, running the tests and measuring test coverage.

6.3.1 *Writing Tests*

Tangible uses the pytest[4] framework for writing tests. pytest provides both tools for easy and lightweight testing as well as a simple method for test discovery.

In contrast to Python's builtin `unittest` module, which was heavily inspired by Java, pytest does not require the developer to use explicit methods for expressing assertions like `assertEquals` or `assertGreater`. Instead it relies solely on the `assert` keyword while still providing useful error messages by using language reflection.

A simple test could just look like the following example:

```
def test_the_answer():
    assert 19 + 23 == 42
```

In contrast, the `unittest` version would look like this:

```
import unittest
class TestTheAnswer(unittest.TestCase):
    def testIt(self):
        self.assertEquals(19 + 23, 42)
```

As the Zen of Python [9] states, *"Simple is better than complex."*. In this regard pytest seems to be a big improvement over the `unittest`. But simpler asserts are not the only advantage that pytest offers over unittest. Another feature that has been used extensively in *Tangible* is parametrized testing. By specifying a set of possible values for the test parameters using the `pytest.mark.parametrize` decorator, multiple tests are generated from a single test function. If one of five input values makes the test fail, the test result would be 4 successful tests out of a total of 5, whereas putting all the values in a nonparametrized function would lead to a single failing test.

Here is an example of a parametrized test from the *Tangible* test suite:

4 `http://pytest.org/`

```python
import pytest
from tangible import scales

@pytest.mark.parametrize(('param', 'clamp', 'expected'), [
    (2, False, 10),
    (3.5, False, 17.5),
    (6, False, 30),
    (6, True, 20),
])
def test_linear(param, clamp, expected):
    domain, codomain = (2, 4), (10, 20)
    scale = scales.linear(domain, codomain, clamp)
    assert scale(param) == expected
```

6.3.2 *Running Tests*

pytest offers highly customizable test discovery out of the box. After configuring the patterns to be used for detecting tests, the suite can be run by simply issueing `py.test` in the main project directory.

But manually running tests is a step that's often forgotten while developing. Tests are better when they're fully automated. Travis CI (`https://travis-ci.org/`) is a startup company that offers free continuous integration for open source projects. A minimal configuration file is all that's needed to get everything up and running. The test results are presented in an easy to understand manner in the web browser:

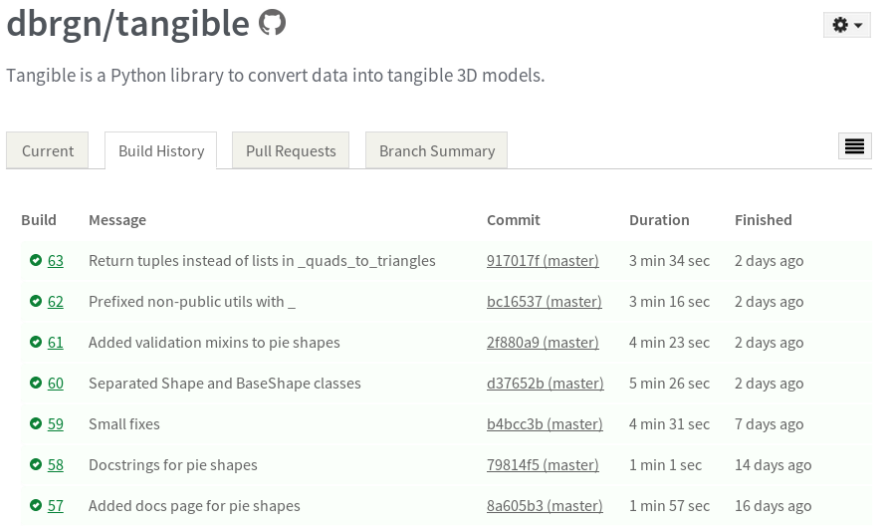


Figure 10: Travis CI

6.3.3 *Measuring Test Coverage*

Simply having many tests does not necessarily mean that the code is well tested. A high test coverage does not either, but it's a good

indicator on how many lines of code have been hit by the tests, and how many haven't.

In the *Tangible* project, coverage is measured through the coverage.py[5] module. The result is displayed each time the test suite runs thanks to the pytest-cov[6] plugin for pytest.

Change in test coverage is tracked by the Coveralls service (`https://coveralls.io/`). The current coverage percentage can be displayed in the README file by embedding a dynamic image from the Coveralls server. This way, the current coverage is always visible.

6.4 TOOLS USED

During the time of this thesis, the following tools have been used:

- The Vim text editor for editing both program source code and documentation.
  `http://www.vim.org/`
- Flake8 for code style checks and static code analysis.
  `https://flake8.readthedocs.org/`
- pytest for running the test suite.
  `http://pytest.org/`
- Travis CI for automated testing.
  `https://travis-ci.org/`
- Coveralls for automated test coverage measuring.
  `https://coveralls.io/`
- OpenSCAD to convert the generated model code to actual STL files.
  `http://www.openscad.org/`
- Makerbot / Makerware to test-print a few of the generated 3D models.
  `http://www.makerbot.com/makerware/`
- LaTeX for typesetting this project documentation.
  `http://www.latex-project.org/`
- Sphinx for generating the online user documentation.
  `http://sphinx-doc.org/`
- Git and Github for version control.
  `http://git-scm.com/`
  `https://github.com/`

5 `http://nedbatchelder.com/code/coverage/`
6 `https://pypi.python.org/pypi/pytest-cov`

7

DESIGN DECISIONS & IMPLEMENTATION DETAILS

7.1 PAIRWISE ITERATOR

The pairwise iterator (in `tangible/utils.py`, see section 4.5.2.1) has an interesting implementation that might not be immediately obvious to someone new to Python and its standard library:

```
from itertools import tee, izip

def pairwise(iterable):
    a, b = tee(iterable)
    next(b, None)
    return izip(a, b)
```

The `tee(iterable, n)` function returns `n` independent iterators from a single iterable. The default value for `n` is `2`, so in the example above two iterators called `a` and `b` are created from the original iterable.

The second iterator is then advanced once by applying the `next()` function to it. The return value is discarded. This results in two iterators, one of them with an offset of `1`.

As a last step, the two iterators are zipped together. By using the `izip()` function instead of the regular `zip()`, a lazy generator is returned instead of a list. This decreases memory consumption, especially when handling large lists.

Now each time the returned generator is advanced one step, the sliding window is shifted by `1` position and the resulting tuple is returned, until the end of the iterator is reached.

```
>>> data = [1, 2, 3, 'a']
>>> pairs = pairwise(data)
>>> pairs
<itertools.izip object at 0x151def0>
>>> next(pairs)
(1, 2)
>>> next(pairs)
(2, 3)
>>> next(pairs)
(3, 'a')
>>> next(pairs)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
        StopIteration
```

7.2 CODE GENERATION

Code generating code is often something quite messy, with many conditionals and a lot of `print` statements and string formatting. Such an approach is both hard to read and hard to maintain. Additionally, it does not reflect the structure of the generated code.

In the OpenSCAD backend implementation, *Tangible* uses an approach proposed by Tomer Filiba [6], which builds upon Python's context managers to generate nested blocks of code.

Context Managers are Python constructs that create a runtime context for a piece of code when used in combination with the `with` statement. They provide enter- and exit-hooks that are invoked before and after executing that code.

There is a top level class called `Program` which exposes a `statement` method and a `block` context manager. The class holds a stack of blocks and a list of child blocks and statements. Each time a block is entered (by using a `with`-statement), it is pushed to the stack and appended to the list of children. When leaving the context manager, the block is removed again from the stack.

The final code is generated by walking the list of children in the `Program` instance recursively. This is also the point where language-specific features can be implemented, for example indentation of a block in Python or inserting curly braces in Java or C.

A special feature that was implemented in the code generation is the support for predefined code snippets to be included in the generated output. This part of the code is called the "preamble". Snippets can be inserted into the preamble multiple times, but they're rendered only once. This has proven to be very useful while implementing code generation for circle sectors (see section 7.3).

Right now all the described classes are contained in the OpenSCAD backend. By generalizing the code, it would be possible to create a base class for all code generation backends, with the possibility to configure the language-specific details in a single subclass. This might be a good idea for a future version of *Tangible*.

An extract from the actual code which decides how the AST is mapped to the backend syntax is shown on the next page.

```
class OpenScadBackend(object):
    """Render AST to OpenSCAD source code."""

    def __init__(self, ast):
        self.ast = ast

    def generate(self):
        prgm = Program()
        BLOCK = prgm.block
        STMT = prgm.statement
        PRE = prgm.preamble
        SEP = prgm.emptyline

        def _generate(node):
            """Recursive code generating function."""

            istype = lambda t: node.__class__ is t

            # Handle lists
            if istype(list):
                for item in node:
                    _generate(item)

            # Simple statements
            elif istype(ast.Circle):
                STMT('circle({0})', node.radius)
            elif istype(ast.Rectangle):
                STMT('square([{0}, {1}])', node.width, node.height)

            # Blocks
            elif istype(ast.Union):
                with BLOCK('union()'):
                    _generate(node.items)

            # (...)

        _generate(self.ast)

        return prgm.render()
```

7.3 CIRCLE SECTORS IN OPENSCAD

By default, OpenSCAD does not support circle sectors. Therefore the shape had to be developed manually as a module.

```
module circle_sector(r, a) {
    a1 = a % 360; a2 = 360 - (a % 360);
    if (a1 <= 180) {
        intersection() {
            circle(r);
            polygon([
                [0,0],
                [0,r],
                [sin(a1/2)*r, r + cos(a1/2)*r],
                [sin(a1)*r + sin(a1/2)*r, cos(a1)*r + cos(a1/2)*r],
                [sin(a1)*r, cos(a1)*r],
            ]);
        }
    } else {
        difference() {
            circle(r);
            mirror([1,0]) {
                polygon([
                    [0,0],
                    [0,r],
                    [sin(a2/2)*r, r + cos(a2/2)*r],
                    [sin(a2)*r + sin(a2/2)*r, cos(a2)*r + cos(a2/2)*r],
                    [sin(a2)*r, cos(a2)*r],
                ]);
            };
        }
    }
};
```

The base concept is a boolean combination of a circle and a polygon, depending on the angle. If the angle is less than or equal to 180°, the resulting shape is the intersection of the circle and the polygon. If it's larger than 180°, the difference between the two shapes is returned.

The polygon always consists of five points, which are calculated as shown in the following table for angles less or equal to 180°. For angles between 180° and 360°, the polygon is simply mirrored along the y axis.

x	y
0	0
0	r
$\sin(\alpha/2) \cdot r$	$r + \cos(\alpha/2) \cdot r$
$\sin(\alpha) \cdot r + \sin(\alpha/2) \cdot r$	$\cos(\alpha) \cdot r + \cos(\alpha/2) \cdot r$
$\sin(\alpha) \cdot r$	$\cos(\alpha) \cdot r$

The following six images show the circle and polygon combinations for 45, 90, 180, 225, 270 and 315 degrees.

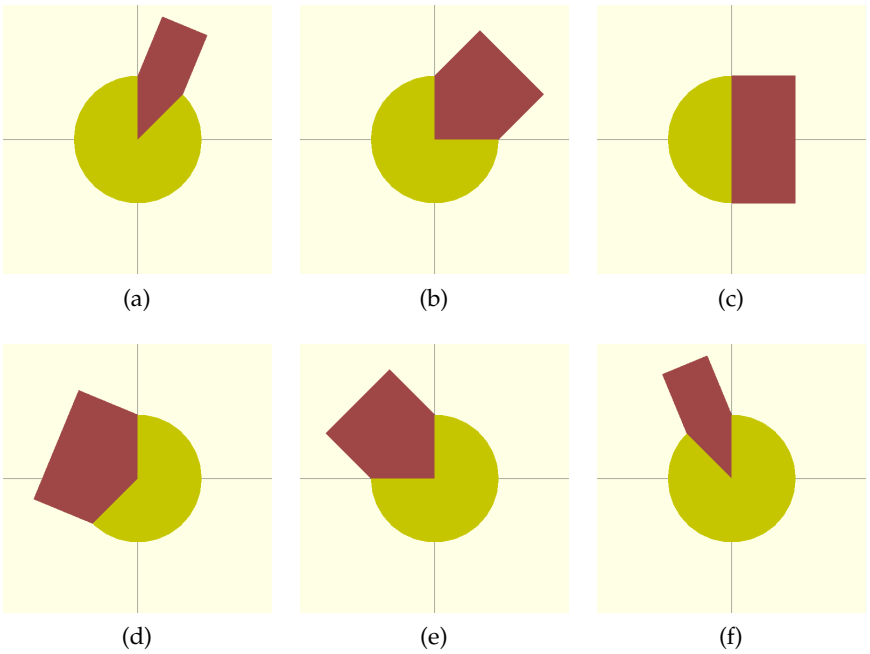


Figure 11: Circle and polygon combinations at different angles

By combining the two shapes in such a way, any circle sector can be created. Example for `circle_sector(10, 225)`:

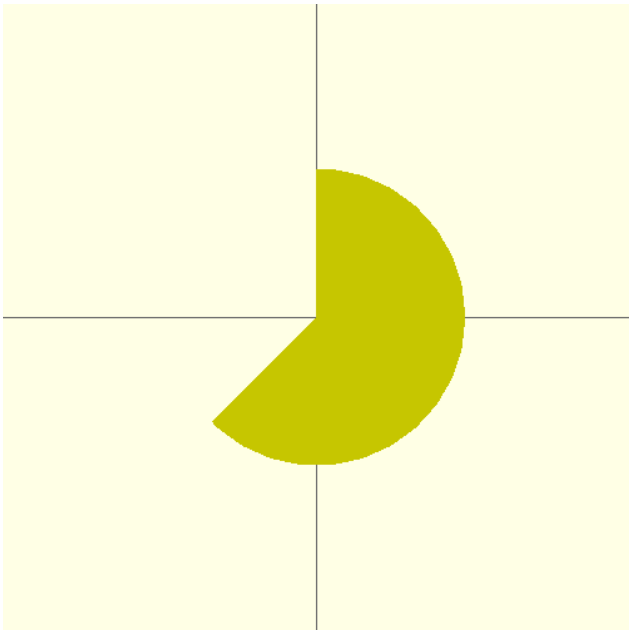


Figure 12: The resulting circle sector for an angle of 225°

The module source code is used in the OpenSCAD backend implementation as a preamble snippet.

Part IV

APPENDIX

BIBLIOGRAPHY

[1] Boris Beizer. *Software Testing Techniques*. Dreamtech, 2003. ISBN 9788177222609.

[2] Erik De Bruijn. On the viability of the open source development model for the design of physical objects. Master's thesis, Tilburg University, 2010. URL `http://thesis.erikdebruijn.nl/master/MScThesis-ErikDeBruijn-2010.pdf`.

[3] S. Scott Crump. Apparatus and method for creating three-dimensional objects, June 1992. URL `https://www.google.com/patents/US5121329`.

[4] Karen Day. Andreas Nicolas Fischer: Data Visualization Art, 2009. URL `http://www.coolhunting.com/culture/andreas-nicolas.php`.

[5] Hristo Deshev. Quick Tips on Making Your Code Python 3 Ready, 2012. URL `http://stackful-dev.com/quick-tips-on-making-your-code-python-3-ready.html`.

[6] Tomer Filiba. Code Generation using Context Managers, 2012. URL `http://tomerfiliba.com/blog/Code-Generation-Context-Managers/`.

[7] Vitaly Friedman. Data Visualization and Infographics. *Smashing Magazine*, January 2008. `http://www.smashingmagazine.com/2008/01/14/monday-inspiration-data-visualization-and-infographics/`.

[8] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D Representations for Software Visualization. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 27–ff, New York, NY, USA, 2003. ACM. ISBN 1-58113-642-0. doi: 10.1145/774833.774837. URL `http://doi.acm.org/10.1145/774833.774837`.

[9] Tim Peters. The Zen of Python, 2004. URL `http://www.python.org/dev/peps/pep-0020/`.

[10] Dylan Schenker. Miebach and Posavec – Data, Visualization, Poetry and Sculpture, 2012. URL `http://www.creativeapplications.net/theory/meibach-and-posavec-data-visualization-poetry-and-sculpture/`.

[11] Guido Van Rossum, Barry Warsaw, and Nick Coghlan. Pep8: Style Guide for Python Code, 2001. URL http://www.python.org/dev/peps/pep-0008/.

DECLARATION

Hereby I acknowledge,

- that I conducted this thesis by myself and without any external help, except with those, which are explicitly mentioned,
- that all used sources are cited academically correct, and
- that I didn't use any copyright protected materials (e.g. images) in an unauthorized manner.

Rapperswil, Fall 2013

Danilo Bargen, December 20, 2013