# Gds2Mesh

## 3D TCAD Model Constructor

Version 1.0.0

## Gds2Mesh User Guide:

Cogenda Pte Ltd

## License Grant



## Disclaimer



This documentation was typed in DocBook XML format, and typeset with the ConTEXt program. We sincerely thank the contributors of the two projects, for their excellent work as well as their generoisty.

# Contents

CHAPTER 1

# Using Gds2Mesh

The Gds2Mesh tool constructs 3D device models from planar mask layouts, according to a set of process rules and process parameters, and large through extruding 2D graphs to 3D objects.

In this chapter, the basic procedures of using the Gds2Mesh tool is described, using the examples shipped in the software package.

## The GUI Quick Start

A simple GUI is provided for beginners to access the Gds2Mesh tool.The main window of the GUI is shown in **Figure 1.1**, **p. 1**.

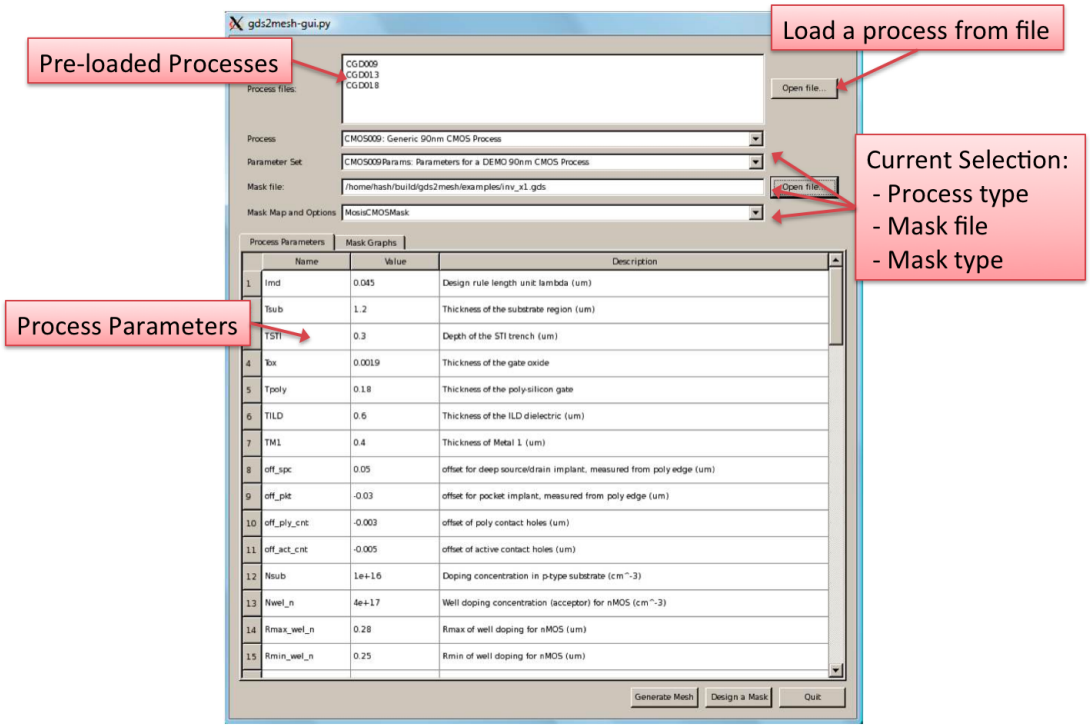



**Figure 1.1** Main window of the Gds2Mesh GUI.

To construct a device model, four pieces of information are needed:

- A set of process rules.
- Process parameters.
- A mask layout.
- A mask layer map.

One can easily identify the corresponding GUI controls in the main window for inputting the above information.

### Process Rules

Process rules are central to Gds2Mesh, as they define how 3D objects are constructed from 2D graphs, how doping profiles are placed, and the mesh constraints.

Some process rules are pre-loaded. For example, three demo CMOS process rules are pre-loaded.

One can also load process rules from files. For example, one could load the demo PN diode process from the file `examples/diode.py`.

Each process rule is a Python class extended from the class `ProcessBase`, which an advanced user may want to check out in the file `lib/ProcessDesc.py`.

Multiple processes can be loaded at any time, but only one is used for creating a device structure. One can select the current process rule in the drop-down list, along with the corresponding process parameters.

### Process Parameters

Each process is argumented by a set of parameters. The bottom half of the main window displays the list of parameters, with a short description for each of them. Editing the parameters to affect how the device model is built.

### Gdsii Masks

One can load a mask file in Gdsii format. In our example, the file `examples/inv_x1.gds` (**Figure 1.2**, **p. 3**) is loaded. Since the GDSII format does not in itself specify the meaning of the numeric layer numbers, one need to select a layer map from the drop-down list.

One can select the Mask Graph tab in the main window, to view the graphs in the mask layout, as shown in **Figure 1.3**, **p. 3**. All layers are listed in the left, and in the right, the graphs in the selected layers are displayed.

Different process rules may require different sets of mask layers. The demo CMOS process, for examples, requires the layers named `N_WELL`, `P_WELL`, `ACTIVE`, `POLY`, etc. The mask layout in the example GDSII files are designed with the MOSIS design rule, and the MOSIS layer map is pre-loaded. One just select this layer map in the drop-down list.

One can load other layer maps from `.py` files containing classes extended from the class `MaskBase`.

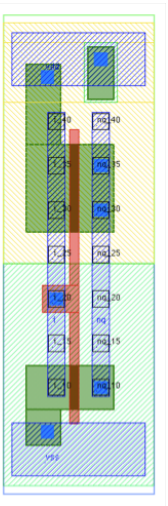


**Figure 1.2** Mask layout of the inverter.

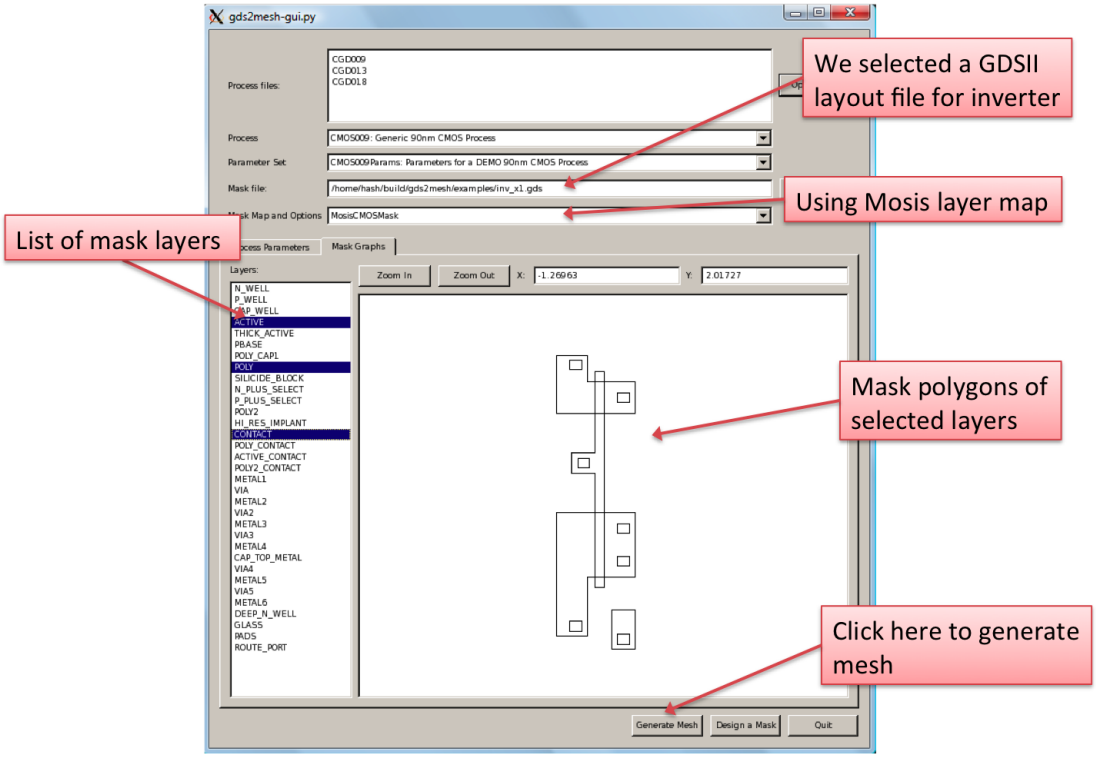



**Figure 1.3** Viewing graphs load from a Gdsii mask file.

## Generated Structure

One clicks the Generate Mesh button to build the device structure and mesh it. The generated mesh is saved to `inv_x1.tif3d`, and we can use VisualTCAD to examine it, as shown in **Figure 1.4, p. 4**. The correspondence between the mask and the generated mesh structure is apparent.

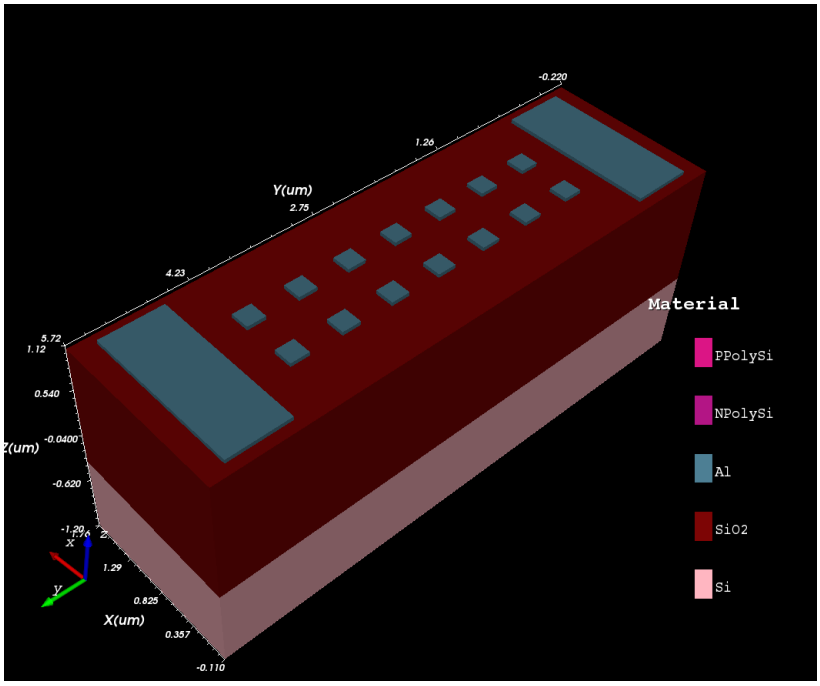



**Figure 1.4** Inverter structure generated by Gds2Mesh with the demo script.

If one hides the oxide region in the display, the inverter circuit with one n-channel and one p-channel transistors becomes apparent in **Figure 1.5**, **p. 4**. One also notes that mesh density is high in the active region, and sparse in the substrate, passivation oxide and metal regions.

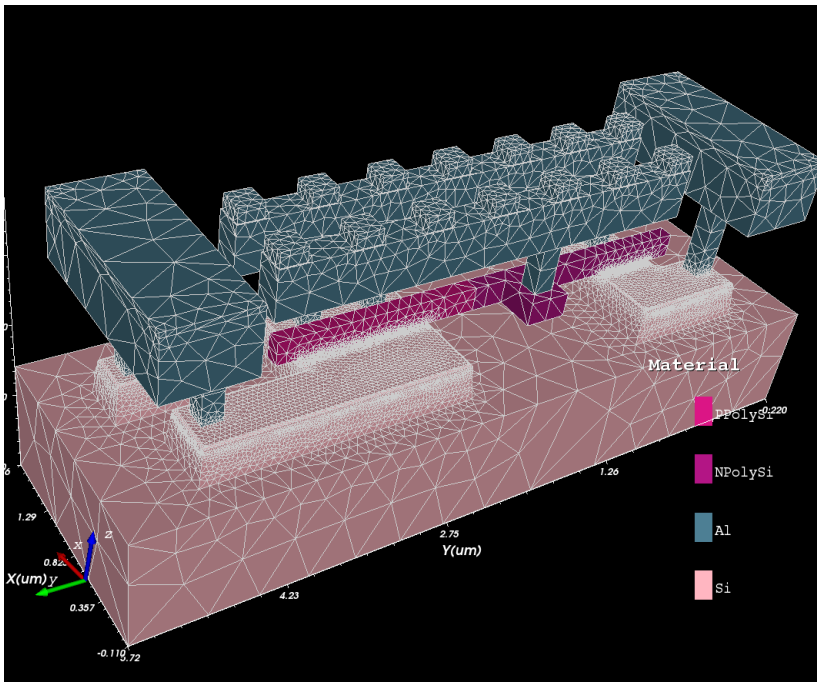



**Figure 1.5** Inverter structure with oxide region stripped.

The acceptor doping profile is displayed in **Figure 1.6**, **p. 5**, around the pMOSFET region.

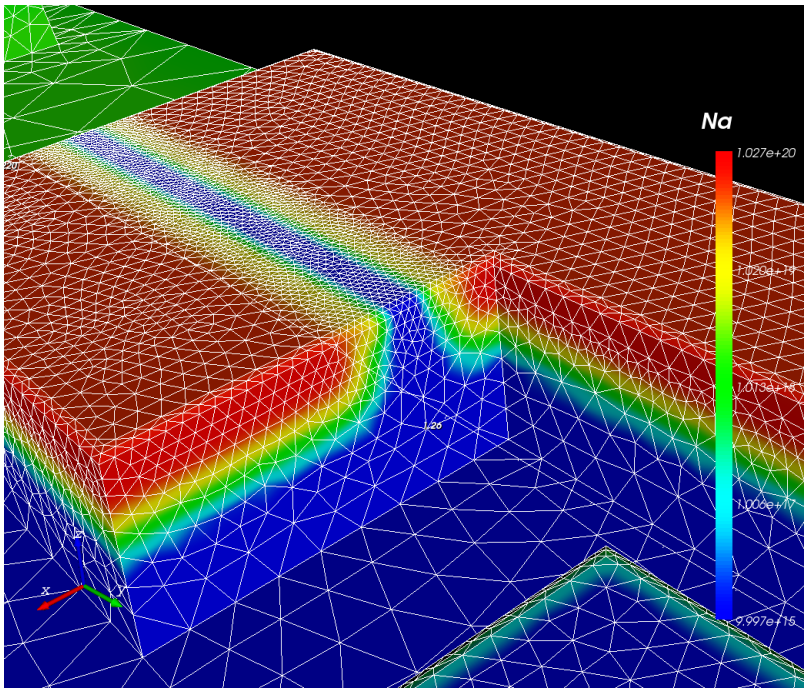



**Figure 1.6** Doping profile of the inverter structure.

One also observes the rounding of STI corners, and the high-quality, adaptive tetrahedral mesh, which Gds2Mesh is capable of.

## Simple Polygon Masks

Apart from loading a GDSII mask layout, one can define simple masks with polygon items. One can click the Design a Mask button in the main window to open the mask editing dialog window, as in **Figure 1.7**, **p. 6**.

Each mask layout consists of a set of graph items. Three types of items are currently supported: Rectangle, Circle and Polygon. Each type of item has several parameters to describe the position and shape.

Each item is assigned a layer name, and all items with the same layer name are combined by addition (union operation of the graph boolean algebra).

One can save these simple masks to files, and load them later. In the meanwhile, the simple "pickle" format is used (`.pkl`), but could be replaced by some more proper data format later.

In the main window, instead of loading GDSII mask files, one can load the .pkl mask files (**Figure 1.8**, **p. 6**). One also needs to select the layer map called SimpleMask, for these simple masks.

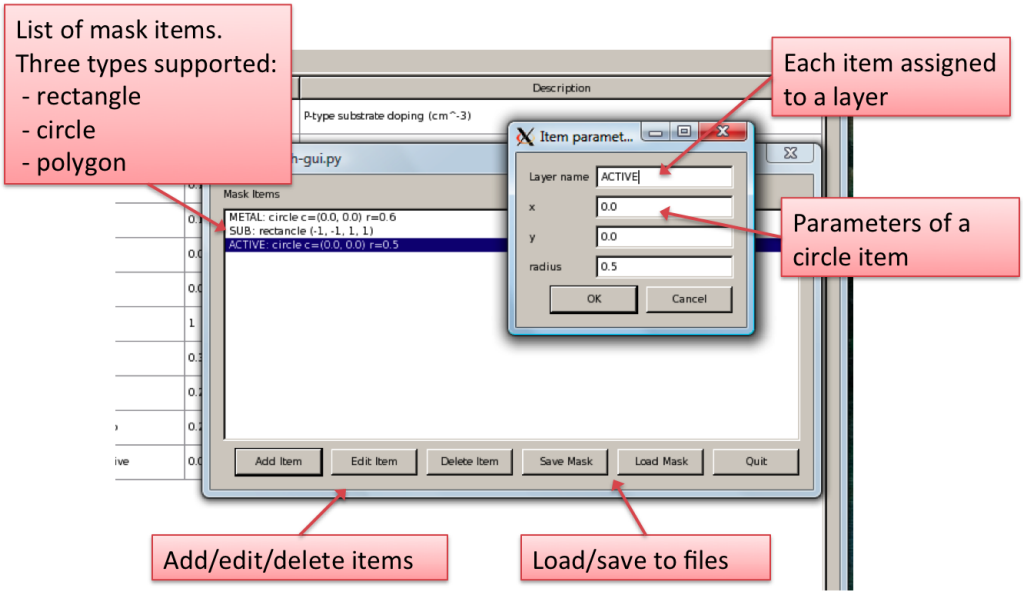



**Figure 1.7** Editing an item in a simple mask.

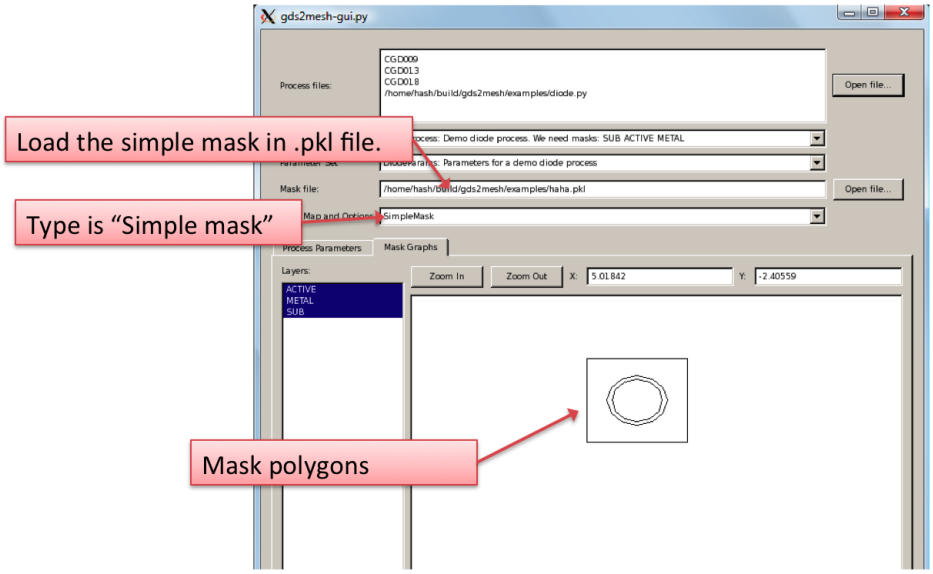



**Figure 1.8** Viewing graphs of the created simple mask.

In the preceding section, we have introduced the basic procedure of using the Gds2Mesh GUI. Since the Gds2Mesh tool is centered around the process rules written in Python scripts, it is necessary for advanced users to understand how the scripts work. In the next section, we give an overview on using the tool through the scripts.

# The Demo Script

Gds2mesh is driven by scripts written in Python programming language. Examples of the driver scripts are included in the `/opt/cogenda/gds2mesh/examples/` directory. To try out the examples, one makes a copy of that directory, and in that directory enters the following command

```
$ /opt/cogenda/gds2mesh/bin/gds2mesh demo.py
```

where `demo.py` is the name of the script file, residing in this directory. The following output showed up in the terminal

Screen output

```
$ /opt/cogenda/gds2mesh/bin/gds2mesh demo.py
       lmd                0.055 Design rule length unit lambda (um)
      Tsub                  1.2 Thickness of the substrate region (um)
      TSTI                  0.3 Depth of the STI trench (um)
       Tox               0.0032 Thickness of the gate oxide
     Tpoly                  0.2 Thickness of the poly-silicon gate
      TILD                  0.6 Thickness of the ILD dielectric (um)
       TM1                  0.4 Thickness of Metal 1 (um)
   off_spc                 0.06 offset for deep source/drain implant, measured
from poly edge (um)
   off_pkt                -0.04 offset for pocket implant, measured from poly
edge (um)
      Nsub                1e+16 Doping concentration in p-type substrate (cm^-3)
    Nwel_n                2e+18 Well doping concentration (acceptor) for nMOS
(cm^-3)
Rmax_wel_n                 0.25 Rmax of well doping for nMOS (um)
.
.
.
 SwapImprove2
 ImproveMesh
Mesh has 118759 points and 690153 elements
Writing TIF3D file...inv_x1.tif3d
writing mesh nodes
writing faces
writing elements
writing regions
writing boundaries
writing profiles
****************************************************************
```

### Driving Script

To understand how the demo script works, let us examine its content.

Program listing of the demo script.

```
from CGD013 import *                                                    1
                                                                        2
# default parameters for .13 CMOS process                               3
params = CMOS013.Params()                                               4
                                                                        5
# optionally we change some of the parameters                           6
#params.setParam('Nsub', 6e15)                                          7
                                                                        8
print params                                                            9
                                                                        10
# create a device structure using the .13 CMOS process                  11
device = CMOS013(params)                                                12
                                                                        13
fname = 'inv_x1'                                                        14
# load GDSII mask data                                                  15
device.setMask(MosisCMOSMask(fname+'.gds'))                             16
                                                                        17
# get the list of IO pads in the layout                                 18
print device.getIOPadList()                                             19
                                                                        20
# optionally, set the IO pads we want to build.                         21
# Otherwise, all possible IO pads are built.                            22
#device.setIOPadList(['i_20','nq_40'])                                  23
                                                                        24
# build geometry and doping profile                                     25
device.buildDevice()                                                    26
                                                                        27
# create mesh                                                           28
device.doMesh(0.3)                                                      29
                                                                        30
# save mesh in .tif3d format, to be simulated by Genius                 31
device.save(fname+'.tif3d')                                             32
```

In line 14-16, one sees that the mask layout file `inv_x1.gds` is loaded. The mask drawing of the inverter is shown in **Figure 1.2, p. 3**.

One first load the necessary libraries, which should be present at the beginning of every Gds2Mesh script. The libraries are located in `/opt/cogenda/gds2mesh/lib`. The library `CGD013` contains the fabrication process parameters and the procedures of building the mesh structure for the demo 0.13 μm CMOS process.

```
from CGD013 import *
```

We create a default parameter set, and optionally change some of the parameters. The full list of parameters, including a short description, is printed to terminal, as we have seen earlier. A device structure is created using this CMOS013 process.

```
# default parameters for .13 CMOS process
params = CMOS013.Params()

# optionally we change some of the parameters
#params.setParam('Nsub', 6e15)

print params

device = CMOS013(params)
```

Subsequently, we load the design from a mask file `inv_x1.gds`. The geometric objects in GDSII files are organized with numeric layer numbers. In order to assign meaning to these layers, a layer map must be used. In this case, the Mosis layer map is used.

```
fname = 'inv_x1'
# load GDSII mask data
device.setMask(MosisCMOSMask(fname+'.gds'))
```

The mask layout contains a set of IO pads, which will become contact terminals in TCAD simulation. One can optionally choose to use a selected list of IO pads, otherwise all of them are used.

```
print device.getIOPadList()
#device.setIOPadList(['i_20','nq_40'])
```

Finally, one builds the device structure, generate mesh of it, and save it to disk in the *TIF3D* format. Both Genius and VisualTCAD is capable of importing device structures in this format.

```
device.buildDevice()
device.doMesh(0.3)
device.save(fname+'.tif3d')
```

One notes that the impurity doping profile is placed in the CGD013 library, not in the driver script.

## Further Examples

The demo script is able to generate any circuit cell with the same CMOS process, given the appropriate mask layout. By changing the GdsII file names in the script, one can also generate the NAND gate and half-adder structure.

The script `sram.py` contains example of hand-crafted mask layout of an SRAM cell, also using the CMOS013 process. The `nmos013.py` script, on the other hand, does not use the ready-made process library. Instead, all geometry definition, profile placement, and mesh refinement instructions are contained in the driver script itself. The SRAM structure is shown in **Figure 1.9**, **p. 10**.

In principle, Gds2Mesh is able to construct models of any circuit blocks fabricated with planar processes, as long as a suitable mask layout (GdsII file) and a process library (similar to CMOS013) is present. For examples, the pharosc project (http://www.vlsitechnology.org) provides several free standard cell libraries, which have all been tested in Gds2Mesh.

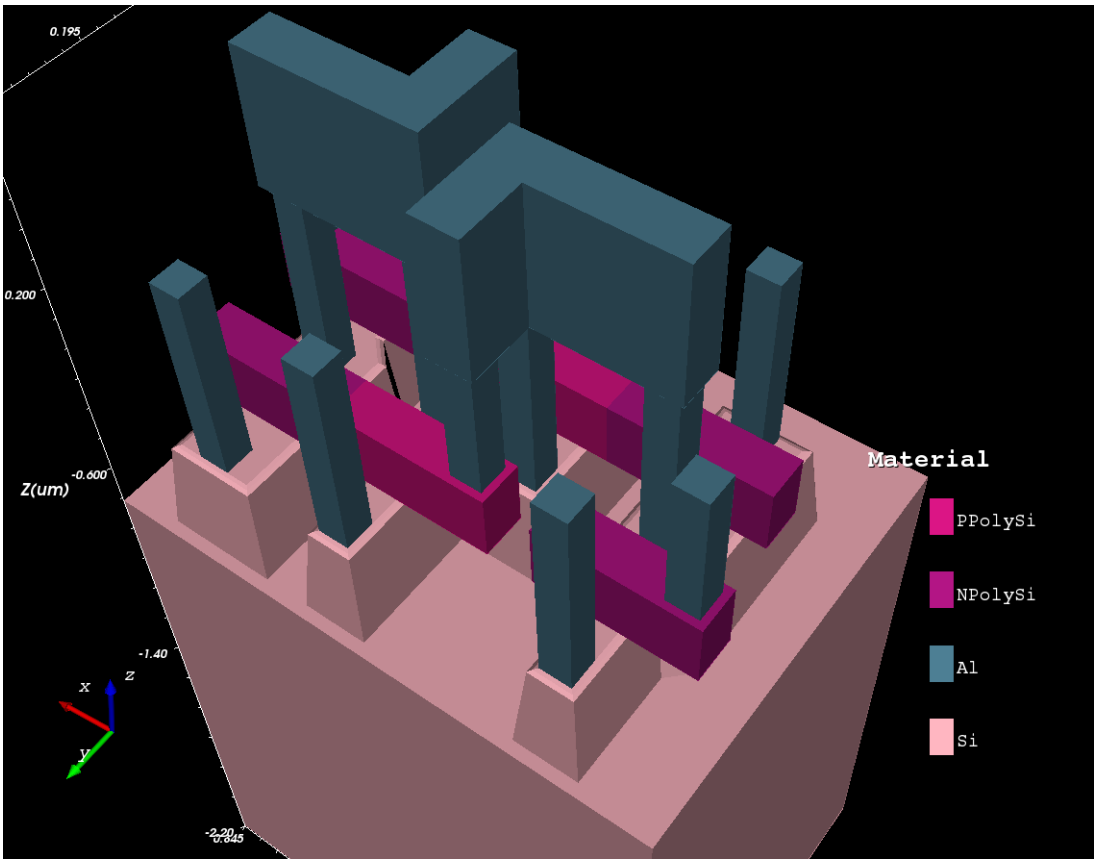



**Figure 1.9** Generated SRAM structure.

# Simulate Generated Structure in Genius

Genius and VisualTCAD with version 1.7.0 and above are capable of simulating device structures generated by Gds2Mesh.

The following input deck loads in the inverter we generated in the last section, and simulates its transient switching characteristics.

Program listing of the Genius input deck for the inverter.

```
#-------------------------------------------------------
# GENIUS 3D inverter example
# Need about 1.5h by XEON 5410 CPU (4 core)
#-------------------------------------------------------

GLOBAL T=300 Doping=1e21 ResistiveMetal=true

# Load 3D inverter
IMPORT TIF3D=inv_x1.tif3d

# set electrical pad
BOUNDARY ID=i_10  Type=solderpad Res=100 Cap=1e-15
BOUNDARY ID=nq_40 Type=solderpad Res=1e6 Cap=1e-15
BOUNDARY ID=vss   Type=solderpad Res=10  Cap=1e-13
BOUNDARY ID=vdd   Type=solderpad Res=10  Cap=1e-13
BOUNDARY ID=Sub   Type=ohmic


# set electrical power and signals used by simulation
VSOURCE ID=VCC Type=VDC Vconst=1.2
VSOURCE ID=GND Type=VDC Vconst=0.0
VSOURCE ID=VIN Type=VPulse Tdelay=0.1e-9 V1=0 V2=1.2 \
        TR=0.1e-9 PW=0.9e-9 TF=0.1e-9 PR=2e-9

# use poisson solver to get an approximate initial guess
METHOD Type=Poisson LS=GMRES PC=ASM
SOLVE

# set physical parameters
PMI Region=active_1 Type=mob model=HP
PMI Region=active_2 Type=mob model=HP
# disable high field mob in well/substrate region
MODEL Region=active_0 H.Mob=false
MODEL Region=sub H.Mob=false

```

```
# solve equilibrium state
METHOD Type=DDML1 NS=Newton LS=MUMPS damping=potential \
       toler.relax=1e6 relative.tol=1e-4
SOLVE Type=equ
EXPORT CGNS=inv_x1.cgns VTK=inv_x1.vtu

# ramp-up vdd
SOLVE Type=DC VScan=vdd Vstart=0 Vstep=0.1 Vstop=1.2
EXPORT CGNS=inv_x1.bias.cgns VTK=inv_x1.bias.vtu
# after vdd ramp-up, attach const voltage source to Vdd electrode
ATTACH Electrode=vdd Vapp=VCC

# do transient simulation with input pulse
ATTACH Electrode=i_10 Vapp=VIN
METHOD Type=DDML1 NS=Newton LS=MUMPS maxit=8 \
       toler.relax=1e6 relative.tol=1e-4
HOOK  Load=vtk  # export vtk file at each time step
SOLVE Type=tran tstart=1e-12 tstep=10e-12 tstepmax=0.2e-9 tstop=2e-9 \
      Vstepmax=0.1 out.prefix=inv_switch

END
```

While it is not the purpose of this document to explain the simulation detail, a few commands in the input deck must be highlighted.

In the `GLOBAL` command, one has to enable the `ResistiveMetal` option, introduced in the 1.7.0 version, for Genius to work with the generated structure. In the `ResistiveMetal` mode, metal regions has finite resistivity, and RC delay is included in the simulation.

```
GLOBAL T=300 Doping=1e21 ResistiveMetal=true
IMPORT TIF3D=inv_x1.tif3d
```

The IO pads of the circuit were labelled faces on the face of metal regions. These boundaries must be labelled as of the `SolderPad` type. The input terminal is *`i_10`*, and a load resistor is attached to the output terminal *`nq_40`*.

```
BOUNDARY ID=i_10  Type=solderpad Res=100 Cap=1e-15
BOUNDARY ID=nq_40 Type=solderpad Res=1e6 Cap=1e-15
BOUNDARY ID=vss   Type=solderpad Res=10  Cap=1e-13
BOUNDARY ID=vdd   Type=solderpad Res=10  Cap=1e-13
BOUNDARY ID=Sub   Type=ohmic
```

This input deck simulates first ramps up the power supplies in DC sweep mode, then simulates a cycle of switches of the inverter in transient mode. The simulated wave fronts of the input and output terminal are shown in Figure 1.10, p. 13.

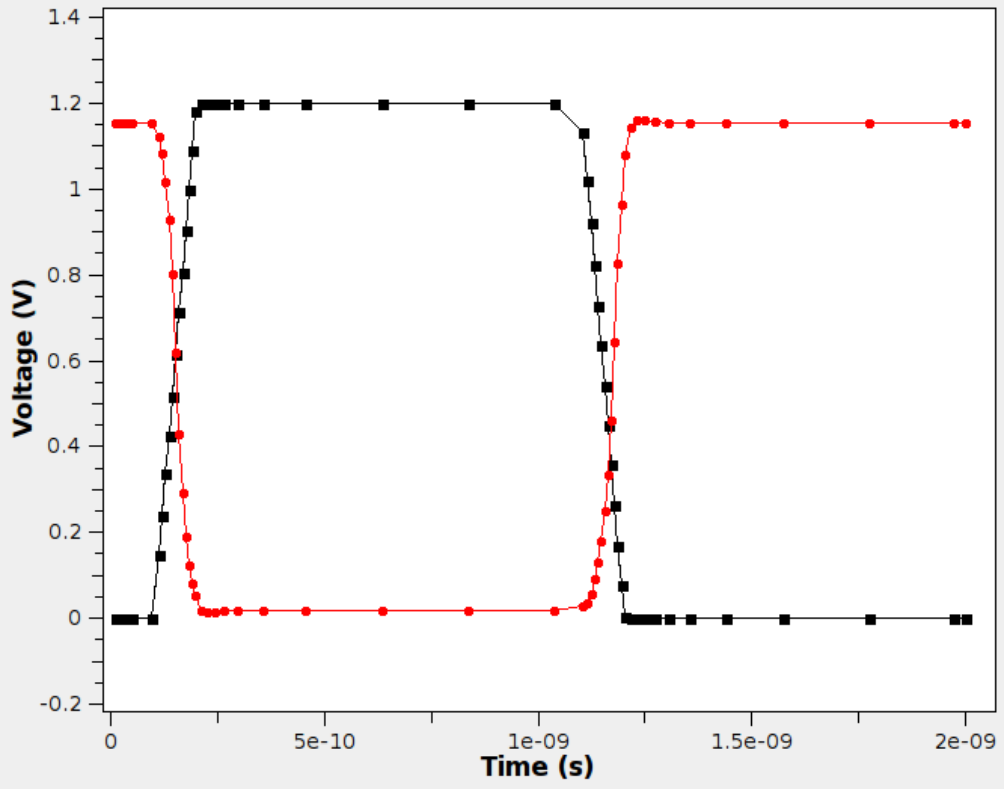



**Figure 1.10** Switching Waveform of the inverter.

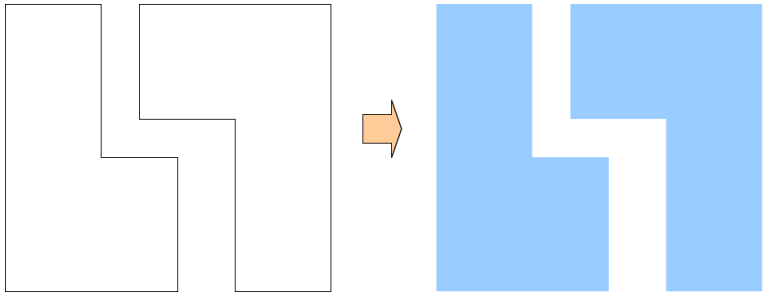
CHAPTER 2

# Features of Gds2Mesh

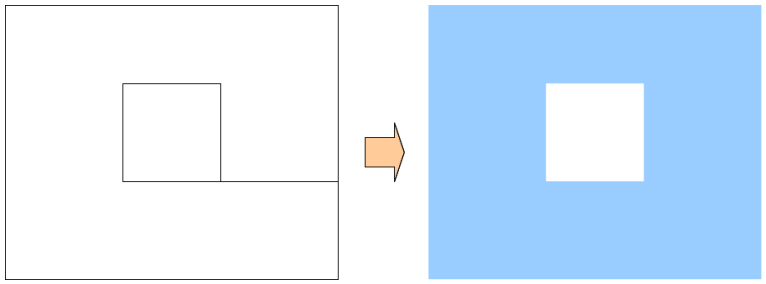
## 2D Graphs

The class `SimplePolygonGraph` provides facilities to create and manipulate 2D polygon graphs.

Graphs can be constructed from polygons, as shown in **Figure 2.1**, **p. 15**. Graphs can have holes, and can be constructed from polygons, too.



(a) A simple graph with two polygons



(b) A graph with one hole.

**Figure 2.1** Examples of graphs constructed from polygons.

Boolean operations of graphs are supported. Please see "**Class SimplePolygonGraph**", **p. 47** for documentation on the class.

Another operation for graph is offsetting, as shown in **Figure 2.2**, **p. 16**. Positive offset means expanding the graph, while negative offset mean shrinking. The edges of the offsetted graph are parallel to the corresponding edges of the original graph. If the graph contains holes, positively offsetting the graph will increase the area of the graph, by growing the outline while shrinking the hole.

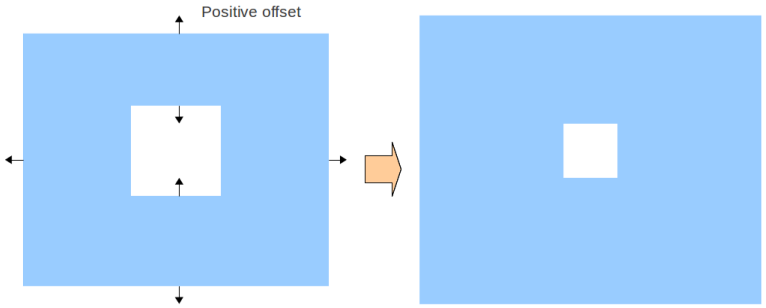



**Figure 2.2** Offsetting a graph.

# Create 3D Objects by Extrusion

The main mechanism to create 3D objects from 2D graphs in Gds2Mesh is by extrusion.

One starts from a 2D graph in the x-y plane, and raise it vertically in the z-direction. Side faces are then added to form a closed 3D object, as illustrated in **Figure 2.3**, **p. 17**.

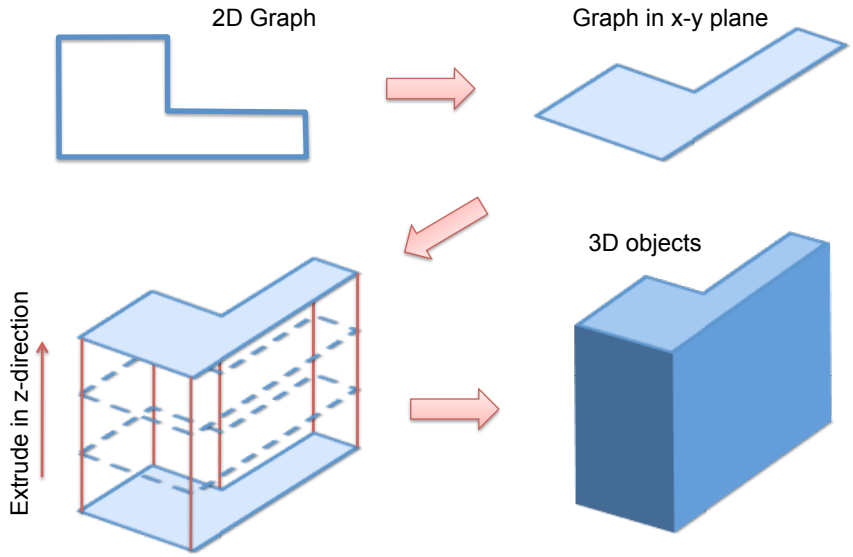



**Figure 2.3** Basic extrusion operation.

As one raise the graph in the z-direction, one can offset the graph. Side faces become slanted in this case, as shown in **Figure 2.4**, **p. 17**.

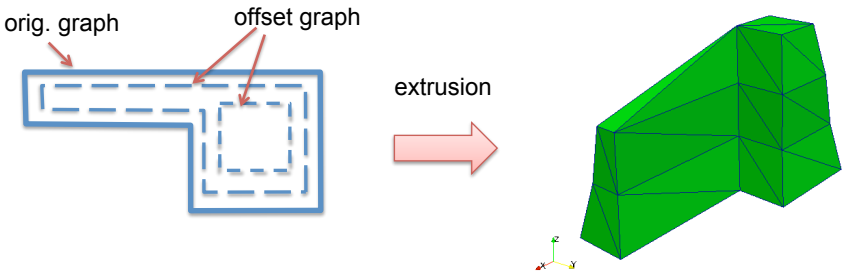



**Figure 2.4** Extrusion with offsetting.

**Figure 2.5**, **p. 18** illustrates building a hemisphere by extrusion with offsetting.

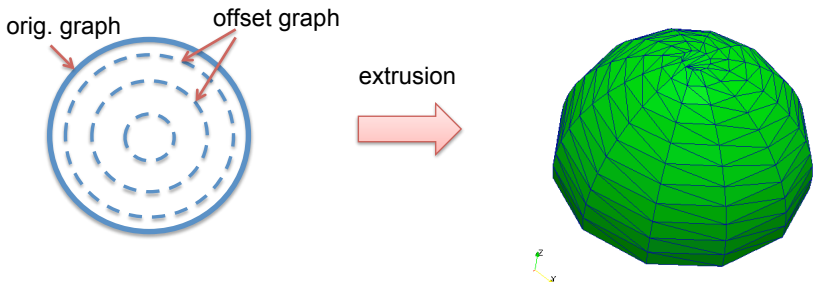



**Figure 2.5** Hemisphere built with extrusion.

Holes in 2D graphs become holes in the 3D objects after extrusion, as shown in **Figure 2.6, p. 18**.

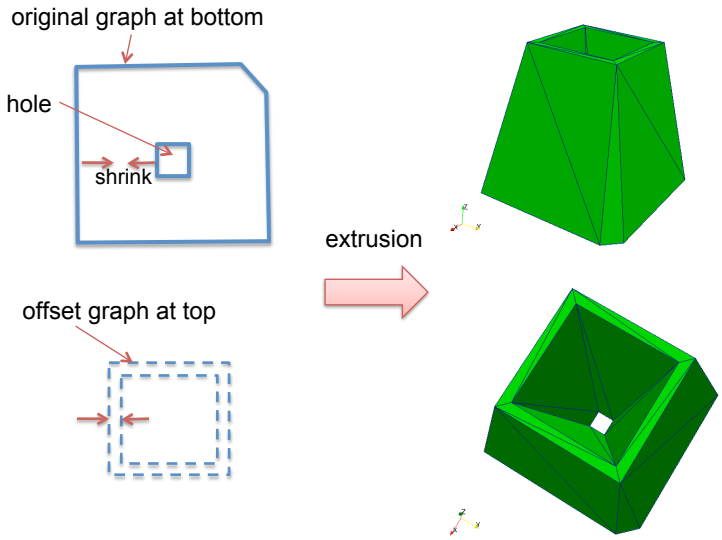



**Figure 2.6** Extrusion of graphs with holes.

One uses the `ExtrudedPolygonNG` class to creates 3D objects from extrusion, which is documented in "**Class ExtrudedPolygonNG**", **p. 57**.

# Doping Functions

Doping profiles can be introduced into the device structure with the `structure.add_profil` function, which emulates the ion implantation processes. One defines a mask window through which ion beams enters the device, as shown in **Figure 2.7**, **p. 19**.

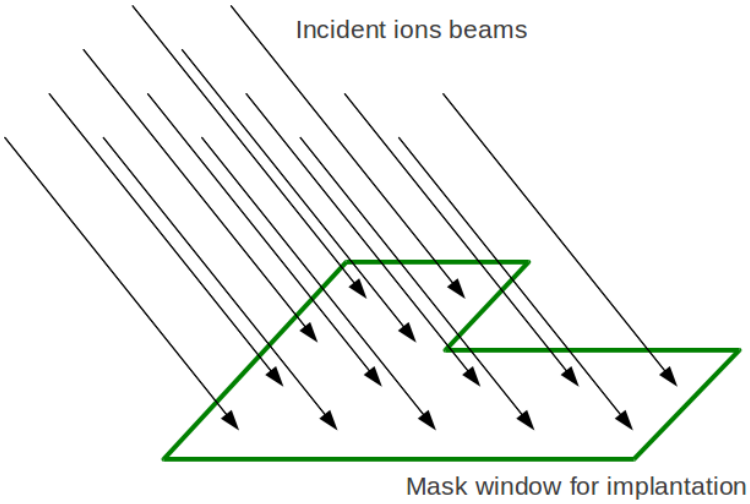



**Figure 2.7** Implantation through a mask window.

Ion implantation is emulated for each ion beam, in the range/lateral coordinates system illustrated in **Figure 2.8**, **p. 19**. The mask is always planar in the x-y plane, at coordinate $z = z_0$. The beams enters with inclination and azimuth angles θand ϕ, respectively.

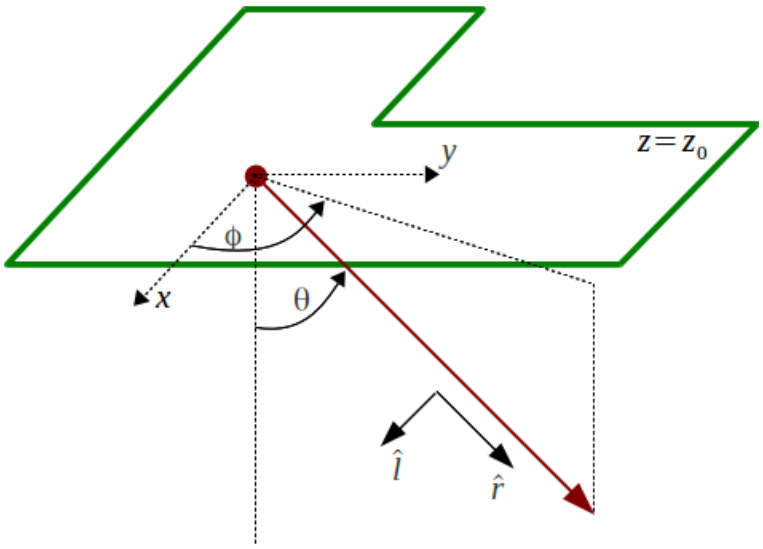



**Figure 2.8** Implantation coordinates, with reference plane $z = z_0$, range unit vector $\hat{r}$ and lateral unit vector $\hat{l}$.

Each ion beam contribute a doping profile depicted in **Figure 2.9**, **p. 20** to the device structure. The doping function has a lateral and a range component. The lateral component is a radially symmetrical gaussian distribution with characterisitic length $L_l$. The range component can be either a modified gaussian distribution (as in **Figure 2.9**, **p. 20**) or an erfc distribution, with characterisitic length $L_r$.

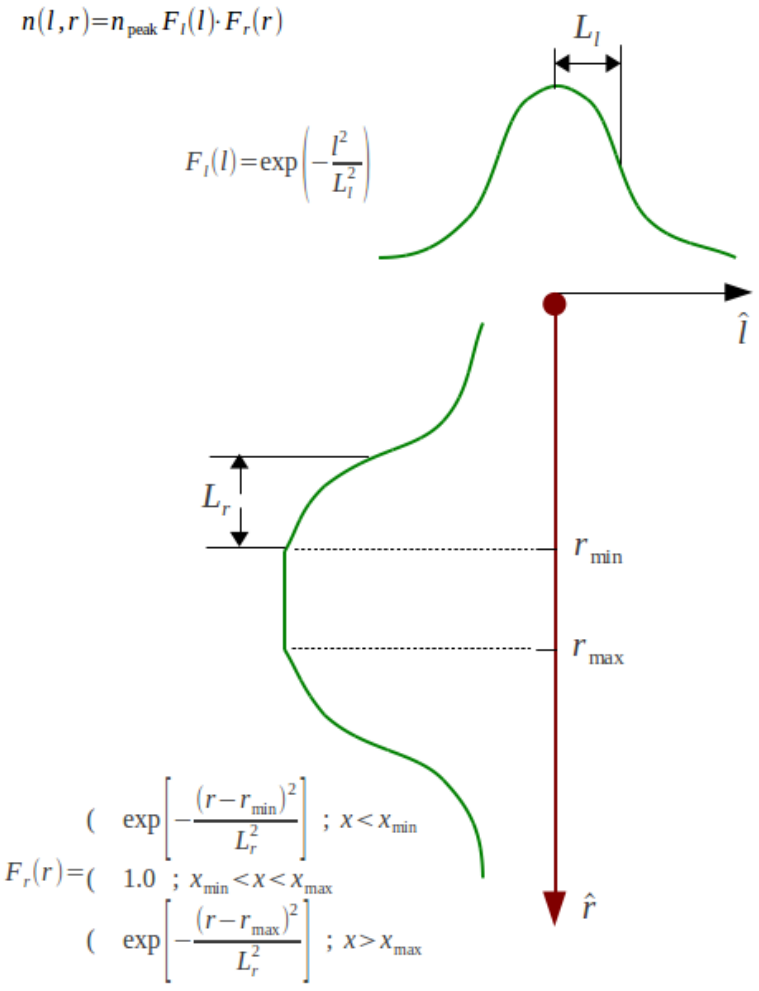



**Figure 2.9** Distribution function

As many ion beams go through the mask parallel to each other, the accumulated doping profile follows the erfc profile illustrated in **Figure 2.10**, **p. 21**. The edges of the mask window coincide with the width at half-maximum of the lateral distribution.

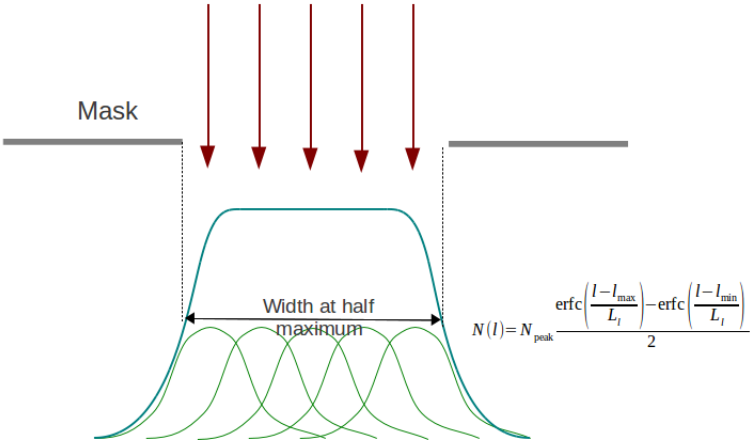



**Figure 2.10** Integral of doping function in the lateral direction.

CHAPTER 3

# Defining Process Rules

In this chapter, we describe aspects about creating a process rule for creating device models.

## Example Diode Process

We build a PN-junction diode in this section with a custom process rule. The structure of the diode is shown in **Figure 3.1**, **p. 23**.

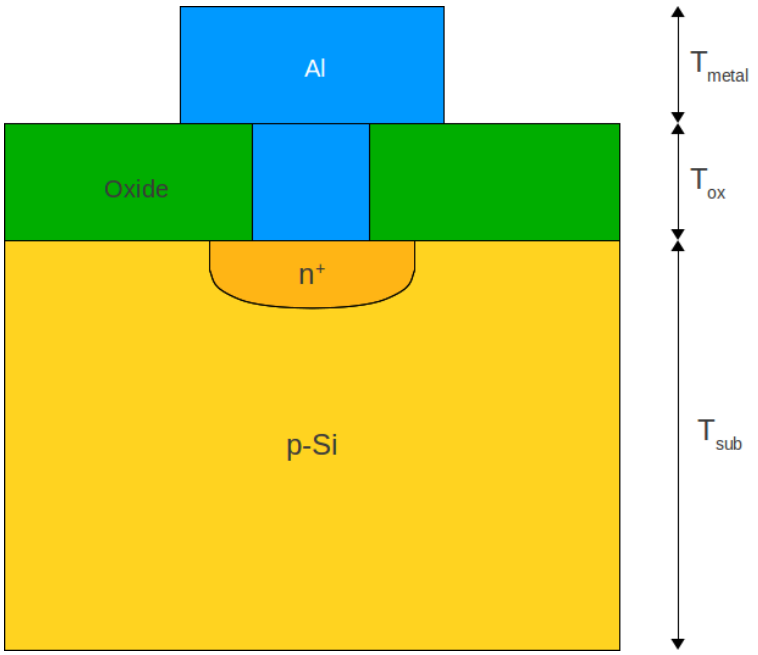



**Figure 3.1** Diagram of the PN-junction diode.

### Process Script

The process rule building this diodes is listed below. We shall go through the example line-by-line.

Program listing of the demo script.

```
__all__=['DiodeProcess']
```

```
from ProcessDesc import *

class DiodeParams(ParameterSet):
    '''Parameters for a demo diode process'''
    def __init__(self):
        super(DiodeParams, self).__init__()
        self.params = [
  ('Nsub',        1e16,   'P-type substrate doping (cm^-3)'),
  ('Ndiff',       1e20,   'Peak doping of the N+ diffusion (cm^-3)'),
  ('Rmax',        0.10,   'Rmax of the diffusion (um)'),
  ('Rmin',        0.10,   'Rmin of the diffusion (um)'),
  ('Ll',          0.07,   'Lateral char. length of the diffusion (um)'),
  ('Lr',          0.07,   'Vertical char. length of the diffusion (um)'),
  ('Tsub',        1.00,   'Substrate thickness (um)'),
  ('Tox',         0.30,   'Oxide passivation thickness (um)'),
  ('Tmetal',      0.20,   'Metal thickness (um)'),
  ('msz_sub',     0.20,   'Mesh size in substrate (um)'),
  ('msz_active',  0.05,   'Mesh size in diffusion region (um)'),
           ]

class DiodeProcess(ProcessBase):
    '''Demo diode process. We need masks: SUB ACTIVE METAL'''

    Params = DiodeParams

    def __init__(self, params):
        super(DiodeProcess, self).__init__(params)


    def buildDevice(self):
        params = self.params
        msz_sub, msz_active = params.getParams(['msz_sub', 'msz_active'])
        Tsub, Tox, Tmetal = params.getParams(['Tsub', 'Tox', 'Tmetal'])

        g_sub    = self.mask.getLayer('SUB')
        g_active = self.mask.getLayer('ACTIVE')
        g_metal  = self.mask.getLayer('METAL')

        # Substrate
        obj = Extrusion(g_sub, -Tsub, 0.)
        self.device.add_object(obj, 'sub', 'Si', '', 'bottom', msz_sub)

        # Metal plug
        obj = Extrusion(g_active, 0., Tox)
        self.device.add_object(obj, 'contact', 'Al', '', '', msz_sub)
```

```
# Metal                                                                    49
obj = Extrusion(g_metal, Tox, Tox+Tmetal)                                  50
self.device.add_object(obj, 'metal', 'Al', 'top', '', msz_sub)             51
                                                                           52
# oxide passivation                                                        53
obj = Extrusion(g_sub, 0.0, Tox)                                           54
self.device.set_fill_object(obj, "ox", "SiO2")                             55
                                                                           56
# doping                                                                   57
Nsub, Ndiff, Rmax, Rmin, Ll, Lr = \                                        58
    params.getParams(['Nsub', 'Ndiff', 'Rmax', 'Rmin', 'Ll', 'Lr'])        59
                                                                           60
prf = PlanarUniformProfile(g_sub, -Tsub, 0., "Acceptor", Nsub)             61
self.device.add_profile(prf)                                               62
                                                                           63
prf = PlanarAnalyticProfile(g_active, 0.0, Rmin, Rmax, \                   64
                            "Donor", Ndiff, Lr, Ll, \                      65
                            PlanarAnalyticProfile.GAUSSIAN)                66
self.device.add_profile(prf)                                               67
                                                                           68
# mesh size                                                                69
self.device.add_mesh_size_control(g_sub,  -0.1,  msz_active)               70
self.device.add_mesh_size_control(g_sub,   0.0,  msz_active)               71
```

Line 1 lists the process rule classes that will be exported by this script, and seen by gds2mesh.

In line 3, we import everything from the `ProcessDesc` module, which includes `ParameterSet` and `ProcessBase` used later in the script.

### Classes

A process rule typically consists of two classes. The first class is derived from `ParameterSet`(“**Class ParameterSet**”, **p. 64**), declaring process parameters (line 5-21).

The other class is derived from `ProcessBase` (“**Class ProcessBase**”, **p. 68**), and defines how the device structure should be built from the mask (line 23-71). The constructor method of the process class typically has an argument `params`, which contains the parameters.

Gds2mesh calls the `buildDevice()` method for building the device structure, which is the core of a process rule.

### Parameters

We then define the `DiodeParams` class, which is derived from the `ParameterSet` base class. In line 9-21, we declares the list of parameters that characterize this process. Each parameter is a tuple with three elements, the name, the default

value and a brief description. The parameters that determines layer thicknesses are shown in **Figure 3.1**, **p. 23**.

Line 33-35 demonstrates how parameters are used in the process class. The process class has an attribute `params`, and one can use its `getParams()` method to read a list of parameters.

### Mask Graphs

The process class also has an attribute `mask`, and Gds2mesh is responsible for setting it to a `MaskBase` ("**Class MaskBase**", **p. 65**) object that contains a mask set.

A mask set consists of several layers, referenced by names. In lines 37-39, we build a `SimplePolygonGraph` object ("**Class SimplePolygonGraph**", **p. 47**) from each of the three mask layers.

### 3D Objects

We can construct a 3D object by extruding from the graphs, as the substrate object in line 42. The z-coordinates of the bottom and top surfaces of the substrate object are `-Tsub` and 0.0, respectively.

For more options of extrusion, please see "**Class ExtrudedPolygonNG**", **p. 57** for details.

### Device Regions

In line 43, we add the 3D object to the device structure as a region, named *`sub`*. The material of this region is `silicon`. The top surface is not used; while the bottom surface is labeled *`bottom`*, and will be used as a boundary condition in device simulation, as we shall see in the next simulation.

The metal plug and metal contact regions are similarly defined in lines 45-51, and another boundary *`top`* is declared.

We then fill the empty spaces with oxide with the `set_fill_object()` method.

### Doping Profiles

In line 61, we define a uniform acceptor doping profile within the graph `g_sub` between the heights `-Tsub` and 0.0, with concentration `Nsub`. The profile is then added to the device.

In lines 64-66, we defines a Gaussian profile for the N+ diffusion of the junction. The active mask layer is the implant window, the range, characteristics lengths and peak concentrations come from parameters.

### Mesh-Size Constraints

In lines 69-71, we apply mesh size constraints at the depths -0.1 and 0.0, under the graph `g_sub`, and the length constraint comes from the parameter `msz_active`.

### Generating Mesh

This concludes the process rule script, and one can load this script in the Gds2mesh GUI. The process *`DiodeProcess`* and parameter *`DiodeParams`* will appear in the list of available processes and parameters.

This process rule requires a mask set with three layers, *SUB*, *ACTIVE* and *METAL*. An example mask is located at `examples/simple_mask.pkl`.

## Device Simulation

After generating the `.tif3d` mesh file for the diode with the process rule defined in the previous section. The input deck for simulating the diode is listed below.

Program listing of the Genius input deck for the diode.

```
GLOBAL T=300 Doping=1e19 Resistive=true

IMPORT TIF3D=diode.tif3d

BOUNDARY Id=top    Type=SolderPad
BOUNDARY Id=bottom Type=Ohmic

METHOD Type=Poisson NS=Newton LS=BCGS PC=ASM
SOLVE

METHOD Type=DDML1 NS=Newton LS=BCGS PC=ASM
SOLVE Type=Equ

METHOD Type=DDML1 NS=Newton LS=BCGS PC=ASM
SOLVE Type=DCSweep Vscan=top Vstart=0 Vstop=-0.8 Vstep=-0.05
```

One can import the device geometry and mesh from the `.tif3d` file. We enable the option `ResistiveMetal` in the `GLOBAL` command, since we need hope to treat metal regions as realistic metals with resistivity.

The two boundaries, *top* and *bottom*, declared in process rule, are defined as SolderPad and Ohmic boundaries for device simulation, respectively, in the `.inp` deck.

# Gdsii Layer Map

Mask set in Gds2mesh can be generated in several ways, programmatically or from GDSII files. Layers in GDSII files are identified by numeric IDs. However, the process rules of Gds2mesh refer to mask layers by their names. One therefore has to map the numeric GDSII layer number to the logical names so that they can be used in Gds2mesh.

A mapping script, for the layer definition of MOSIS CMOS process, is shown below.

Program listing of the MOSIS mask map definition.

```
__all__=['MosisCMOSMask']

from ProcessDesc import *

class MosisCMOSMask(GdsiiMask):
    map = {
            'N_WELL':           42,
            'P_WELL':           41,
            'ACTIVE':           43,
            'POLY':             46,
            'N_PLUS_SELECT':    45,
            'P_PLUS_SELECT':    44,
            'CONTACT':          25,
            'METAL1':           49,
            'ROUTE_PORT':       24          # Port for routing
        }

    def __init__(self, fname, params=GdsiiMask.Params(), \
                 top_level_struct=None):
        super(MosisCMOSMask, self).__init__(fname, \
            params=params, top_level_struct=top_level_struct)

    def getLayerList(self):
        return [
            'BBOX',
            ('N_WELL',          0x80ff8d, 0),
            ('P_WELL',          0x80a8ff, 0),
            ('ACTIVE',          0x008000, 5),
            ('POLY',            0xff0000, 4),
            ('N_PLUS_SELECT',   0x01ff6b, 12),
            ('P_PLUS_SELECT',   0xfbe328, 13),
            ('CONTACT',         0x0080ff, 1),
```

```
        ('METAL1',          0x0000ff, 12),
        'ROUTE_PORT']

def getLayer(self, layer):
    if layer=='BBOX':
        return self.getBoundbox()
    else:
        return super(MosisCMOSMask, self).getLayer(layer)
```

The class attribute, `map`, contains the actual mapping from logical name to the GDSII numeric layer number.

The `getLayerList()` method provides the Gds2mesh GUI information about which layers are to be displayed in the mask view. If a layer is defined as a tuple, the second and third elements in the tuple defines the color and fill pattern in which the layer should be displayed.

We extend the `getLayer()` method from the base class, to return the boundbox graph, to be displayed in the Gds2mesh GUI.

# Generic CMOS Process Rules

We need to explain the GenericCMOS script in more detail.

The file is located at `/opt/cogenda/1.7.3/gds2mesh/lib/GenericCMOS.py`

The process rule in this example designed for the GDSII mask files in the open-source cell library developed by Graham Petley (http://www.vlsitechnology.org/), with MOSIS design rules and layer map. Each circuit cell has two wires labeled *vdd* and *vss*, and several possible IO pads for each input and output ports, so that place&route software can connect this cell to the circuit. This example CMOS process rule make use of these labels for IO pads and power pads, as described in section "**Power and IO Pads**", **p. 38**. And the layout of inverter is shown in **Figure 3.2**, **p. 31**.

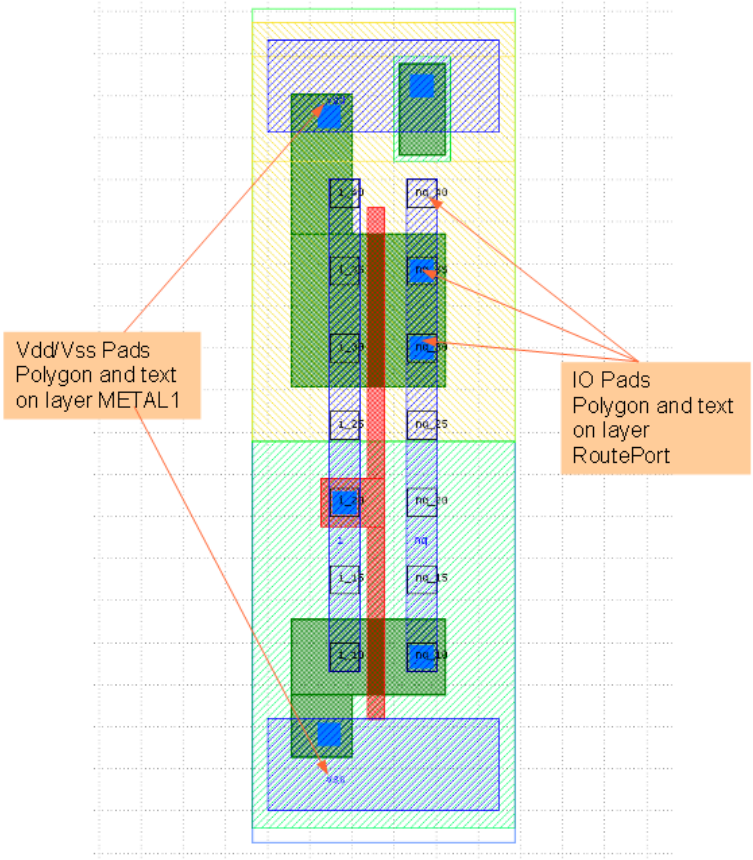



**Figure 3.2** Mask of an inverter circuit

## Overview of CMOS Process

Every process rule in Gds2mesh is a Python class inherited from the `ProcessBase`

class, is shown below.

ProcessBase Class

```
class ProcessBase(object):

    Params = ParameterSet

    def __init__(self, params):
        self.params = params
        self.mask   = None
        self.device = gds2mesh.Structure()
        self.refine = True

    def setMask(self, mask):
        self.mask = mask

    def buildDevice(self):
        pass

    def doMesh(self, quality=0.3):
        self.device.do_mesh(quality)

    def doSurfaceMesh(self, quality=0.3):
        self.device.do_surface_mesh(quality)

    def save(self, fname, ftype=None):
        ...
```

The device structure, as a set of polyhedra, should be stored in `self.device`. It is initially empty, and is supposed to be populated within `builddevice()` method. In the base class, `builddevice()` is empty.

The `CMOSProcess` class extends `ProcessBase`, and defines the of a CMOS process. The general structure of a device produced by this process rule is shown in **Figure 3.3, p. 33**. The thicknesses of layers are defined by process parameters `TSTI`, `Tox`, `Tpoly`, etc.

The `buildDevice()` method is extended in the `CMOSProcess` class, as we see in next listing. Since a CMOS process is rather complex, we break down `buildDevice()` to several steps, e.g. `buildActive()`, `buildPoly()`, `placeChannelDop` etc. These steps are defined in ProcessBase, and some of them extended further in the specific processes, such as the 0.13um process CMOS013. In the followings, we explain the methods in GenericCMOS one by one.

Snippet in CMOSProcess class

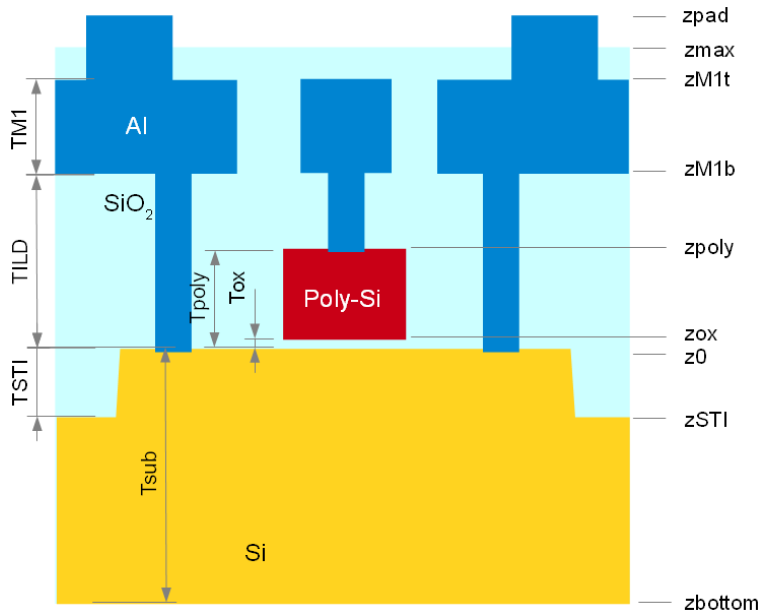



**Figure 3.3** Layers thicknesses and z-coordinates in the CMOS process

```
class CMOSProcess(ProcessBase):
    '''Generic Deep Submicron CMOS Process'''
    Params = CMOSParams
    ...
    def buildDevice(self):
        if self.mask==None:
            raise ValueError
        self.buildSubstrate()
        self.buildActive()
        self.buildGateOxide()
        self.buildPoly()
        self.buildContact()
        self.buildMetal1()
        self.buildPowerPad()
        self.buildIOPad()
        self.buildFillOxide()

        self.placeWaferDoping()
        self.placeWellDoping()
        self.placeChannelDoping()
        self.placePocketDoping()
        self.placeSDDoping()

        if self.refine:
            self.meshSizeControl()
```

# Building Geometry Objects

## Constructor

The constructor method of `GenericCMOS` is shown in the next Listing. In this method, we calculate the z-coordinates of the layers (see also **Figure 3.3, p. 33**), and save them in object variables such as `self.z0`, `self.zpoly`, etc.

The material used in some of the regions are defined as well.

Constructor of the CMOSProcess class

```
def __init__(self, params):                                                    1
    super(CMOSProcess, self).__init__(params)                                  2
    self.IOPadList = None  # default=None. We build every IO pad \             3
       in this case                                                            4
                                                                               5
    Tsub, TSTI, Tox, Tpoly, TILD, TM1, lmd = self.params.getParams([ \         6
      'Tsub','TSTI', 'Tox', 'Tpoly', 'TILD', 'TM1', 'lmd'])                    7
    self.z0 = 0.0                                                              8
    self.zbottom = self.z0 - Tsub                                              9
    self.zSTI    = self.z0 - TSTI                                             10
    self.zox     = self.z0 + Tox                                              11
    self.zpoly   = self.z0 + Tpoly                                            12
    self.zM1b    = self.z0 + TILD                                             13
    self.zM1t    = self.z0 + TILD + TM1                                       14
    self.zmax    = self.zM1t + 2*lmd  # top of oxide                          15
    self.zpad    = self.zmax + 0*lmd  # top of pad                            16
                                                                              17
    self.materials = {'npoly':           'NPoly',                             18
                      'ppoly':           'PPoly',                             19
                      'active_contact':  'Al',                                20
                      'poly_contact':    'Al',                                21
                      'metal1':          'Al',}                               22
```

## Substrate and Active

The silicon substrate below the bottom of STI isolation, i.e. zbottom > z > zSTI, is referred to as the `sub` region. In `buildSubstrate()`, we obtain the bound-box of the mask, and through extrusion, builds the substrate object, is shown in below. The bottom surface of `sub` region is defined as a boundary, also named `sub`. The maximum size of the mesh elements in substrate region is constrained by the `msz_sub` parameter.

In the `buildActive()` method, is shown in below, the active regions, defined by the `ACTIVE` layer in the masks, are built above the substrate.

buildSubstrate() method of CMOSProcess class

```
def buildSubstrate(self):
    if self.refine:
        msz_sub = self.params.getParams('msz_sub')
    else: msz_sub = 1e3
    g = self.mask.getBoundbox()
    obj = Extrusion(g, self.zbottom, self.zSTI)
    self.device.add_object(obj, "sub", "Si", "", "sub", msz_sub)
       # bottom surface with label
```

The active region is not built from simple extrusion. In order to model the corner rounding at the edges of STI, a set of offsets are defined as process parameters, and used during extrusion. The offset/height parameters are illustrated in **Figure 3.4, p. 36**.

buildActive() method of CMOSProcess class

```
def buildActive(self):
    if self.refine:
        msz_active = self.params.getParams('msz_active')
    else: msz_active = 1e3
    Ro0, Rh1, Ro1, Rh2, Ro2 = self.params.getParams(['Ro0_STI',\
       'Rh1_STI', 'Ro1_STI', 'Rh2_STI', 'Ro2_STI' ])
    TSTI, rsl_STI = self.params.getParams(['TSTI', 'rsl_STI'])

    heights = [self.zSTI, self.z0-Rh2, self.z0-Rh1, self.z0]
    offsets = [TSTI*rsl_STI, -Ro2, -Ro1, -Ro0]
    obj = Extrusion(self.mask.getLayer('ACTIVE'), heights, offsets)
    self.device.add_object(obj, "active", "Si", "", "", msz_active)
```

There are probably several active polygons in the mask. Accordingly, gds2mesh builds several active regions, named as `active_0`, `active_1`, `active_2`, etc.

## Gate Regions

In this example, gate oxide, STI fill-in, and passivation oxide are not distinguished. They are assigned the same SiO2 material, and constructed last in the `buildFillOxide()` method in section `Error: Reference source not found.`

Next, we build the polysilicon gate regions from the POLY mask layer with the `buildPoly()` method is shown below.

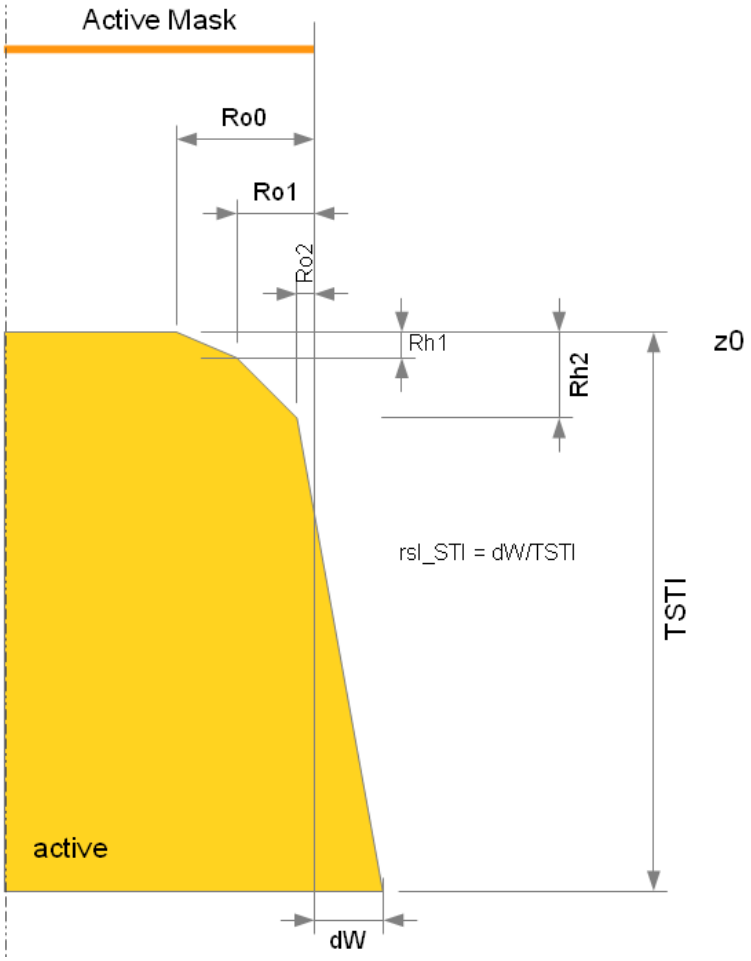



**Figure 3.4** The active region is defined by the active mask. A set of Offset and height parameters are used for defining the corner-rounding of the active regions

buildPoly() method of CMOSProcess class

```
def buildGateOxide(self):
    pass

def buildPoly(self):
    lmd = self.params.getParams('lmd')
    if self.refine: msz=lmd
    else:           msz=1e3
    g_poly   = self.mask.getLayer('POLY')
    g_p_plus = self.mask.getLayer('P_PLUS_SELECT')

    g_p_poly = Polygon.intersect(g_poly, g_p_plus)
    g_n_poly = Polygon.subtract(g_poly, g_p_plus)

    npoly = Extrusion(g_n_poly, self.zox, self.zpoly)
    self.device.add_object(npoly, 'npoly', self.materials['npoly'],\
```

```
        '', '', msz)

    ppoly = Extrusion(g_p_poly, self.zox, self.zpoly)
    self.device.add_object(ppoly, 'ppoly', self.materials['ppoly'], \
       '', '', msz)
```

In Genius simulator, there are two ways to model polysilicon gate material. In the first approach, one treats the gate as a metal with work-function that corresponds to the heavily doped n-type or p-type polysilicon, and assign material `npoly` or `ppoly` to the region. By doing this, one ignores poly-depletion effects. Doping concentration in these regions are ignored. Alternatively, one can assign the Polysilicon material to the region, and treat it as a semiconductor. In this approach, work-function is determined by doping concentration. In this example, the first approach is used. The portion of gate region within P_PLUS_SELECT mask is assigned `ppoly`, the rest is assigned `npoly`.

In the design rule, the minimum width of POLY is usually $2\lambda$. The mesh-size constraint is $\lambda$ here, making sure that there are some internal mesh points in the poly regions. As the active regions, the poly regions are named `n_poly_0`, `n_poly_1`, `p_poly_0`, etc.

## Contact Holes

There are two types of contact holes, both derived from the CONTACT layer in the mask. When a CONTACT polygon is drawn within a POLY polygon, the contact hole is made to the poly region, i.e. between height `zpoly` and `zM1b`. On the other hand, when a CONTACT polygon is drawn within ACTIVE but outside POLY, the contact hole is made to the active region, i.e. between height `z0` and `zM1b`. In the `buildContact()` method in below, `g_poly_contact` and `g_active_contact` graphs are calculated with the rule described above.

buildContact() method of CMOSProcess class

```
def buildContact(self):
    lmd = self.params.getParams('lmd')
    if self.refine: msz=lmd
    else:           msz=1e3
    off_ply_cnt, off_act_cnt = self.params.getParams(['off_ply_cnt', 'off_act_cnt

    g_poly    = self.mask.getLayer('POLY')
    g_contact = self.mask.getLayer('CONTACT')

    g_poly_contact   = Polygon.intersect(g_contact, g_poly)
    g_active_contact = Polygon.subtract(g_contact, g_poly)
```

```
    heights = [self.z0, self.zM1b]
    offsets = [off_act_cnt, off_act_cnt]
    active_contact = Extrusion(g_active_contact, heights, offsets)
    self.device.add_object(active_contact, 'active_contact',\
                           self.materials['active_contact'],  '', '', msz)

    heights = [self.zpoly, self.zM1b]
    offsets = [off_ply_cnt, off_ply_cnt]
    poly_contact = Extrusion(g_poly_contact, heights, offsets)
    self.device.add_object(poly_contact, 'poly_contact',\
                           self.materials['poly_contact'],  '', '', msz)
```

## Metal 1

The `buildMetal1()` method in below is rather straight-forward.

buildMetal1() method of CMOSProcess class

```
def buildMetal1(self):
    lmd = self.params.getParams('lmd')
    if self.refine: msz=1.5*lmd
    else:           msz=1e3
    obj = Extrusion(self.mask.getLayer('METAL1'), self.zM1b, self.zM1t)
    self.device.add_object(obj,     'metal',\
                           self.materials["metal1"],  '', '', msz)
```

## Power and IO Pads

In `buildIOPad()`, the list of pad labels on the `ROUTE_PORT` layer is first obtained, and for each label, the polygon bounding the label is obtained, and a pad object is extruded from the polygon. The top of the pad is marked as a boundary using the label as its identifier.

buildIOPad() method of CMOSProcess class

```
def getIOPadList(self):
    if self.IOPadList==None:
        return self.mask.getLabels('ROUTE_PORT')
    else:
      return self.IOPadList

def buildIOPad(self):
    padList = list(set(self.getIOPadList())) ## remove duplicates
    lmd = self.params.getParams('lmd')
```

```
    if self.refine: msz=lmd                                          10
    else:           msz=1e3                                          11
    for pad in padList:                                              12
        g = self.mask.getPad('ROUTE_PORT', pad)                      13
        if g==None:                                                  14
          print 'IO pad "%s" not found' % pad                        15
          raise ValueError                                           16
        obj = Extrusion(g, self.zM1t, self.zpad)                     17
        self.device.add_object(obj,  pad, 'Al', pad, '', msz)        18
```

## Fill-in Oxide

The `buildFillOxide()` method fills the empty space in the device structure with oxide material is shown in below. This includes the STI isolation, gate insulator, ILD, IMD and passivation oxide.

In `buildFillOxide()`, an extrusion object is first built with the mask bound-box, from the bottom of STI to the top of the device structure.

The `set_fill_object()` method is then called, assigning the SiO2 material. Internally, `set_fill_object()` takes the extrusion obj, subtract from it all objects previously added to the device structure, i.e. active, poly, contact regions, and finally add the resultant object to the device.

buildFillOxide() method of CMOSProcess class

```
def buildFillOxide(self):                                            1
    if self.refine:                                                  2
        msz_ox = self.params.getParams('msz_ox')                     3
    else: msz_ox = 1e3                                               4
    obj = Extrusion(self.mask.getBoundbox(), self.zSTI, self.zmax)   5
    self.device.set_fill_object(obj, "ox", "SiO2", msz_ox)           6
```

# Doping Profiles

## Substrate Doping

Substrate doping is applied to the entire substrate and all active regions, it is shown in below. If the Nsub parameter is positive, p-type doping is used. Otherwise, n-type doping is used.

placeWaferDoping() method of CMOSProcess class

```
def placeWaferDoping(self):
    Nsub = self.params.getParams('Nsub')
    # substrate doping
    g_sub = Polygon.offsetted(self.mask.getBoundbox(), 0.1)
    if Nsub>0:
        s = "Acceptor"
    else:
        Nsub *= -1
        s = "Donor"
    self.device.add_profile(PlanarUniformProfile(g_sub,\
       self.zbottom, self.z0, s, Nsub))
```

## Well Doping

P-well doping and N-well doping are applied to regions enclosed by P-WELL and N-WELL mask, respectively, as shown in below. These are Guassian doping profiles, with reference plane z=z0, i.e. at the surface of silicon active regions. The well doping is usually deep enough that it penetrates STI isolation and enters the substrate.

In some designs, only N-WELL polygons are drawn, and the P-WELL polygons are implicit. In this example, the inference of implicit P-WELL polygons should be handled by the mask class.

placeWellDoping() method of CMOSProcess class

```
def placeWellDoping(self):
    g_well_n = self.mask.getLayer('P_WELL')
    g_well_p = self.mask.getLayer('N_WELL')

    N_n, Rmax_n, Rmin_n, Ll_n, Lr_n = self.params.getParams(['Nwel_n',\
       'Rmax_wel_n', 'Rmin_wel_n', 'Ll_wel_n', 'Lr_wel_n'])
    self.device.add_profile(PlanarAnalyticProfile(g_well_n, self.z0,\
       Rmin_n, Rmax_n, "Acceptor", N_n, Lr_n, Ll_n,\
```

```
        PlanarAnalyticProfile.GAUSSIAN))

    N_p, Rmax_p, Rmin_p, Ll_p, Lr_p = self.params.getParams(['Nwel_p',\
        'Rmax_wel_p', 'Rmin_wel_p', 'Ll_wel_p', 'Lr_wel_p'])
    self.device.add_profile(PlanarAnalyticProfile(g_well_p, self.z0,\
        Rmin_p, Rmax_p, "Donor", N_p, Lr_p, Ll_p,\
        PlanarAnalyticProfile.GAUSSIAN))
```

## Channel Doping

Channel doping profiles are also defined by the well mask, and are similarly Gaussian profiles.

placeChannelDoping() method of CMOSProcess class

```
def placeChannelDoping(self):
    g_well_n = self.mask.getLayer('P_WELL')
    g_well_p = self.mask.getLayer('N_WELL')

    N_n, Rmax_n, Rmin_n, Ll_n, Lr_n = self.params.getParams(['Nchn_n',\
        'Rmax_chn_n', 'Rmin_chn_n', 'Ll_chn_n', 'Lr_chn_n'])
    self.device.add_profile(PlanarAnalyticProfile(g_well_n, self.z0,\
        Rmin_n, Rmax_n, "Acceptor", N_n, Lr_n, Ll_n,\
        PlanarAnalyticProfile.GAUSSIAN))

    N_p, Rmax_p, Rmin_p, Ll_p, Lr_p = self.params.getParams(['Nchn_p',\
        'Rmax_chn_p', 'Rmin_chn_p', 'Ll_chn_p', 'Lr_chn_p'])
    self.device.add_profile(PlanarAnalyticProfile(g_well_p, self.z0,\
        Rmin_p, Rmax_p, "Donor", N_p, Lr_p, Ll_p,\
        PlanarAnalyticProfile.GAUSSIAN))
```

## Pocket Doping

the `placePocketDoping()` method for pocket doping profile is shown in below.

placePocketDoping() method of CMOSProcess class

```
def placePocketDoping(self):
    off_pkt = self.params.getParams('off_pkt')
    g_well_n = self.mask.getLayer('P_WELL')
    g_well_p = self.mask.getLayer('N_WELL')
    g_poly   = self.mask.getLayer('POLY')
    g_off_pkt = Polygon.offsetted(g_poly, off_pkt)

```

```
        g_pkt_n = Polygon.subtract(g_well_n, g_off_pkt)                          8
        g_pkt_p = Polygon.subtract(g_well_p, g_off_pkt)                          9
                                                                                 10
        theta_pkt = self.params.getParams('theta_pkt')                           11
        N_n, Rmax_n, Rmin_n, Ll_n, Lr_n = self.params.getParams(['Npkt_n',\      12
            'Rmax_pkt_n', 'Rmin_pkt_n', 'Ll_pkt_n', 'Lr_pkt_n'])                 13
        N_n *= 0.25 # four implant at different phi                              14
        for phi in [45, 135, 225, 315]:                                          15
            self.device.add_profile(PlanarAnalyticProfile(g_pkt_n, \             16
                self.z0, Rmin_n, Rmax_n, theta_pkt, phi, "Acceptor",\            17
                N_n, Lr_n, Ll_n, PlanarAnalyticProfile.GAUSSIAN))                18
                                                                                 19
        N_p, Rmax_p, Rmin_p, Ll_p, Lr_p = self.params.getParams(['Npkt_p',\      20
            'Rmax_pkt_p', 'Rmin_pkt_p', 'Ll_pkt_p', 'Lr_pkt_p'])                 21
        N_p *= 0.25 # four implant at different phi                              22
        for phi in [45, 135, 225, 315]:                                          23
            self.device.add_profile(PlanarAnalyticProfile(g_pkt_p,\              24
               self.z0, Rmin_p, Rmax_p, theta_pkt, phi, "Donor", \               25
               N_p, Lr_p, Ll_p, PlanarAnalyticProfile.GAUSSIAN))                 26
```

In this example, the regions where pocket doping are applied are calculated from the `P_WELL`, `N_WELL` and `POLY` layers. More specifically, `POLY` graph is first expanded by `off_pkt`, to account for the offset spacer at both sides of the gate poly (line 15). It then subtract `P_WELL` by the offsetted poly graph (line 17), and apply the pocket doping for nMOSFET in this area (line 24). The pocket doping area for pMOSFET is calculated similarly. Since large angle implant is used, four implants with different rotation are used, each with a quarter of the dose.

## Source Drain Doping

Lastly, the source/drain doping profile are defined in the `placeSDDoping()` method, as shown in below. The nMOSFET source drain extension is placed in the area `g_sde_n`, which is calculated in line 8 as

```
SDE_n=N_PLUS-POLY
```

placeSDDoping() method of CMOSProcess class

```
def placeSDDoping(self):                                                         1
    off_spc = self.params.getParams('off_spc')                                   2
    g_n_plus  = self.mask.getLayer('N_PLUS_SELECT')                              3
    g_p_plus  = self.mask.getLayer('P_PLUS_SELECT')                              4
    g_poly    = self.mask.getLayer('POLY')                                       5
    g_off_spc = Polygon.offsetted(g_poly, off_spc)                               6
                                                                                 7
```

```
g_sde_n = Polygon.subtract(g_n_plus, g_poly)                              8
g_sde_p = Polygon.subtract(g_p_plus, g_poly)                              9
g_sd_n  = Polygon.subtract(g_n_plus, g_off_spc)                           10
g_sd_p  = Polygon.subtract(g_p_plus, g_off_spc)                           11
                                                                          12
# s/d extension                                                           13
N_n, Rmax_n, Rmin_n, Ll_n, Lr_n = self.params.getParams(['Nsde_n',\       14
   'Rmax_sde_n', 'Rmin_sde_n', 'Ll_sde_n', 'Lr_sde_n'])                   15
self.device.add_profile(PlanarAnalyticProfile(g_sde_n, self.z0, \         16
   Rmin_n, Rmax_n, "Donor", N_n, Lr_n, Ll_n, \                            17
   PlanarAnalyticProfile.GAUSSIAN))                                       18
                                                                          19
N_p, Rmax_p, Rmin_p, Ll_p, Lr_p = self.params.getParams(['Nsde_p',\       20
   'Rmax_sde_p', 'Rmin_sde_p', 'Ll_sde_p', 'Lr_sde_p'])                   21
self.device.add_profile(PlanarAnalyticProfile(g_sde_p, self.z0, \         22
   Rmin_p, Rmax_p, "Acceptor", N_p, Lr_p, Ll_p, \                         23
   PlanarAnalyticProfile.GAUSSIAN))                                       24
                                                                          25
# deep s/d                                                                26
N_n, Rmax_n, Rmin_n, Ll_n, Lr_n = self.params.getParams(['Nsd_n', \       27
    'Rmax_sd_n', 'Rmin_sd_n', 'Ll_sd_n', 'Lr_sd_n'])                      28
self.device.add_profile(PlanarAnalyticProfile(g_sd_n, self.z0, \          29
    Rmin_n, Rmax_n, "Donor", N_n, Lr_n, Ll_n, \                           30
    PlanarAnalyticProfile.GAUSSIAN))                                      31
                                                                          32
N_p, Rmax_p, Rmin_p, Ll_p, Lr_p = self.params.getParams(['Nsd_p',\        33
    'Rmax_sd_p', 'Rmin_sd_p', 'Ll_sd_p', 'Lr_sd_p'])                      34
self.device.add_profile(PlanarAnalyticProfile(g_sd_p, self.z0, \          35
    Rmin_p, Rmax_p, "Acceptor", N_p, Lr_p, Ll_p, \                        36
    PlanarAnalyticProfile.GAUSSIAN))                                      37
```

On the other hand the deep source drain is placed in the area `g_sd` (line 10), which is offsetted by the spacer as

```
SD_n=N_PLUS-POLY_expanded_off_spc
```

## Mesh Size Control

The `meshSizeControl()` method in the base class `CMOSProcess` is empty, which means it is left to the sub-classes to define how mesh should be refined.

The `CMOS013` class in `lib/CGD013.py` extends `CMOSProcess`, primarily the `meshSizeControl()` method, as shown in below. It derives a few graphs from the mask layers in lines 12-17, as illustrated in **Figure 3.5, p. 44**. With these graphs defined, the mesh size in various parts in the device is constrained as below.

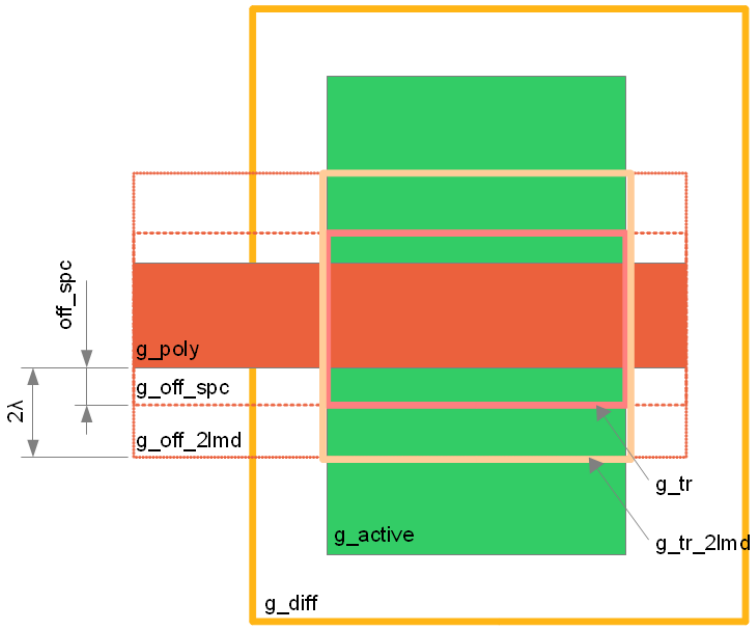



**Figure 3.5** Derived graph in meshSizeControl()

- The graph `g_tr` bounds the channel of transistor, in which area and near the surface, the mesh is the finest, constrained by `msz_surf` (lines 39-40).
- Deeper in the channel area (`g_tr`) is the MOS depletion region, where the mesh gradually gets coarser to `msz_depl` (lines 33-38).
- The graph `g_diff` includes all PN junction formed by diffusion, i.e. `N_PLUS` in `p_WELL` or `P_PLUS` in `N_WELL`. In this area, mesh size is constrained around the junction depth (lines 27-32).
- Similarly, the mesh size in wells is constrained by `msz_well`.

meshSizeControl() method of CMOSProcess class

```
def meshSizeControl(self, external_file=None):                    1
    lmd     = self.params.getParams('lmd')                        2
    off_spc = self.params.getParams('off_spc')                    3
    off_pkt = self.params.getParams('off_pkt')                    4
                                                                  5
```

```
g_active = self.mask.getLayer('ACTIVE')
g_contact= self.mask.getLayer('CONTACT')
g_poly   = self.mask.getLayer('POLY')

g_n_plus= self.mask.getLayer('N_PLUS_SELECT')
g_p_plus= self.mask.getLayer('P_PLUS_SELECT')
g_diff  = Polygon.intersect(g_active, Polygon.add(g_n_plus, g_p_plus))

g_off_spc = Polygon.offsetted(g_poly, 1.3*off_spc)
g_off_2lmd = Polygon.offsetted(g_poly, 2.2*lmd)
g_tr      = Polygon.intersect(g_active, g_off_spc)
g_tr_2lmd = Polygon.intersect(g_active, g_off_2lmd)

g_active_contact = Polygon.subtract(g_contact, g_poly)

msz_active, msz_wel, msz_sd, msz_depl, msz_surf = \
    self.params.getParams(['msz_active', 'msz_wel',\
    'msz_sd', 'msz_depl', 'msz_surf'])
self.device.add_mesh_size_control(g_active, self.zSTI, \
   msz_active)
self.device.add_mesh_size_control(g_active, self.z0-0.20,  msz_wel)
self.device.add_mesh_size_control(g_active, self.z0-0.10,  msz_wel)
self.device.add_mesh_size_control(g_active, self.z0-0.01,  msz_wel)

self.device.add_mesh_size_control(g_diff,   self.z0-0.22,  msz_sd)
self.device.add_mesh_size_control(g_diff,   self.z0-0.18,  msz_sd)
self.device.add_mesh_size_control(g_diff,   self.z0-0.14,  msz_sd)
self.device.add_mesh_size_control(g_diff,   self.z0-0.10,  msz_sd)

self.device.add_mesh_size_control(g_tr,     self.z0-0.12, \
    0.5*(msz_depl+msz_sd))
self.device.add_mesh_size_control(g_tr,     self.z0-0.09,  msz_depl)
self.device.add_mesh_size_control(g_tr,     self.z0-0.06,  msz_depl)
self.device.add_mesh_size_control(g_tr,     self.z0-0.04,  msz_depl)
self.device.add_mesh_size_control(g_tr,     self.z0-0.02,  msz_depl)
self.device.add_mesh_size_control(g_tr,     self.z0-0.01, \
    0.5*msz_depl+0.5*msz_surf)
self.device.add_mesh_size_control(g_tr,     self.z0-7e-3,\
    0.3*msz_depl+0.7*msz_surf)
self.device.add_mesh_size_control(g_tr,     self.z0-4e-3,  msz_surf)
self.device.add_mesh_size_control(g_tr,     self.z0-1e-3,  msz_surf)

self.device.add_mesh_size_control(g_tr_2lmd, self.z0-1e-3,2*msz_surf)

self.device.add_mesh_size_control(g_active_contact, \
    self.z0-0.001, 0.2*lmd)
```

```
                                                                    52
        if not external_file==None:                                 53
            self.device.add_mesh_size_control(external_file)        54
```

CHAPTER 4

# API Reference

Gds2Mesh consists of libraries written in both C++ and in Python, and a uniform Python API is provided. This chapter describes this API.

## Module gds2mesh

The `gds2mesh` module is written in C++ and exposes an API in the Python language. It provides the following classes

### Class SimplePolygonGraph

The SimplePolygonGraph class describes a polygon graph containing several disjoint polygons, possibly with holes. The polygons must be simple and NOT self-intersecting.

#### Constructor `SimplePolygonGraph`

```
SimplePolygonGraph()
```

**Returns** An empty graph.

#### Constructor `fromPolygon`

```
fromPolygon(points)
```

**Arguments** `points` list of corner points of the polygon, each point is represented by a tuple `(x,y)`.

**Returns** A graph containing polygon (possibly with holes) constructed from the corner.

#### Factory Method `add`

```
add(graph_a, graph_b, simplify=True)
```

Calculate the union of two graphs and return the result as a new graph.

**Arguments**

`graph_a` graph, first operand, of the `SimplePolygonGraph` type.

`graph_b` graph, second operand, of the `SimplePolygonGraph` type.

`simplify` boolean, whether to simplify the resultant by removing redundant points.

**Returns**

The union of the two graphs.

### Factory Method `subtract`

```
subtract(graph_a, graph_b, simplify=True)
```

Calculate the difference of two graphs, and return the result as a new graph.

**Arguments**

`graph_a` graph, first operand.

`graph_b` graph, second operand.

`simplify` boolean, whether to simplify the resultant by removing redundant points.

**Returns**

The difference of the two graphs $a - b$.

### Factory Method `intersect`

```
intersect(graph_a, graph_b, simplify=True)
```

Calculate the intersection of two graphs, and return the result as a new graph.

**Arguments**

`graph_a` graph, first operand.

`graph_b` graph, second operand.

`simplify` boolean, whether to simplify the resultant by removing redundant points.

**Returns**

The intersection of the two graphs.

### Factory Method `sum`

```
sum(graphs, simplify=True)
```

Calculate the union of a list of graphs, and return the result as a new graph.

**Arguments** `graphs` list of graph

`simplify` boolean, whether to simplify the resultant by removing redundant points.

**Returns** The union of the list of graphs.

### Factory Method `offsetted`

```
offsetted(graph, offset)
```

Offset the edges of a graph, and return the offsetted graph as a new graph.

**Arguments** `graph_a` graph, first operand.

`offset` float, amount of offset. Positive values means expanding the graph, and negative means shrinking.

**Returns** The graph offsetted by the specified distance.

### Method `is_in_filled_region`

```
is_in_filled_region(point)
```

Check if the given point is in a filled region in this graph.

**Arguments** `point` Point object.

**Returns** Return True if the given point is in a filled region in this graph, False otherwise.

### Method `get_boundbox`

```
get_boundbox(bl, tr)
```

Calculate a rectangular boundbox that encloses all shapes in the graph.

**Arguments** `bl` Point object. Bottom-left corner of the boundbox will be stored here.

`tr` Point object. Top-right corner of the boundbox will be stored here.

**Returns** Boundbox of the graph will be stored in `bl` and `tr`.

### Method `export_svg`

```
export_svg(filename)
```

Export the graph in the SVG format.

**Arguments** `filename` String. Path to the `.svg` file where the graph should be saved.

### Method `save`

```
save(filename)
```

Export the vertices, edges and faces of a graph to a text file, which is human readable, and useful in debugging.

**Arguments** `filename` String. Path to the `.txt` file where the graph should be saved.

### Method `toPathList`

```
toPathList()
```

**Returns** List of corner points in the graph, each point is represented as a tuple `(x,y)`.

## Class GdsReader

The `GdsReader` class is responsible for reading and parsing the GdsII mask layout file.

### Factory Method `fromGdsFile`

```
fromGdsFile(fname)
```

**Arguments** `fname` string, path to the GdsII file.

**Returns** `GdsReader` object constructed from the GdsII file.

### Method `layers`

```
layers()
```

**Returns** List of integer layer numbers in the GdsII file.

### Method `top_level_structures`

```
top_level_structures()
```

**Returns** List of strings containing all the top-level instances in the GdsII file.

## Class LayerGraph

The `LayerGraph` class builds polygon graphs from the specified structure and layer from the GdsII mask layout.

### Constructor `LayerGraph`

```
LayerGraph(gds_reader)
```

**Arguments** `gds_reader` is a `GdsReader` object.

**Returns** `LayerGraph` object

### Method `build`

```
build(structure, layer)
```

**Arguments** `structure` string, GdsII instance name from which the polygon graph is generated.

`layer` integer, GdsII layer number.

**Returns** `SimplePolygonGraph` object containing the polygon graph of the given layer of the GdsII instance.

### Method `build_pad`

```
build_pad(structure, layer, label)
```

**Arguments** `structure` string, GdsII instance name from which the polygon graph is generated.

`layer` integer, GdsII layer number.

`label` string, the GdsII text label's name.

**Returns** `SimplePolygonGraph` object containing the polygon graph of the given text label of the GdsII instance.

### Method `build_boundbox`

```
build_boundbox(structure, margin=0.0)
```

**Arguments** `structure` string, GdsII instance name for which the bounding box is needed.

`margin` float

**Returns** `SimplePolygonGraph` object containing the bounding-box around the structure, with the given margin.

## Method `gds_labels`

```
gds_labels(structure, layer)
```

**Arguments** `structure` string, GdsII instance name.

**Returns** List of strings containing GdsII text labels in the layer.

## Class Structure

The `Structure` class contains the device structure being built by Gds2Mesh, and is the central class in this program.

### Constructor `Structure`

```
Structure()
```

**Returns** Empty `Structure` object

### Method `add_object`

```
add_object(polyhedron, label, material, top_bc="", \
           bottom_bc="", maxh=1e10)
```

**Arguments**

`polyhedron` `ExtrudedPolygonNG` object, region to be added to the device.

`label` string, label of the region.

`material` string, material name of this region.

`top_bc` string, name of boundary of the top surface. The defulat value is empty, which means no boundary specified.

`bottom_bc` string, name of boundary of the bottom surface. The defulat value is empty, which means no boundary specified.

`maxh` float, the max mesh size of this object. The defulat value is sufficient large to take no effect.

### Method `set_fill_object`

```
set_fill_object(polyhedron, label, material, maxh=1e10)
```

**Arguments**

`polyhedron` `ExtrudedPolygonNG` object, in which empty space will be filled by this fill region.

`label` string, label of the region.

`material` string, material name of this region.

`maxh` float, the max mesh size of fillin object. The defulat value is sufficient large to take no effect.

### Method `add_mesh_size_control`

```
add_mesh_size_control(graph, z, h)
```

**Arguments**

`graph` `SimplePolygonGraph` object, planar region to place mesh size control.

`z` float, the z-coordinate of the mesh size control.

`h` float, the mesh edge length constraint.

### Method `add_mesh_size_control`

```
add_mesh_size_control(fname)
```

**Arguments**

`fname` string, the path to the mesh size constraint file.

### Method `add_profile`

```
add_profile(profile)
```

**Arguments**

`profile` `ProfileBase` object.

### Method `do_mesh`

```
do_mesh(quality=0.5, verbose=3)
```

**Arguments**

`quality` float, mesh smooth factor between 0 and 1. High quality factor results in finer mesh, but may fail in certain situations.

`verbose` integer, meshing output verbosity level.

### Method `export_tif3d`

```
export_tif3d(tif3d_file)
```

Export the meshed structure in `TIF` format.

**Arguments** `tif3d_file` string containing the output file name.

### Method `export_gdml`

```
export_gdml(gdml_file)
```

Export the meshed structure in `GDML` format.

**Arguments** `gdml_file` string containing the output file name.

## Class ExtrudedPolygonNG

The ExtrudedPolygonNG builds a polyhedron out of a planar graph by extrusion. This is the main mechanism in Gds2Mesh to describe object geometry for planar processes.

### Constructor

**Arguments** `graph` `SimplePolygonGraph` object of the planar graph.

`base` float, z-coordinate of the base face of the extruded object.

`top` float, z-coordinate of the top face of the extruded object.

**Returns** `ExtrudedPolygonNG` object

### Constructor

**Arguments** `graph` `SimplePolygonGraph` object of the planar graph.

`heights` list of floats, containing z-coordinate of the offset stations. The first element of the list corresponds to the bottom face of the extruded object, and the last element the top face of the object.

`offsets` list of floats, containing the amount of offsets at the corresponding offset stations. All Offset values are with respect to `graph`.

**Returns** `ExtrudedPolygonNG` object

## Class PlanarUniformProfile

The `PlanarUniformProfile` class is derived from the abstract class `ProfileBase`.

### Constructors

**Arguments** `mask` `SimplePolygonGraph` object of mask graph.

`zmin` float, minimum z-coordinate of the profile.

`zmax` float, maximum z-coordinate of the profile. Profile value between `rmin` and `rmax` is constant and equals to `npeak`, and is zero elsewhere.

`species` string, species of the profile. Commonly used names are *`Acceptor`* and *`Donor`*.

`npeak` float, peak concentration ( $cm^3$).

**Returns** `PlanarUniformProfile` object

## Class PlanarAnalyticProfile

The `PlanarAnalyticProfile` class is derived from the abstract class `ProfileBase`. It supports 3D analytic profiles of Gaussian or Erfc distribution function under polygon mask.

### Constructors

**Arguments**

`mask` `SimplePolygonGraph` object of mask graph.

`z_plane` float, z-coordinate of reference plane, from which location *`rmin`* and *`rmax`* is measured.

`rmin` float, coordinate along the principal axis of the profile.

`rmax` float, coordinate along the principal axis of the profile. Profile value between `rmin` and `rmax` is constant and equals to `npeak`.

`theta` float, inclination of the principle axis, i.e. angle (in degrees) between the principal axis and $-z$ axis. Default value is 0.

`phi` float, azimuth angle (in degrees) of the principal axis. Default value is 0.

`species` string, species of the profile. Commonly used names are *`Acceptor`* and *`Donor`*.

`npeak` float, peak concentration ( $cm^3$).

`char_depth` float, characteristic length of the profile along the principal axis.

`char_lateral` float, characteristic length in the plane perpendicular to the principal axis.

`type` integer enum, possible values are *`GAUSSIAN`* and *`ERFC`*.

**Returns**

`PlanarAnalyticProfile` object

# Module graph_util

This module provides convenience utilities for creating and manipulating `gds2mesh.SimplePo` objects.

## Function `build_polygon_graph`

**Arguments**

`points` list of (x,y) tuples, each tuple sequentially describes a point in the polygon. It is not necessary to have the last point coincide with the first, the loop will be automatically closed.

`polygons` list of lists. Each nested list describes a polygon, which is in turn a list of tuples (see argument `points`).

**Returns**

`SimplePolygonGraph` object containing the union of all polygons, possibly with holes.

## Function `build_rect_graph`

**Arguments** `rects` list of (x1,y1, x2, y2) tuples, each tuple describes a rectangle.

**Returns** `SimplePolygonGraph` object containing the union of all rectangles, possibly with holes.

## Function `build_circ_graph`

**Arguments**

`center` Coordinates of the center of the circle.

`radius` Radius of the circle.

`nSegs` Number of polygon segments by which the circle is approximated.

**Returns**

`SimplePolygonGraph` object containing the circle.

## Function `calc_boundbox`

Calculate the rectangular boundbox of the graph.

**Arguments**

`graph` `SimplePolygonGraph` object.

`graphs` List of `SimplePolygonGraph` object.

**Returns**

Tuple (xmin, ymin, xmax, ymax) of the calculated boundbox.

# Module ProcessDesc

This Python module provides the `PrcoessParameters` and `ProcessCMOS classes, and aliases the following names from the gds2mesh module for convenience:`

- `Polygon` : `gds2mesh.GL2D.SimplePolygonGraph`
- `Extrusion` : `gds2mesh.ExtrudedPolygonNG`
- `PlanarAnalyticProfile` : `gds2mesh.PlanarAnalyticProfile`
- `PlanarUniformProfile` : `gds2mesh.PlanarUniformProfile`

## Class ParameterSet

### Constructor

```
ParameterSet()
```

**Returns** Empty `ParameterSet` object

### Method `getParamList`

```
getParamList()
```

**Returns** A list containing names of all parameters.

### Method `getParams`

```
getParams(keys)
```

**Arguments** `keys` Either a list containing parameter names as strings, or a single parameter name.

**Returns** A list containing values of parameters, in the order of being specified in the argument.

### Method `setParam`

```
setParam(name, val)
```

**Arguments** `name` string, parameter name.

`val` float, new value of the parameter.

# Class MaskBase

## Method `getLayerList`

```
getLayerList()
```

**Returns** Should return the list of layer names. `MaskBase` returns an empty list.

## Method `getBoundbox`

```
getBoundbox()
```

**Returns** Should return the bounding box of the mask. `MaskBase` returns None.

## Method `getLayer`

```
getLayer(layer)
```

**Arguments** `layer` string, name of the layer.

**Returns** Should return the `SimplePolygonGraph` object containing the polygons in this layer. `MaskBase` returns None.

## Class GdsiiMask

Extends from the `MaskBase` class.

### Constructor

```
GdsiiMask(fname, top_level_structure=None)
```

**Arguments** `fname` string, path to the input GDSII file.

`top_level_structure` string, name of the top level instances in the GDSII.

**Returns** Return the `GdsiiMask` object.

### Method `getLayerList`

```
getLayerList()
```

**Returns** List of layer names.

### Method `getBoundbox`

```
getBoundbox()
```

**Returns** Bounding box of the mask.

### Method `getLayer`

```
getLayer(layer)
```

**Arguments** `layer` string, name of the layer.

**Returns** `SimplePolygonGraph` object containing the polygons in this layer.

### Method `getLabels`

```
getLabels(layer)
```

**Arguments** `layer` string, name of the layer.

**Returns** list of text labels in the layer.

## Method `getPad`

```
getLabels(layer, name)
```

**Arguments** `layer` string, name of the layer.

`name` string, name of text label for the pad.

**Returns** `SimplePolygonGraph` object, the polygon in the layer that immediately contains the text label.

# Class ProcessBase

## Constructor

```
ProcessBase(params)
```

**Arguments** `params` `ParameterSet` object, process parameters.

**Returns** Return the `ProcessBase` object.

## Method `buildDevice`

```
buildDevice()
```

Build the 3D geometry of the device structure.

## Method `doMesh`

```
doMesh(quality=0.3)
```

Mesh the device structure, must be called after `buildDevice`().

**Arguments** `quality` float, smooth factor of the mesh.

## Method `save`

```
save(fname)
```

Save the mesh to a file in TIF3D or GDML format.

**Arguments** `fname` string, path to the output file.

# Function loadProcessFile

```
loadProcessFile(fname)
```

Load a process script file, and import from the script classes that are derived from `ProcessBase` or `MaskBase`.

**Arguments** `fname` string, path to the .py process script file.

# A File Format of TIF3D

The TIF3D is a simple ASCII file for describing 3D tetrahedral mesh of semicondcutor device. Each line of the TID3D is a record begin with its record type indicator and followed by all the record data.

Help record: Begin with 'H', followed by user defined string, usually contains description of this file.

```
H TIF3D V1.1 created by GDS2MESH
```

Coordinate record: Begin with 'C', followed by 0 based index and three float numbers which specify space location of a node.

```
C <index> <x> <y> <z>
```

Face record: Begin with 'F', followed by 0 based index, three node indexes which consist the triangular face and a integer boundary mark. The order of nodes on the face is not concerned. Gds2mesh will export all the boundary faces of each region. And the face which has boundary label specified by interface record has the mark greater than one.

```
F <index> <n0> <n1> <n2> <mark>
```

Tetrahedron record: Begin with 'T', followed by 0 based index, then the integer region index, and the four node of the tetrahedron. The nodes are ordered as node n4 is above the plane made by n1 n2 and n3.

```
T <index> <region> <n1> <n2> <n3> <n4>
```

Region record: Begin with 'R', followed by 0 based index, the material string, the label string of the region and a integer group code. The region record is used to specify material infomation for the tetrahedrons with the corresponding region index.

```
R <index> <material> <label> <group>
```

Interface record: Begin with 'I', followed with 0 based index, the label of the interface and the integer boundary mark. The interface record links the boundary label and the boundary make given in face record.

```
I <index> <label> <mark>
```

Solution record: Begin with 'S', followed by solution number and the name of each solution. The TIF3D generated by Gds2Mesh contains doping information such as: "Net", "Total", "Donor" and "Acceptor".

```
S <num> <sol> <sol> ...
```

Data record: Begin with 'N', followed with node index, the region index of the node belongs to and then the solution data corresponding to solution record. For node on the interface of different regions, there will be multiply data records for each region exist.

```
N <n> <index> <data> <data> ...
```