Part 6 – Utilities

2 – DF–ISE

2.1 About DF–ISE

A unique file format is used to store all information about the current status of a device. If new data items have to be exchanged between some of the simulators, using keywords and grouping data into data records allows the extension of the file format without changing the tools.

DF–ISE tools use a common file that contains information about materials and functions that is shared by different simulation tools. Generic properties (for example, materials insulator, semiconductor, and metal) can be used to enable some treatment of new materials or functions.

The detailed structures of the DF–ISE data blocks recognized by all IIS/ISE simulation tools are described in the following chapters. DF–ISE data blocks contain a representation of the device geometry (that is, a representation of the regions of the device or a spatial tessellation of the regions) and/or data values, defined on that tessellation.

A DF–ISE file has, at least, three main parts:

- A header that is used to identify DF–ISE files
- An `Info` block that contains general information about the data stored in this file
- A `Data` block giving a detailed description of geometry, datasets, or properties

User-defined keywords and data items can be added freely after the `Data` section. The IIS/ISE tools ignore these data records (other tools may not).

A library providing basic reading and writing has been developed. The description of the library functions with the exact calling sequences is described in the include files of this library.

2.2 DF–ISE files

DF–ISE files are organized in blocks. An overview of the top-level blocks of a DF–ISE file is presented:

`DF-ISE`	This block identifies DF–ISE files. No comments or other blocks precede this identifier. The `DF-ISE` block has no arguments and no body; therefore, it does not enclose any other block.
`text | binary`	This keyword follows `DF-ISE` and indicates whether the rest of the file is readable or in binary format. Only integer and floating point values can be stored in binary format.
`Info`	This block provides additional information about the stored data.
`Data`	This block is linked to a preceding `Info` block. Its content depends on the type entry of the information block. The following data block types are currently supported by the format:

`layout`	Layout geometry used for patterning operations. `layout` file names should have the extension `.lyt`.

`cell`	Cell structures, mainly used for 3D solid modeling. `cell` file names should have the extension `.cel`.
`recursive-tensor`	Recursive tensor product grids used for compact storage of some devices. `recursive-tensor` – File names should have the extension `.ten`.
`boundary`	The geometrical data in this block is a boundary description of a semiconductor device. `boundary` file names should have the extension `.bnd`.
`grid`	A finite-element discretization of the device is stored in this block. The boundary representation of regions that do not need a simulation grid are not stored here, but in the `boundary data` block of the corresponding `boundary` file. `grid` file names should have the extension `.grd`.
`dataset`	This section is used to store scalar or vector values on topological elements of `recursive-tensor`, `boundary`, and `grid` files. `dataset` file names should have the extension `.dat`.
`xyplot`	Data that does not correspond to a simulation grid or boundary representation is stored in this section. Data can be displayed with x-y or x-y-z plot tools. `Plot` file names should have the extension`.plt`.
`property`	The global property database is stored in these data blocks. IIS/ISE tools use a common database that contains information about materials and functions that need to be shared by different simulation tools. A `property` file should have the extension `.pro`.

2.3 DF–ISE syntax

A formal definition of the DF–ISE syntax follows:

```
file        := DF–ISE text | binary <new_line> blocks
blocks      := <empty> | block | blocks
block       := keyword ['(' integer ')'] [body]
body        := '{' [blocks | values] '}' | '=' data
data        := array | values
array       := '[' values ']'
values      := <empty> | value | values
value       := integer | float | keyword | string
```

A DF–ISE file comprises a sequence of blocks, which can contain further blocks. A block always has the following structure: `block_name (argument){body}`.

Both the argument and the body of a block are optional. The body of a block is either a sequence of blocks or a sequence of data values. Finally, a data value is an integer, a floating point value, a keyword, a string delimited by quotation marks, or an array of data values. Comments start with `#` and end at the end of line.

NOTE Comments cannot precede the `DF-ISE` block.

Blocks without body and data values are separated by space characters or new lines. The order of the blocks in a file is given by the DF–ISE format.

When storing a binary `DF-ISE` file, an application can mix readable integer and floating values with compressed values. An application that reads a binary `DF-ISE` file must be able to determine whether the next value in the file is readable or in binary format. The DF–ISE library contains all the functions needed for reading and writing values.

NOTE The precision of readable floating point values depends on the application that writes the file.

2.4 DF–ISE coordinate system

The DF–ISE coordinate system is always a right-handed Cartesian system. All coordinates are in micrometers. However, the 'up' direction – the vector starting from the substrate floor and pointing towards the device surface – depends on the application. The old generation of IIS/ISE tools uses the negative y-axis as the up direction. Whenever possible, the new orientation proposed below should be used.

2.4.1 Reference coordinate system

The DF–ISE reference coordinate system is right-handed as shown in Figure 6.1. The `z = 0` plane should be placed on the initial substrate surface, for example, before starting process operations.

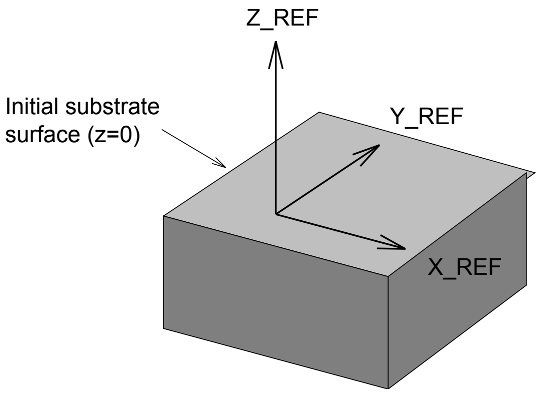

Figure 6.1 Reference coordinate system with initial substrate example

The layout data for patterning operations on the rear side of the wafer must be defined with the same coordinate system used for the front side. This means that the user who edits mask layers is always looking at the front side of the wafer (see Figure 6.2).

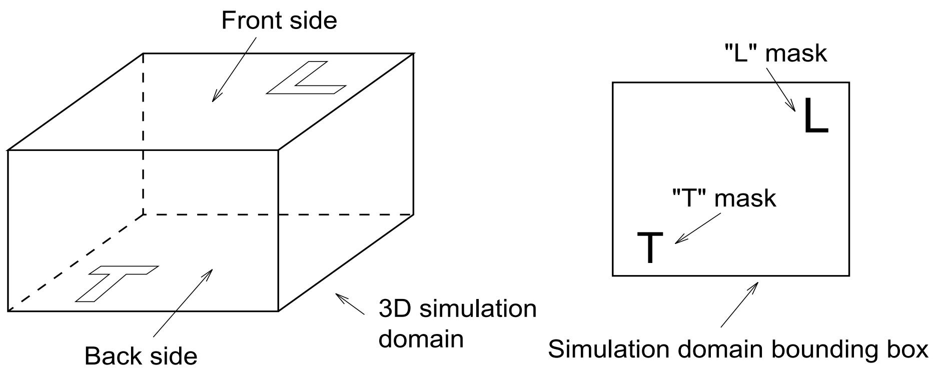

Figure 6.2 Editing layout data for front and rear sides

2.4.2 Coordinate transformation

A coordinate transformation can be specified in `recursive-tensor`, `boundary`, and `grid` files. A transformation is used to map 1D, 2D, or 3D devices to their real place in the 3D wafer. An application that reads devices only has to apply the coordinate transformation to the vertices of the device, except if the application is using the original 3D wafer coordinate system internally.

Nothing is assumed regarding the up direction of the global 3D coordinate system. However, the orientation proposed in the previous section should be used whenever possible.

The transformation from user coordinates to the global reference coordinate system is performed as follows:

$$\begin{pmatrix} x_{ref} \\ y_{ref} \\ z_{ref} \end{pmatrix} = \begin{bmatrix} xx & xy & xz \\ yx & yy & yz \\ zx & zy & zz \end{bmatrix} \times \begin{pmatrix} x_{user} \\ y_{user} \\ z_{user} \end{pmatrix} + \begin{pmatrix} dx \\ dy \\ dz \end{pmatrix} \quad \text{[Eq. 6.1]}$$

The translation vector and transformation matrix are given in the `CoordSystem` block in this way:

```
translate = [ <dx> <dy> <dz> ]
transform = [ <xx> <xy> <xz> <yx> <yy> <yz> <zx> <zy> <zz> ]
```

2.5 Location codes

Location codes identify the position of vertices, edges, or faces. For example, a location code tells whether a face is external (or visible), or if it is on an interface between two regions, or if the face is fully inside a 3D region. Location codes must be stored in `boundary` and `grid` files.

2.5.1 One-dimensional geometries

For one-dimensional geometries, a location code is defined for vertices. If a vertex is shared by more than one segment belonging to different regions, it obtains the location code `f` (internal interface). If the vertex is defined only in one segment of one region, it has the location code `e` (external interface), and if more than one segment of the same region share the vertex, it has the code `i` (internal). For vertices shared only by points or segments that do not belong to any region, the location code is `u` (undefined).

2.5.2 Two-dimensional geometries

For two-dimensional geometries, the location code is defined for edges. If an edge is contained only in one 2D element (triangle, rectangle, polygon) which belongs to a region, the edge is called ‘external interface.’ If it is shared by more than one 2D elements that belong to different regions, it is called ‘internal interface.’ If more than one 2D elements that belong to a region share the edge, this edge is called ‘internal,’ and if there are only elements not belonging to any region and line segments sharing the edge, the location code of the edge is ‘undefined.’

2.5.3 Three-dimensional geometries

For three-dimensional geometries, the location code is defined for faces. If a face belongs to exactly one 3D element included in a region, the face is called ‘external interface.’ If the face is shared by two 3D elements belonging to different regions, it is called ‘internal interface.’ If the face is shared by two 3D elements all belonging to the same region, the face is called ‘internal.’ Finally, if the face is shared by 3D elements that do not belong to any region or by 2D elements, the location code is ‘undefined.’

2.6 Properties

Properties identify and characterize datasets. A global file `dfise.pro` that contains a list of properties is provided in the ISE TCAD distribution. A tool that needs the information contained in this file must read it during startup. `layout`, `cell`, `recursive-tensor`, `boundary`, `grid`, `dataset`, and `plot` files reference the properties defined in the property database. Multiple properties for one region or one dataset are allowed. In this case, the material or function name is a list of property names concatenated with `&`. For example, an interface region can have the material `Silicon&Oxide`. This combined material name is identical to `Oxide&Silicon`, but should not be confused with `SiliconOxide`. The attributes of combined properties are not defined. An application can select any of the entries for setting colors, interpolation, and so on. Binary compression is not allowed for this file.

Two main property types are defined: `Material`, which is used to define region properties, and `Function`, which defines the property of any dataset. `cell` and `layout` files do not distinguish between `Material` and `Function` properties. Thus, a property name must be unique within the complete property file.

Currently, the following attributes are understood by most of the IIS/ISE tools:

`symbol`	A short quoted string that identifies this property. The symbol is for display purposes only, thus it does not have to be unique.
`color`	Any valid X11 color. It is used for displaying curves or drawing regions.
`style`	Only valid for `Function`. Line style when displaying curves (`solid`, `dashed`, `dotted`, `ldashed`, `ldotted`).

`formula`	Only valid for `Function`. It is used to compute datasets that have this property from other datasets. A mathematical expression using property names must be provided here. Variable or property names must be enclosed in angle brackets < >.
`unit`	Only valid for `Function`. This is a quoted string that defines the unit of a dataset.
`interpol`	Only valid for `Function`. Interpolation function (linear, log, arsinh).

```
DF-ISE text
Info {
  version = 1.0
  type = property
}
Data {
  Material (SiliconOxide) {
        symbol = "SiO2"
        color  = brown
  }
  ...
  Function (eDensity) {
        symbol   = "n"
        unit     = "1/cm^3"
        interpol = arsinh
        color    = red
        style    = dashed
  }
  Function (TotalDopingConcentration) {
        symbol   = "N-tot"
        unit     = "1/cm^3"
        interpol = arsinh
        color    = red
        style    = dashed
        formula  = "<BoronConcentration> + ..."
  }
  ...
}
```

2.7 Layout

The `layout` file type stores the starting geometry (`layout` files) and the photolithography data (`mask` files) required by all patterning operations of the process flow.

NOTE Simulation domains (a point for 1D, a line for 2D, and a polygon for 3D) can also be stored in `DF-ISE layout` files. They have `Sim1D`, `Sim2D`, and `Sim3D` as predefined property names. The property list is also used to find the color of the layers in the material blocks of the property database.

A polygon entry of the `Polygons` block can have one or two vertices when it is used to define a 1D or 2D simulation domain. The order of appearance of `Region` blocks must correspond to the `regions` entry list of the `Info` block. The file structure is:

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = layout
  nb_vertices = <int>
  nb_polygons = <int>
```

```
    nb_regions  = <int>
    regions     = [ "<region1>" "<region2>" ... ]
    materials   = [ <material1> <material2> ... ]
  }

  Data {

    Vertices (<nb_vertices>) {
          <x0> <y0>
          <x1> <y1>
           ...
    }

    Polygons (<nb_polygons>) {
          <nb_vertices> <vertex> <vertex> ...
          <nb_vertices> <vertex> <vertex> ...
          ...
    }

    Region ("<material1>") {
          material = <material1>
          Elements (<nb_elts>) { <elt0> <elt1> ... }
    }

    ...
  }
```

2.8 Cell

The ‘Cell’ data structure uses a regular subdivision of the simulation domain into blocks of equal size (cubes). Each block can be refined into groups of cells by splitting it along the x-, y-, or z-axis (see Figure 6.3). The construction of general, not axis-aligned structures is achieved by defining a material density value for each cell. Storing material densities is optional – by default, the maximal density value 1.0 is assumed. An unrefined block always has a material density of 1.0.

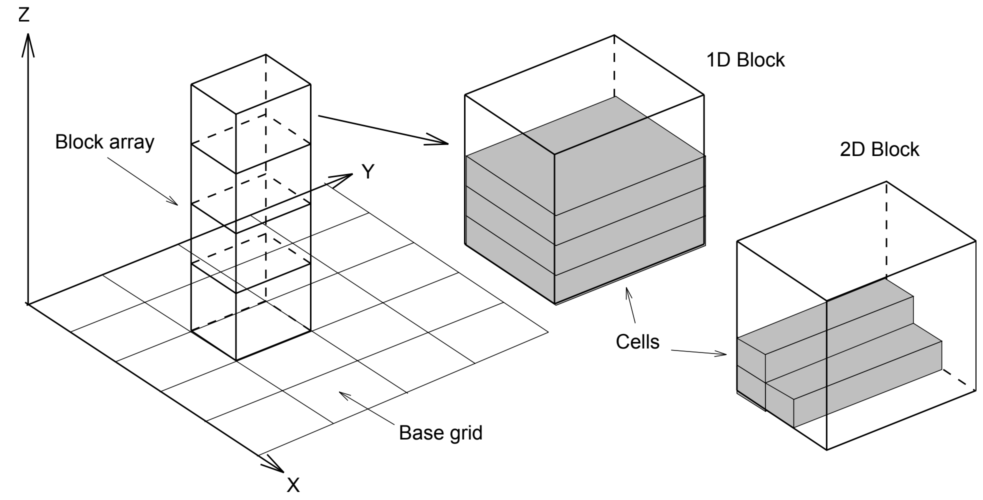

Figure 6.3 Cell data structure

Geometry and datasets are stored in the same file and use the same format. Region identifiers and material densities are considered as normal datasets. However, they have the reserved property names *regions* and *densities*.

NOTE The refinement of the different datasets does not have to be identical, except for the material density refinement, which must always match the region data. A `property` entry can either be an entry of the material list or an entry of the function list in the property database.

The `Info` block defines the simulation domain size and refinement information. `block_size` is the size in micrometers of the initial cubes building the simulation domain. The size of the simulation domain is given by multiplying `block_size` with `nb_xblocks`, `nb_yblocks`, or `nb_zblocks`. When a block (cube) is refined, `nb_xsplits`, `nb_ysplits`, and `nb_zsplits` define the number of cells for the given axis.

The numbering of blocks and cells always starts at the lowest coordinate. Indices are incremented by traversing the structure first along the x-axis, then along the y-axis, and finally along the z-axis. In practice, traversing the data structure is performed in this way:

```
for (bz = 0; bz < nb_zblocks; bz++) {
for (by = 0; by < nb_yblocks; by++) {
for (bx = 0; bx < nb_xblocks; bx++) {
   if (block[bx][by][bz] is refined)
   {
      for (cz = 0; cz < nb_zsplits; cz++) {
      for (cy = 0; cy < nb_ysplits; cy++) {
      for (cx = 0; cx < nb_xsplits; cx++) {
      ...
      }
      }
      }
   }
}
}
}
```

The order of appearance of `Dataset` blocks must correspond to the `datasets` entry list of the `Info` block. The structure of the file is:

```
DF-ISE text | binary
Info {
   version    = 1.0
   type       = cell
   dimension  = 1 | 2 | 3
   block_size = <double>
   nb_xblocks = <int>
   nb_yblocks = <int>
   nb_zblocks = <int>
   nb_xsplits = <int>
   nb_ysplits = <int>
   nb_zsplits = <int>
   nb_datasets = <int>
   datasets   = [ "<dataset1>" "<dataset2>" ... ]
   properties = [ <property1> <property2> ... ]
}
Data {

Dataset ("<dataset1>") {
   property = <property1>

   Blocks (<nb_blocks>) {
      <ref_flag> <value> ...
      <ref_flag> <value> ...
      ...
      }
   }

   Dataset ("<dataset2>") {
      ...
   }
   ...

   <ref_flag> is:
      0: no refinement, only one data values follows
      1: refinement along x, nb_xsplits data values follow
      2: refinement along y, nb_ysplits data values follow
      4: refinement along z, nb_zsplits data values follow
      3: refinement along x and y, nb_xsplits * nb_ysplits values follow
      ...
      7: refinement along x, y, and z, nb_xsplits * nb_ysplits
         * nb_zsplits data values follow
   }
```

2.9 Recursive tensor

A recursive grid is a generalization of a tensor grid. In addition to a base grid that is a tensor grid, it allows recursive subdivision of elements. Vertices, edges, faces, and elements are defined implicitly through the grid and its traversal rules. The tensor base grid can be:

Uniform	Rectilinear grid with constant spacing in x-, y-, and z-direction (the spacing for each direction is given separately).
Rectilinear	Rectilinear grid with nonconstant spacing between coordinate planes.
Warped	Grid with tensor mesh topology and explicit coordinates for each vertex.

The numbering of vertices, edges, faces, and elements starts with index 0 at the lowest coordinates and goes first through the x-axis, y, and then z. The entries `nb_x`, `nb_y`, and `nb_z` in the `Info` block represent the number of vertices of the base grid in each of the dimensions. Vertices, edges, faces, and elements are implicitly numbered as follows:

- There is a total of $(nb\_x)(nb\_y)(nb\_z)$ nodes. Node (i,j,k) has number $i+j(nb\_y)+k(nb\_y)(nb\_z)$, where $0<i<nb\_x$, $0<j<nb\_y$, $0<k<nb\_z$.
- There is a total of $(nb\_x-1)(nb\_y)(nb\_z)+(nb\_x)(nb\_y-1)(nb\_z)+(nb\_x)(nb\_y)(nb\_z-1)$ edges. The edges are numbered or traversed in the following way: first we go through the vertices in the order given below, and for each vertex we take first the edge pointing along the positive x-axis, then the one pointing along the positive y-axis, and finally the one pointing along the positive z-axis. At the boundary of the simulation domain, the edge index is, of course, not incremented when no edge exists in the given direction.
- There is a total of $(nb\_x)(nb\_y-1)(nb\_z-1)+(nb\_x-1)(nb\_y)(nb\_z-1)+(nb\_x-1)(nb\_y-1)(nb\_z)$ faces. The face numbering is analogous to the edge numbering. At each vertex, we take the three faces orthogonal to the x, y, and z axes.
- There is, in principle, a total of $(nb\_x-1)(nb\_y-1)(nb\_z-1)$ elements. However, additional dummy elements are added at the upper bounds of the coordinate ranges, so that total number of elements is $(nb\_x)(nb\_y)(nb\_z)$. Element (i,j,k) has, therefore, the number $i+j(nb\_y)+k(nb\_y)(nb\_z)$, where $0<i< nb\_x$, $0<j<nb\_y$, $0<k<nb\_z$.

The contents of the `BaseGrid` block depend on the tensor grid subtype as follows:

- For `uniform` grids, it contains the spacings Δx, Δy, and Δz in the directions of the coordinate planes. Vertex number 0 is implicitly at (0,0,0); the `CordSystem` block can be used for further moving.
- For `rectilinear` grids, it contains the spacings $\Delta x_1, ..., \Delta x_{nb\_x-1}, \Delta y_1, ..., \Delta y_{nb\_y-1}, \Delta z_1, ..., \Delta z_{nb\_z-1}$, that is, there are nb_x-1 entries for the x-direction, nb_y-1 entries for the y-direction, and nb_z-1 entries for the z-direction. Vertex number 0 is implicitly at (0,0,0); the `CordSystem` block can be used for further moving.
- For `warped` grids, it contains the coordinates of the grid vertices x_0, y_0, z_0, x_1, y_1, z_1, ..., x_{nb_v}, y_{nb_v}, and z_{nb_v} where the numbering is the natural cell numbering, that is, varying fastest along the x-axis.

The `CellTree` block is used only if the mesh is recursively refined; if it is missing, the data structure is an ordinary tensor mesh. Otherwise, for each macroelement including the dummy elements, the recursive grid structure is indicated:

```
<min_lev> <max_lev> <nb_el> <sub_el0> <sub_el1> ... <sub_el_nb_el>
```

`min_lev` and `max_lev` are the minimum and maximum refinement levels in the macroelement. A value of 0 means that the macroelement is unrefined. The number of elements `nb_el` gives the number of elements in the macroelement that result in the final mesh. A value of 0 indicates that the macroelement is a dummy element or deleted from the mesh; a value of 1 means that it is unrefined.

If the minimum and maximum refinement levels of a macroelement are equal, the mesh can be constructed implicitly, and the mesh description within the macroelement is complete. If the refinement levels are not equal, they are followed by a list of the subelements `sub_el` within the macroelement. A simple numbering scheme is used to indicate the subelements. The numbering proceeds looping through each of the parent elements at a level. The following is an example of a 2D scheme:

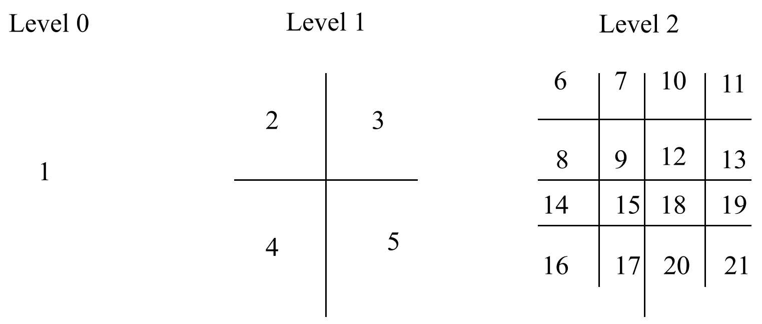

Figure 6.4 Numbering scheme for recursively subdivided cells

The coordinates of the joints (vertices) of the refined grid are currently determined uniquely by the base grid and the cell tree.

`Regions` are used to denote materials and boundary conditions. `Regions` can reference vertices, edges, faces, and elements of the base grid (not of the refined grid) through their implicitly defined number (see above). The order of appearance of `Region` blocks must correspond to the `regions` entry list of the `Info` block. The material entry can be left empty.

For data on the recursive grids, the DF–ISE data file format is used.

An example of the structure of a DF–ISE recursive tensor grid file is:

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = recursive-tensor
  subtype     = uniform | rectilinear | warped
  dimension   = 1 | 2 | 3
  extents     = <int> | <int> <int> | <int int>
  nb_regions  = <int>
  regions     = [ "<region1>" "<region2>" ... ]
  materials   = [ <material1> <material2> ... ]
}

Data {

  CoordSystem {
        translate = [ <dx> <dy> <dz> ]
        transform = [ <xx> <xy> <xz> <yx> <yy> <yz> <zx> <zy> <zz> ]
  }

  BaseGrid (<nb_entries>) {
        <e0> <e1> <e2>
  }

  CellTree (<nb_macro_elements>) {
       <min_lev> <max_lev> <nb_el> <sub_el0> <sub_el1> ... <sub_el_nb_el>
       ...
       ...
  }

Region ("<region1>") {
       material = <material1>
        Edges    (<nb_edge>) { <edge0> <edge1> ... }
        Faces    (<nb_face>) { <face0> <face1> ... }
  }

Region ("<region2") {
   material = <material2>
        Vertices (<nb_vert>) { <vert0> <vert1> ... }
  }

Region ("<region3>") {
   material = <material3>
        Faces    (<nb_fac>)  { <fac0>  <fac1>  ... }
  }
Region ("<region4 >") {
       material = <material4>
        Elements (<nb_elts>) { <elt0>  <elt1>  ... }
  }
}
```

2.10 Variable tensor

A variable tensor grid is a generalization of a recursive tensor grid. The recursive subdivisions allowed are not restricted to powers of two, and the addition of a property tree allows for property changes within a cell of the tensor grid In addition to a base grid that is a tensor grid, they allow recursive subdivision of elements.

Vertices, edges, faces, and elements are defined implicitly through the grid and its traversal rules. The tensor base grid can be:

Uniform	Rectilinear grid with constant spacing in x-, y-, and z-direction (the spacing for each direction is given separately).
Rectilinear	Rectilinear grid with nonconstant spacing between coordinate planes.
Warped	Grid with tensor mesh topology and explicit coordinates for each vertex.

The numbering of vertices, edges, faces and elements starts with index 0 at the lowest coordinates and goes first through the x-axis, y-axis, and then z-axis. The entries `nb_x`, `nb_y` and `nb_z` in the `Info` block represent the number of vertices of the base grid in each of the dimensions. Vertices, edges, faces, and elements are implicitly numbered as follows:

- There is a total of $(nb\_x)(nb\_y)(nb\_z)$ nodes. Node (i,j,k) has number $i+j(nb\_y)+k(nb\_y)(nb\_z)$, where $0<i<nb\_x$, $0<j<nb\_y$, $0<k<nb\_z$.
- There is a total of $(nb\_x-1)(nb\_y)(nb\_z)+(nb\_x)(nb\_y-1)(nb\_z)+(nb\_x)(nb\_y)(nb\_z-1)$ edges. The edges are numbered or traversed in the following way: first we go through the vertices in the order given below, and for each vertex we take first the edge pointing along the positive x-axis, then the one pointing along the positive y-axis, and finally the one pointing along the positive z-axis. At the boundary of the simulation domain, the edge index is, of course, not incremented when no edge exists in the given direction.
- There is a total of $(nb\_x)(nb\_y-1)(nb\_z-1)+(nb\_x-1)(nb\_y)(nb\_z-1)+(nb\_x-1)(nb\_y-1)(nb\_z)$ faces. The face numbering is analogous to the edge numbering. At each vertex, we take the three faces orthogonal to the x, y, and z axes.
- There is, in principle, a total of $(nb\_x-1)(nb\_y-1)(nb\_z-1)$ elements. However, additional dummy elements are added at the upper bounds of the coordinate ranges, so that total number of elements is $(nb\_x)(nb\_y)(nb\_z)$. Element (i,j,k) has, therefore, the number $i+j(nb\_y)+k(nb\_y)(nb\_z)$, where $0<i< nb\_x$, $0<j<nb\_y$, $0<k<nb\_z$.

The contents of the `BaseGrid` block depend on the tensor grid subtype as follows:

- For `uniform` grids, it contains the spacings, Δx, Δy, and Δz in the directions of the coordinate planes. Vertex number 0 is implicitly at (0,0,0); the `CordSystem` block can be used for further moving.
- For `rectilinear` grids, it contains the spacings $\Delta x_1, ..., \Delta x_{nb\_x-1}, \Delta y_1, ..., \Delta y_{nb\_y-1}, \Delta z_1, ..., \Delta z_{nb\_z-1}$, that is, there are nb_x-1 entries for the x-direction, nb_y-1 entries for the y-direction, and nb_z-1 entries for the z-direction. Vertex number 0 is implicitly at (0,0,0); the `CordSystem` block can be used for further moving.
- For `warped` grids, it contains the coordinates of the grid vertices $x_0, y_0, z_0, x_1, y_1, z_1, ..., x_{nb\_v}, y_{nb\_v}$, and z_{nb_v} where the numbering is the natural cell numbering, that is, varying fastest along the x-axis.

The `CellTree` block is used only if the mesh is recursively refined; if it is missing, the data structure is an ordinary tensor mesh. Only those macro cells that are refined will have an entry in the `CellTree`. Such entries are defined as follows:

```
<index> <x-refinement> <y-refinement> <z-refinement> <nb_el> <sub_el0> <sub_el1> ... <sub_el_nb_el>
```

`index` refers to the cell index of the refined macro cell. `x-refinement`, `y-refinement`, and `z-refinement`, give the number of refinement levels in the respective direction. The number of elements `nb_el` gives the number of elements in the macroelement that result in the final mesh. This is followed by a list of the subelements `sub_el` within the macroelement. A simple numbering scheme is used to indicate the subelements. The numbering proceeds looping through each of the parent elements at a level. This example shows a 2D scheme:

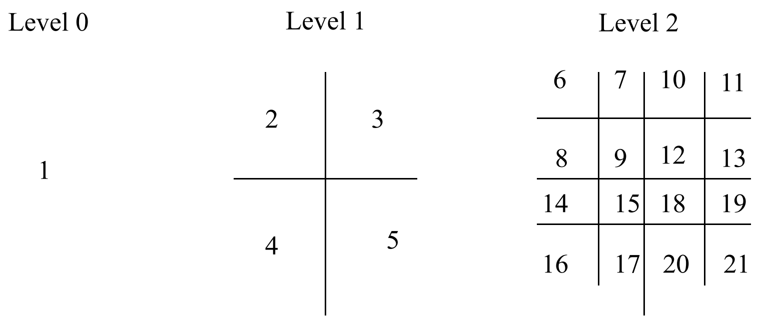

Figure 6.5 Numbering scheme for recursively subdivided cells

The coordinates of the joints (vertices) of the refined grid is currently determined uniquely by the base grid and the cell tree. The numbering scheme and calculation of coordinates are performed analogously to the case of the recursive tensor grid.

The *Properties* can be used to assign different properties, for example, materials, to the subcells of a refined macro cell:

```
<index> <nb_el> <prop_sub_el0> <prop_sub_el1> ... <prop_sub_el_nb_el>
```

`index` refers to the cell index of the refined macro cell; it should also appear in the `CellTree`. The refinements are taken from `CellTree`, and the properties of the subcells have to refer one-to-one to the subcells of the respective entry in `CellTree`. Although each entry of *Properties* should have an entry in `CellTree`, the reverse is not true. In general, single entries or the entire `Properties` block can be missing, in which case all subcells are assumed to have the same property.

Regions are used to denote materials and boundary conditions. Regions can reference vertices, edges, faces, and elements of the base grid (not of the refined grid) through their implicitly defined number (see above). The order of appearance of `Region` blocks must correspond to the `regions` entry list of the `Info` block. The `material` entry can be left empty.

For data on the variable tensor grids, the DF–ISE data file format is used.

An example of the structure of a DF–ISE variable tensor grid file is:

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = variable-tensor
  subtype     = uniform | rectilinear | warped
  dimension   = 1 | 2 | 3
  extents     = <int> | <int> <int> | <int int>
  nb_regions  = <int>
  regions     = [ "<region1>" "<region2>" ... ]
  materials   = [ <material1> <material2> ... ]
}

Data {

  CoordSystem {
        translate = [ <dx> <dy> <dz> ]
        transform = [ <xx> <xy> <xz> <yx> <yy> <yz> <zx> <zy> <zz> ]
  }

BaseGrid (<nb_entries>) {
        <e0> <e1> <e2>
  }

  CellTree (<nb_macro_elements>) {
   <index> <x-refinement> <y-refinement> <z-refinement> <nb_el> <sub_el0> <sub_el1> ...
   <sub_el_nb_el>
   ...
   ...
  }


Properties(<nb_macro_elements>) {
   <index> <nb_el> <prop_sub_el0> <prop_sub_el1> ... <prop_sub_el_nb_el>
   ...
   ...
  }

Region ("<region1>") {
   material = <material1>
       Edges    (<nb_edge>) { <edge0> <edge1> ... }
       Faces    (<nb_face>) { <face0> <face1> ... }
  }
Region ("<region2") {
   material = <material2>
       Vertices (<nb_vert>) { <vert0> <vert1> ... }
  }
Region ("<region3>") {
   material = <material3>
       Faces    (<nb_fac>)  { <fac0>  <fac1>  ... }
  }
Region ("<region4 >") {
   material = <material4>
       Elements (<nb_elts>) { <elt0>  <elt1>  ... }
  }

}
```

2.11 Boundary and grid

Boundary and grid files have a similar format. However, the interpretation of the contents is different. Each region must have one material name. The numbering of vertices, edges, and faces is given in Figure 6.6 on page 6.19 and Figure 6.7 on page 6.20. The second picture is only required when data must be stored on vertices or edges locally for each element.

An index to a face is signed. When negative, the real face index is computed by: $face\_index = -index - 1$. The negative sign means that the edge order of the face as well as the edge orientation must be inverted. Face orientation is important to obtain a consistent orientation of the object (counterclockwise orientation of the edges when looking from outside of the solid including this face). An edge index is signed in the same way as a face index. A negative edge sign means that the first vertex and not the second is connected to the next edge of a loop.

NOTE The DF–ISE library provides a function to fix the orientation of the faces of a 3D boundary.

Elements have a shape code that defines the element geometry and the data following this code. Elements of different shape can be given in any order. The following elements are predefined:

Point	`0` *`vertex`*
Segment	`1` *`vertex0 vertex1`*
Triangle	`2` *`edge0 edge1 edge2`*
Rectangle	`3` *`edge0 edge1 edge2 edge3`*
Polygon	`4` *`nb_edges edge0 edge1...`*
Tetrahedron	`5` *`face0... face3`*
Pyramid	`6` *`face0... face4`*
Prism	`7` *`face0... face4`*
Brick	`8` *`face0... face5`*
Tetrabrick	`9` *`face0... face6`*
Polyhedron	`10` *`nb_faces face0 face1...`*

The edge and face lists of polygons and polyhedra do not have to be connected. They can be a set of loops and shells that define the outer boundary and internal holes of an element.

An edge orientation for each element is also predefined. The edge orientation is used when storing data on edges locally for each element. The vertex order for each edge of the elements are given below:

Segment:
e0: v0 v1

Triangle:
e0: v0 v1
e1: v1 v2
e2: v2 v0

Rectangle:
e0: v0 v1
e1: v1 v2
e2: v2 v3
e3: v3 v0

Polygon:
e0: v0 v1
e1: v1 v2
…

Tetrahedron:
e0: v0 v1
e1: v1 v2
e2: v2 v0
e3: v0 v3
e4: v1 v3
e5: v2 v3

Pyramid:
e0: v0 v1
e1: v1 v2
e2: v2 v3
e3: v3 v0
e4: v0 v4
e5: v1 v4
e6: v2 v4
e7: v3 v4

Prism:
e0: v0 v1
e1: v1 v2
e2: v2 v0
e3: v0 v3
e4: v1 v4
e5: v2 v5
e6: v3 v4
e7: v4 v5
e8: v5 v3

Brick:
e0: v0 v1
e1: v1 v2
e2: v2 v3
e3: v3 v0
e4: v0 v4
e5: v1 v5
e6: v2 v6
e7: v3 v7
e8: v4 v5
e9: v5 v6
e10: v6 v7
e11: v7 v4

Tetrabrick:

e0: v0 v1	e6: v2 v5
e1: v1 v2	e7: v2 v6
e2: v2 v3	e8: v3 v6
e3: v3 v0	e9: v4 v5
e4: v0 v4	e10: v5 v6
e5: v1 v5	e11: v6 v4

The order of appearance of `Region` blocks must correspond to the `regions` entry list of the `Info` block. Several regions with the same name are allowed.

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = boundary | grid
  dimension   = 1 | 2 | 3
  nb_vertices = <int>
  nb_edges    = <int>
  nb_faces    = <int>
  nb_elements = <int>
  nb_regions  = <int>
  regions     = [ "<region1>" "<region2>" ... ]
  materials   = [ <material1> <material2> ... ]
}
Data {

   CoordSystem {
       translate = [ <dx> <dy> <dz> ]
       transform = [ <xx> <xy> <xz> <yx> <yy> <yz> <zx> <zy> <zz> ]
  }
   for 1d, 2d, and 3d
   Vertices (<nb_vertices>) {
       <x0> <y0> <z0>
       <x1> <y1> <z1>
        ...
  }
   for 2d and 3d
   Edges (<nb_edges>) {
       <vertex0> <vertex1>
       <vertex0> <vertex1>
       ...
  }
   for 3d
   Faces (<nb_faces>) {
       <nb_edges> <edge0> <edge1> ... <edge_n>
       <nb_edges> <edge> <edge> ...
       ...
  }
   nb_vert. for 1d, nb_edges for 2d, nb_faces for 3d
   Locations (<nb_vertices> | <nb_edges> | <nb_faces>) {
       i (internal) | f (internal interface) |
       e (external interface) | u (undefined)
       ...
  }
Elements (<nb_elements>) {
        0 <vertex>
        1 <vertex0 <vertex1>
        2 <edge0 <edge1> <edge2>
        3 <edge0 <edge1> <edge2> <edge3>
        4 <nb_edges> <edge0> <edge1> ... <edge_n>
        5 <face0 ... <face3>
        6 <face0 ... <face4>
        7 <face0 ... <face4>
        8 <face0 ... <face5>
        9 <face0 ... <face6>
        10 <nb_faces> <face0> <face1> ... <face_n>
       ...
  }
Region ("<region1>") {
        material = <material1>
        Elements (<nb_elts>) { <elt0> <elt1> ... }
  }
Region ("<region2>") {
        material = <material2>
        Elements (<nb_elts>) { ... }
  }
}
```

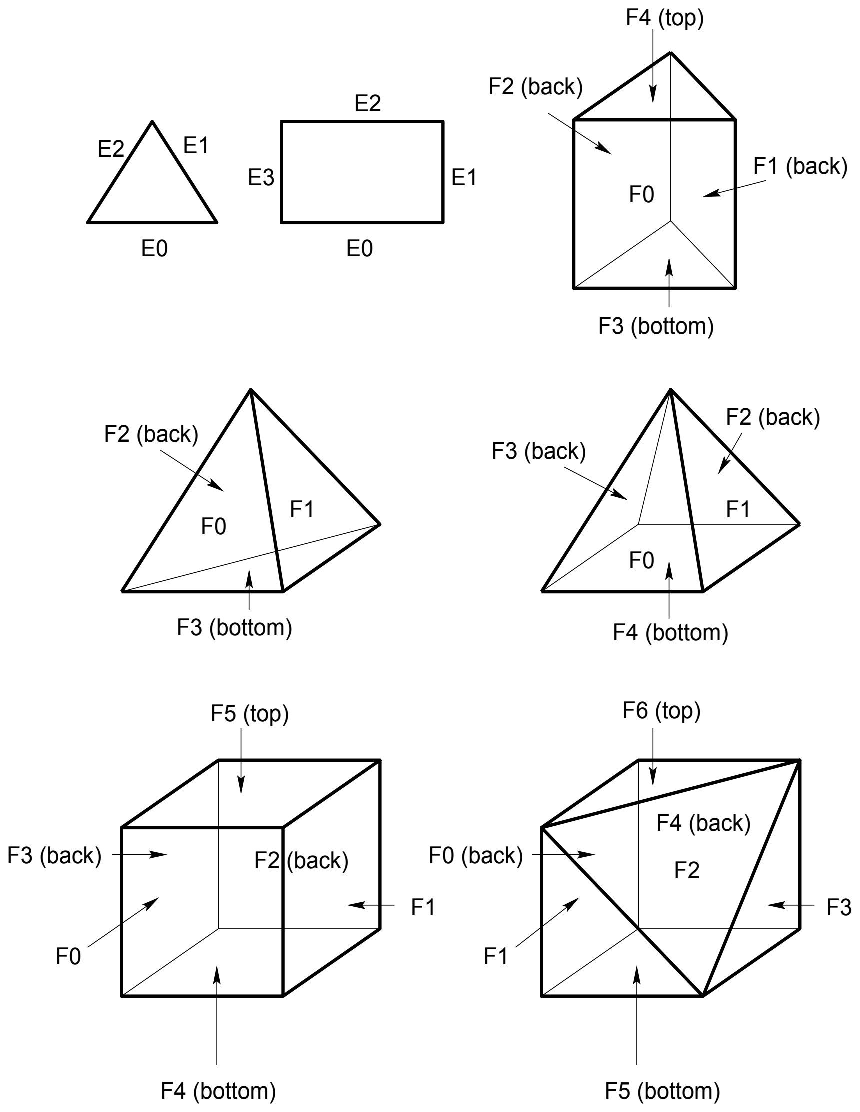

Figure 6.6 Edge and face numbering

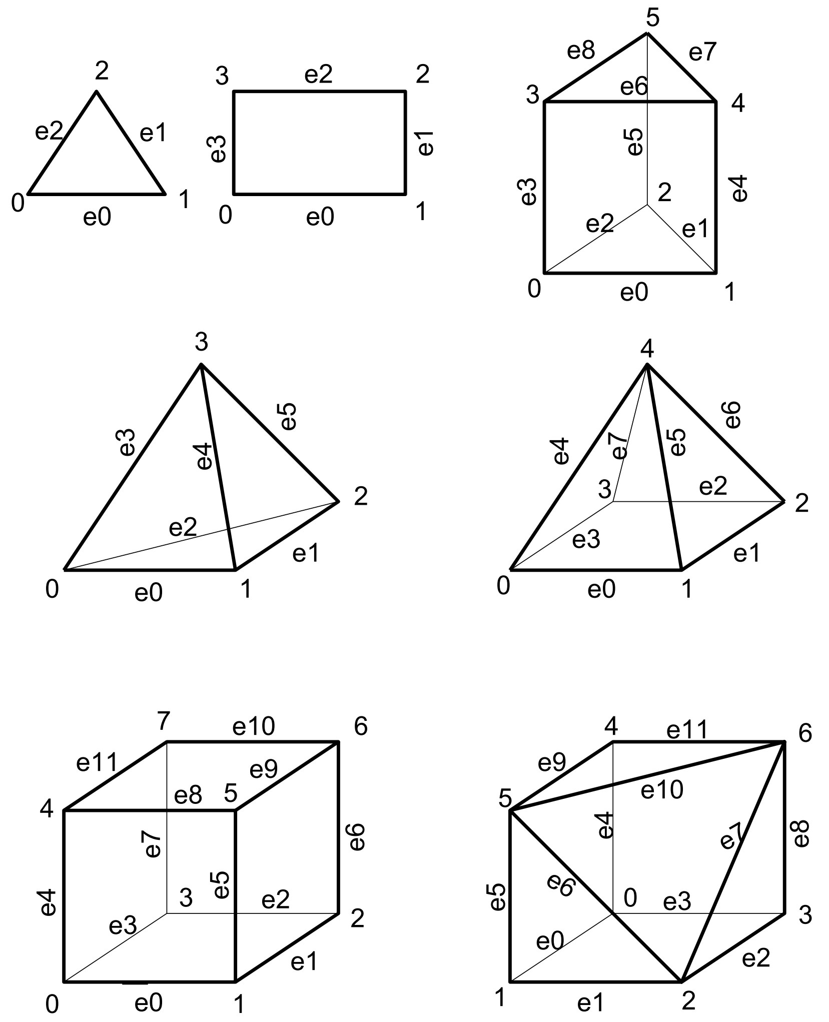
0 1 2 e0 e1 e2
0 1 2 3 e0 e1 e2 e3
0 1 2 3 4 5 e0 e1 e2 e3 e4 e5 e6 e7 e8
0 1 2 3 e0 e1 e2 e3 e4 e5
0 1 2 3 4 e0 e1 e2 e3 e4 e5 e6 e7
0 1 2 3 4 5 6 7 e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11
0 1 2 3 4 5 6 e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11

Figure 6.7 Edge and vertex numbering

2.12 Dataset

The `dataset` files add scalar or vector values to the nodes, edges, faces, elements, or regions of a `grid` file. The data *location* is currently limited to `vertex`, `edge`, `face`, `element`, or `region`. Two additional location codes are used to store Voronoï data: `vertex_element` and `edge_element`. These codes are only understood by DESSIS.

`dimension` gives the set number of data values per location (1 for scalars, > 1 for vectors). The `dimension` entry has a different meaning than the geometry dimension given in the `Info` block. The type of a dataset is currently either a `scalar` or `vector`. More types (`matrix`, `array`) will be added later.

`validity` gives the list of regions in which the dataset is defined. For consistency, data values should only be defined for the entries belonging to these validity regions.

NOTE The DF–ISE library provides some functions to access the values of a dataset for a given validity region. For the moment, this can only be used with datasets whose location is `vertex`.

Discontinuous datasets are a matter of interpretation of the data file format. Several datasets with the *same* quantity can be present in a single data file. Each instance of the dataset should be defined for a different set of regions (including the boundary points of a region). The entire dataset can, therefore, be discontinuous across boundaries. Because of the interpretational nature of this definition of discontinuous datasets, different tools may handle data files with such datasets differently. Refer to specific manuals for detailed descriptions of individual tools.

Data values are given in the order predefined by the `location` entry. When vectors are stored, all components are given per item (`vertex`, `edge`, . . .) in the `Values` block.

The order of appearance of `Dataset` blocks must correspond to the `datasets` entry list of the `Info` block. Functions are references to the `function` blocks stored in the property file. Several datasets with the same name are allowed (with different validity regions, this may be used to define data discontinuities).

```
DF-ISE text | binary

Info {
  version     = 1.0
  type        = dataset
  dimension   = 1 | 2 | 3
  nb_vertices = <int>
  nb_edges    = <int>
  nb_faces    = <int>
  nb_elements = <int>
  nb_regions  = <int>
  datasets    = [ "<dataset1>" "<dataset2>" ... ]
  functions   = [ <function1> <function2> ... ]
}

Data {

  Dataset ("<dataset1>") {
        function  = <function1>
        type      = scalar | vector
        dimension = 1 (1 for scalar, > 1 for vectors or arrays)
        location  = vertex | edge | face | element | region
        validity  = [ "<region0>" "<region1>" ... ]
        Values (<nb_values>) { <value0> <value1> ... }
  }

  Dataset ("<dataset2>") {
        function  = <function2>
  ...
}
```

2.13 XYPlot

The `xyplot` files are used to store curves. The property list gives additional information for each data column of the file. The x-axis is not always in the first column, since the user can freely select the dataset mapped on the x-axis. For undefined values (discontinuities in the curve), the character ‘u’ is used as a placeholder:

```
DF-ISE text | binary

Info {
  version    = 1.0
  type       = xyplot
  datasets   = [ "<dataset1>" "<dataset2>" ... ]
  functions  = [ <function1> <function2> ... ]
}

Data {
  <v0_d1> <v0_d2> ... (one value for each data set)
  <v1_d1> <v1_d2> ... (one value for each data set)
  ...
}
```

2.14 Examples

This section provides examples of the DF–ISE file format.

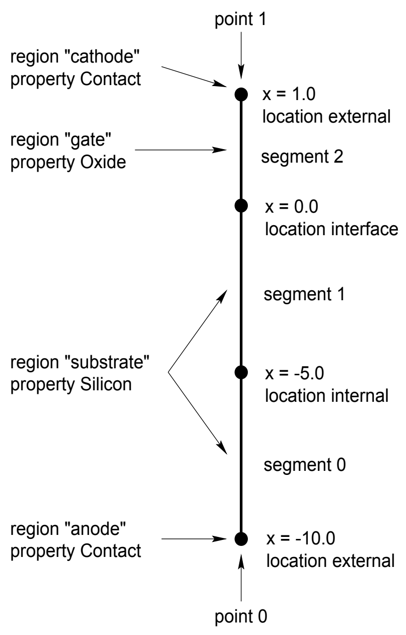

Figure 6.8 One-dimensional grid

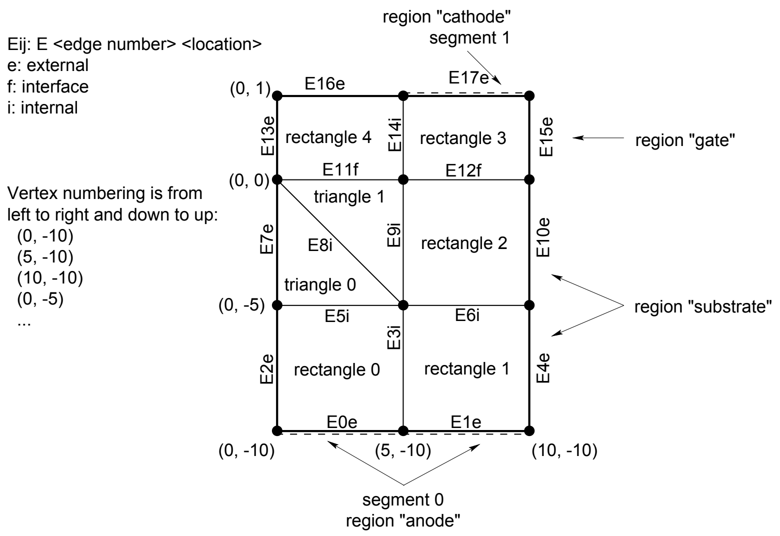

Figure 6.9 Two-dimensional grid

.

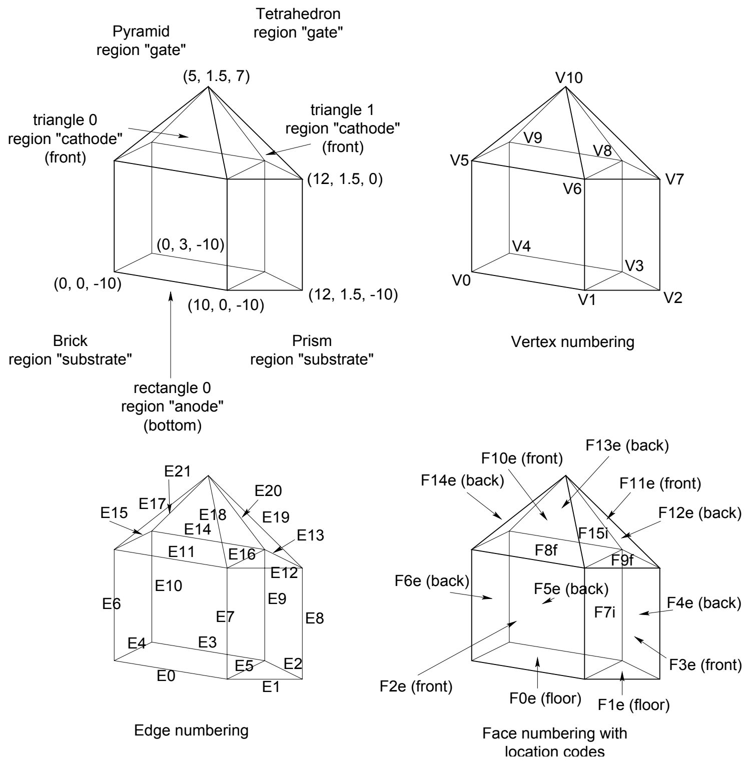

Figure 6.10 Three-dimensional grid