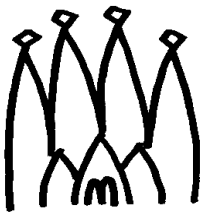

Solids

Version 6.0 (March'**2k+1**)

Ivan Belyaev[1]
(*ITEP, Moscow*)

[1]E-mail: Ivan.Belyaev@itep.ru

Table of Contents

Chapter 1: `ISolid` interface

All solids implements an abstract interface **`ISolid`**. This interface allows the retrieval of all information which are common for all types of solids. One can ask it the following questions

- **`const std::string& name() const`**
  It returns the name of the solid. This name is not required to be unique or specific. It is used only for the user convenience.
- **`const std::string typeName() const`**
  It returns the specific type name for the given type of the solid. It could be used (in parallel with **`dynamic_cast`** and/or **`typeid`**) for determination of the concrete type of the solid.
- **`bool isInside( const HepPoint3D& LocalPoint ) const`**
  It checks if the **`LocalPoint`** is inside of the given solid or not. Point should be given in a local reference system of the solid!
- **`const ISolid* cover() const`**
  It returns the C++ pointer to the "simplified" solid, which can be used for different approximations and simplifications.
- **`const ISolid* coverTop() const`**
  It returns the C++ pointer to "the most simplified" solid, which can be used for different approximations and simplifications. Normally it is just a box.
- an addition an overloaded output operator to **`std::ostream`** is implemented for objects of type **`const ISolid&`** and **`const ISolid*`**.
- The special (technical) method is defined for searching the intersection points of the line and the given solid.

```
/// tick type definition
typedef double             Tick  ;
/// tick container definition
typedef std::vector<Tick>  Ticks ;
virtual inline  unsigned int intersectionTicks (
        const HepPoint3D & Point  ,
        const Hep3Vector & Vector ,
        ISolid::Ticks    & ticks  ) const;
```

  Line is assumed to be parametrised in the local reference system of the given solid as $\vec{\mathrm{r}}(\mathrm{t}) = \vec{\mathrm{p}}_0 + \vec{\mathrm{v}}\mathrm{t}$, where $\vec{\mathrm{p}}_0$ corresponds to **`const HepPoint3D& Point`** and $\vec{\mathrm{v}}$ corresponds to **`const Hep3Vector& Vector`**. The output container **`Ticks& ticks`** is ordered. The returned value is just the number of intersection points (size of the output container).

- The same as previous method, but only ”ticks” between specified minimal and maximal values are returned:

```
virtual inline  unsigned int intersectionTicks (
        const HepPoint3D & Point   ,
        const Hep3Vector & Vector  ,
        const Tick       & tickMin ,
        const Tick       & tickMax ,
        ISolid::Ticks    & ticks   ) const;
```

Please, pay some attention that **all** methods from **`ISolid`** interface are constant methods, which is quite favourable form the safety consideration.

All specific questions concerning a specific shapes and parameters of the solids, pointed by an **`const ISolid*`** pointer, should be addressed to a specific classes after casting performed. For casting to a specific solid type there exist 3 possibility

- sequential casting via **`dynamic_cast`** to all hypothesis till successful casting

```
void f( const ISolid* solid )
 {
    const SolidBox* box =
     dynamic_cast<const SolidBox*>(solid); }
    if( 0 != box )
      { /* do what you want with box  */ }
    else
    {
     const SolidTubs* tubs =
       dynamic_cast<const Solidtubs*>(solid); }
     if( 0 != tubs )
       { /* do what you want with tubs */ }
    }
 }
```

- determine the exact type of solid before casting via **`dynamic_cast`** using **`typeid`** facility:

```
void func( const ISolid* solid )
  {
    if( typeid(*solid) == typeid(SolidBox) )
      {
       const SolidBox* box =
```

```
          dynamic_cast<const SolidBox*>(solid);
        /* here one can do something with box */
       }
     else if( typeid(*solid) == typeid(SolidTubs) )
       {
        const SolidTubs* tubs =
         dynamic_cast<const SolidTubs*>(solid);
        /* here one can do something with tubs */
       }
  }
```

- determine the exact type of solid before casting via **`dynamic_cast`** using **`ISolid::typeName()`** method from **`ISolid`** interface:

```
void func( const ISolid* solid )
  {
    if(      solid->typeName() == "SolidBox" )
      {
       const SolidBox* box =
         dynamic_cast<const SolidBox*>(solid);
       /* here one can do something with box */
      }
    else if( solid->typeName() == "SolidTubs" )
      {
       const SolidTubs* tubs =
         dynamic_cast<const SolidTubs*>(solid);
       /* here one can do something with tubs */
      }
  }
```

The class diagram for all concrete solids implemented in **DETDESC** version **`v6`** is shown in figure 1.1.

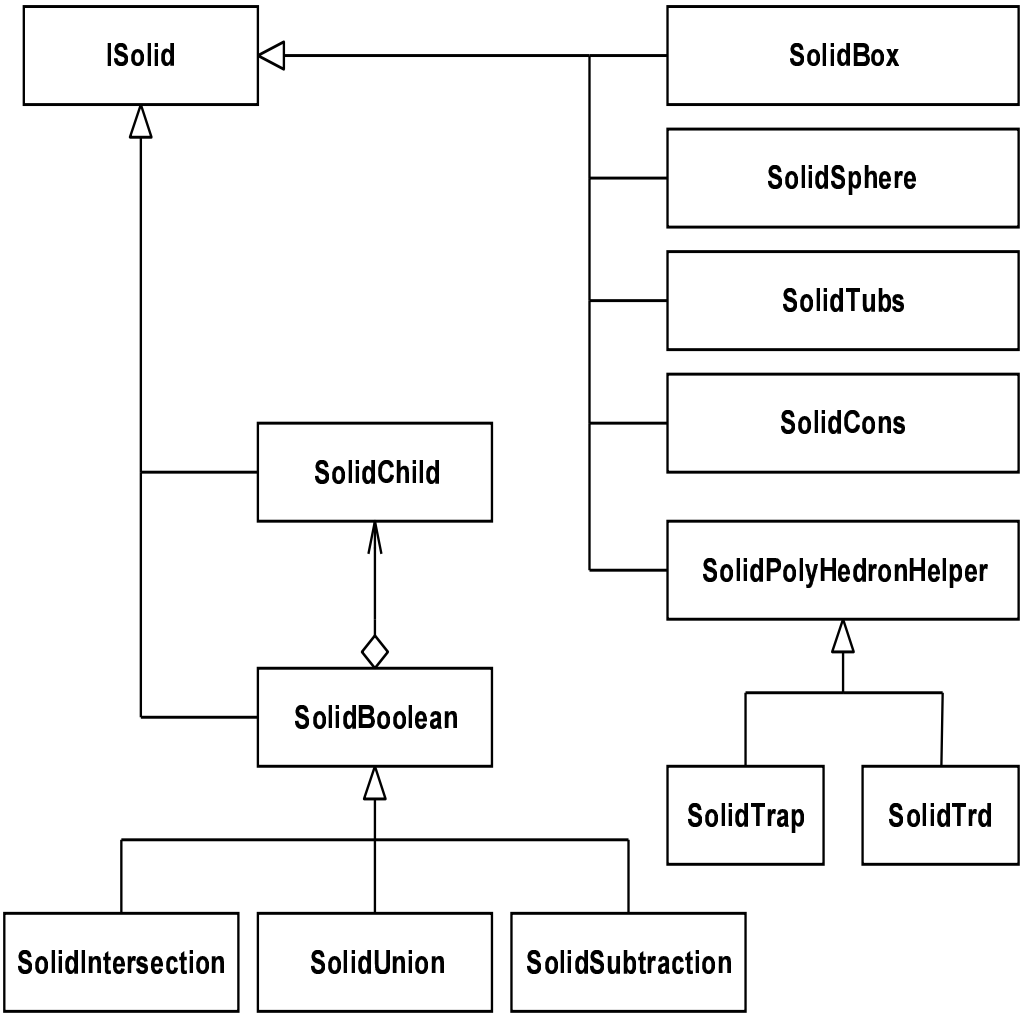

Figure 1.1: The class diagram for all concrete solids implemented in **DETDESC** version **v6**

Chapter 2: Primitive solids

Currently 5 types of ”primitive” solids are implemented. The most frequently solids (”shapes”) were chosen from shapes available in **GEANT3** and **GEANT4** packages. There are no any principal limitations and if one needs some additional shapes from **GEANT3** and/or **GEANT4** tool kits they could be implemented on demand.

For all solids, there is no any ”setters”. One is able to set the parameters only in the creation of the objects via parameters in constructor.

Further only specific methods (or specific details of common method) are described for all available solids.

2.1 Boxes

This shape is described in class **`SolidBox`**[1]. This is the most primitive shape. Essential features of this class are:

- Constructor requires name of the solid and 3 positive[2] parameters to be given. All parameters are mandatory. There is no any default values for them. Keeping in mind that a significant number of physicists have some experience with geometry and shapes in **GEANT3** a half-sizes half size in x-,y- and z- directions) of parameters were chosen as base parameters for constructor.
- Since further ”simplification” of this shape looks not reasonable, both **`SolidBox::cover()`** and **`SolidBox::coverTop()`** methods just returns **`this`** pointer. It means that ”simplified box” is exactly the same box.

XML description of this shape looks like:

```
<box name  = 'Box_name'
     sizeX = '0.4*cm'
     sizeY = '0.5*cm'
     sizeZ = '0.6*cm' />
```

2.2 Simple trapezoids

Trapezoids are implemented in the class **`SolidTrd`**[3]. Essential features of this class are:

- Constructor requires name of the solid and 5 positive[4] parameters to be given. All parameters are mandatory. There is no any default values for them. They are - half length in z-direction, half sizes on x- and y-directions at minimal value of z and at maximum value of z.

[1] It corresponds to **`'BOX '`** shape from **GEANT3** package

[2] Non-positive parameters throw exception via **`SolidException`** class

[3] It corresponds to **`'TRD '`** shape from **`Geant3`** package

[4] Non-positive parameters throw exception via **`SolidException`** class

- Simplification of **`SolidTrd`** is done by a 2 steps. If sizes in x-direction are not equal to sizes in y-direction, the ”simplification” is just the smallest ”symmetric” trapezoid which contains the given trapezoid. If the trapezoid is already symmetric, the ”simplification” is the minimal box, which contains the given symmetric trapezoid.

XML description of this shape looks like:

```
<trd name   = 'Trd_name'
     sizeZ  = '100*cm'
     sizeX1 = '10*cm'
     sizeY1 = '12*cm'
     sizeX2 = '13*cm'
     sizeY2 = '14*cm' />
```

2.3 Tube segments

Tube segments are implemented in the class **`SolidTubs`**[5]. Essential features of this class are:

- Constructor requires name of the solid, the positive half-length of the tube (no default value), the positive outer radius of the tube (no default value), the non-negative inner radius of the tube (default value is **`0`**), start of phi angle (in radians, with default value **`0*degree`**) and the size (non-negative) in phi (in radians, default value is **`360*degree`**) and the parameter with describes the covering model (default value is **`0`**)

- Simplification of **`SolidTubs`** is under the control of the special parameter. The default simplification is done according to the following schema:

  1. simplification for tube segment is tube (no gaps in ϕ)
  2. simplification for tube is cylinder
  3. simplification for the cylinder is the box

  Such schema looks quite natural for long tube segments with small gaps in ϕ. But it looks unnatural for e.g. wafers of vertex detector. Alternative schema is implemented especially for such cases:

  1. simplification for tube segment is cylinder segment (set inner radius to zero)
  2. simplification for cylinder segment is the cylinder itself (no gaps in ϕ)
  3. simplification for the cylinder is the box

  Other simplification schemas can be easily implemented on demand.

XML description of this shape looks like:

[5] It corresponds to **`'TUBS'`** shape from **GEANT3** package

```
<tubs name          = 'Tubs_name'
      sizeZ         = '100*mm'
      outerRadius   = '10*cm'
      innerRadius   = '0*mm'
      startPhiAngle = '0*degree'
      deltaPhiAngle = '360*degree' />
```

Attributes **`innerRadius`**, **`startPhiAngle`** and **`deltaPhiAngle`** could be omitted.

2.4 Conical tube segments

Conical tube segments are implemented in the class **`SolidCons`**[6]. Essential features of this class are:

- Constructor requires name of the solid, the positive half-length of the tube (no default value), the positive outer radius of the tube at minimal z-value (no default value), the positive outer radius of the tube at maximal z-value (no default value), the non-negative inner radius of the tube at minimal z-value (default value is **`0`**), the non-negative inner radius of the tube at maximal z-value (default value is **`0`**), start of phi angle (in radians, with default value **`0*degree`**) and the size (non-negative) in phi (in radians, default value id **`360*degree`**) and the "parameter with describes the covering model (default value is 0)

- Simplification of **`SolidCons`** is under the control of the special parameter. The default simplification is done according to the following schema:

  1. simplification for conical tube segment is conical tube (no gaps in ϕ)
  2. simplification for conical tube is the cone (set inner radius to null)
  3. simplification for the cone is the cylinder

  Such schema looks quite natural for long conical tube segments with small gaps in ϕ. Alternative schema is implemented especially for such cases:

  1. simplification for conical tube segment is cone segment (set inner radius to zero)
  2. simplification for cone segment is cone itself (no gaps in ϕ)
  3. simplification for the cone is the cylinder

  Other simplification schemas can be easily implemented on demand.

XML description of this shape looks like:

[6] It corresponds to **`'CONS'`** shape from **GEANT3** package

```
<cons name          = 'Cons_name'
      sizeZ         = '100*mm'
      outerRadiusPZ = '100*mm'
      outerRadiusMZ = '200*mm'
      innerRadiusPZ = '0*mm'
      innerRadiusMZ = '0*mm'
      startPhiAngle = '0*degree'
      deltaPhiAngle = '360*degree' />
```

Attributes **`innerRadiusPZ`**, **`innerRadiusMZ`**, **`startPhiAngle`** and **`deltaPhiAngle`** could be omitted.

2.5 Sphere segments

Sphere segments are implemented in the class **`SolidSphere`**[7]. Essential features of this class are:

- Constructor requires name of the solid, the positive half-length of the tube (no default value), the positive outer radius of the sphere(no default value), the non-negative inner radius of the sphere (default value is **`0`**), start of ϕ angle (in radians, with default value **`0*degree`**) and the size (non-negative) in ϕ (in radians, default value is **`360*degree`**) start of θ angle (in radians, with default value **`0*degree`**) and the size (non-negative) in θ (in radians, default value is **`180*degree`**) and the parameter with describes the covering model (default value is **`0`**)

- Simplification of **`SolidSphere`** is under the control of the special parameter. The default simplification is done according to the following schema:

  1. simplification for sphere segment is sphere segment with no gaps in θ
  2. simplification for sphere segment with no gaps in θ is just the sphere itself
  3. simplification for the sphere is sphere with inner radius equals to null
  4. simplification for the sphere with inner radius equals to null is the box

  Alternative schema is implemented especially for such cases:

  1. simplification for sphere segment is sphere segment with null inner radius
  2. simplification for sphere segment with inner radius is sphere segment with no gap in ϕ
  3. simplification for sphere segment with no gap in ϕ is sphere itself
  4. simplification for sphere is the box

[7] It corresponds to **`'SPHR'`** shape from **GEANT3** package

Other simplification schemas can be easily implemented on demand.

XML description of this shape looks like:

```
<sphere name            = 'Sphere_name'
        outerRadius     = '100*cm'
        innerRadius     = '0*mm'
        startPhiAngle   = '0*degree'
        deltaPhiAngle   = '360*degree'
        startThetaAngle = '0*degree'
        deltaThetaAngle = '180*degree' />
```

All attributes, but **`outerRadius`** could be omitted.

2.6 The most general trapezoid

the most general trapezoids are implemented in the class **`SolidTrap`**[8]. The faces perpendicular to the **`z`** planes are trapezia, and their centres are not necessarily on a line parallel to the **`z`** axis.

From 11 parameters described below, only 9 are really independent - a check for planarity is made in the calculation of the equation for each plane. If the planes are not parallel, a call to **`SolidException`** is made. Parameters are:

`pDz` Half-length along the **`z`**-axis

`pTheta` Polar angle of the line joining the centres of the faces at **`+/-pDz`**

`pPhi` Azimuthal angle of the line joining the centre of the face at **`-pDz`** to the centre of the face at **`+pDz`**

`pDy1` Half-length along **`y`** of the face at **`-pDz`**

`pDx1` Half-length along **`x`** of the side at **`y=-pDy1`** of the face at **`-pDz`**

`pDx2` Half-length along **`x`** of the side at **`y=+pDy1`** of the face at **`-pDz`**

`pAlp1` Angle with respect to the **`y`**-axis from the centre of the side at **`y=-pDy1`** to the centre at **`y=+pDy1`** of the face at **`-pDz`**

`pDy2` Half-length along **`y`** of the face at **`+pDz`**

`pDx3` Half-length along **`x`** of the side at **`y=-pDy2`** of the face at **`+pDz`**

`pDx4` Half-length along **`x`** of the side at **`y=+pDy2`** of the face at **`+pDz`**

[8] It corresponds to **`'TRAP'`** shape from **GEANT3** package

pAlp2 Angle with respect to the **y**-axis from the centre of the side at **y=-pDy2** to the centre at **y=+pDy2** of the face at **+pDz**

XML description of this shape does not yet exist.

Chapter 3: Boolean solids

Boolean solids represents a new way of construction of complicated shapes from simple one.

The essential features of all boolean solids implemented within **GAUDI** framework are

- all of them are inherited from class **`SolidBoolean`**, which implements **`ISolid`** interface.

- class **`SolidBoolean`** has a notion of ”main” (or ”first”) solid and a (optionally empty) list of ”child solids”. This ”main”(”first”) solid defined the overall reference system. Each ”child solid” is placed with respect to this reference system. Pointers to the existing solids (either primitive or boolean) are used in constructor of boolean solid (where the valid pointer to the ”main”(”first”) solid is mandatory) and in further boolean operation with ”child solids”.

- the greats simplification in the current implementation of **`SolidBoolean`** class is that it simply delegates methods **`SolidBoolean::cover()`** and **`SolidBoolean::coverTop()`** to its ”main” (or ”first”) solid, instead of exact but tedious calculation of the covering.

- all implemented boolean solids - subtraction, unification, and intersection differ only in the implementation of method **`SolidBoolean::isInside(const HepPoint3D& LocalPoint) const`**.

3.1 Subtraction

Subtraction of solids is implemented in class **`SolidSubtraction`**. The essential features of this class are:

- constructor gets the name of the solid and mandatory valid pointer to the solid (either primitive or boolean) to be used as ”main”(”first”) solid.

- **`SolidSubtraction::isInside(const HepPoint3D& LocalPoint)`** method implemented such a way that local point ”is inside” of the given solid if it is inside of ”main” (”first”) solid, and it is not inside any child solid.

- child solid could be added using method **`SolidSubtraction::subtract (ISolid* child, HepTransform3D* mtrx)`**.

For purposes of proper visualisation and **GEANT4** tracking it is recommended to avoid common faces between ”main” solid and subtracted child.

3.2 Union

Unification of solids is implemented in class **`SolidUnion`**. The essential features of this class are:

- constructor gets the name of the solid and mandatory valid pointer to the solid (either primitive or boolean) to be used as ”main”(”first”) solid.
- **`SolidUnion::isInside(const HepPoint3D& LocalPoint)`** method implemented such a way that local point ”is inside” of the given solid if it is inside of ”main” (”first”) solid or it is inside of any child solid.
- child solids could be added via method **`SolidUnion::unite(ISolid* child , HepTransform3D* mtrx)`**.

3.3 Intersection

Subtraction of solids is implemented in class **`SolidIntersection`**. The essential features of this class are:

- constructor gets the name of the solid and mandatory valid pointer to the solid (either primitive or boolean) to be used as ”main”(”first”) solid.
- **`SolidIntersection::isInside(const HepPoint3D& LocalPoint)`** method implemented such a way that local point ”is inside” of the given solid if it is inside of ”main” (”first”) solid and it is inside each child solid.
- child solids could be added via method **`SolidIntersection::intersect (ISolid* child , HepTransform3D* mtrx)`**.