Optimising and Extending the Geometrical Modeller of a Physics Simulation Framework

Thesis for the Degree of Master of Engineering

Technical University of Budapest
Faculty of Electrical Engineering and Informatics

Péter Urbán

Supervisors:

Dr. András Pataricza
Department of Measurement and Information Systems

Jari Sulkimo
CERN European Laboratory for Particle Physics
Information Technology Division

May 20, 1998

Abstract

The design of highly complex particle detectors used in High Energy Physics involves both CAD systems and physics simulation packages like GEANT4. GEANT4 is able to exchange detector geometries with CAD systems, conforming to the Standard for the Exchange of Product Model Data (STEP); Boundary Representation (B-Rep) models are transferred. Particle tracking is performed in these models, requiring efficient and accurate intersection computations from the geometrical modeller. The results of extending and optimising the modeller of GEANT4 form the contents of this thesis. Swept surfaces: surfaces of linear extrusion and surfaces of revolution have been implemented. The problem of classifying points on surfaces bounded by curves as being inside or outside has been solved. These tasks necessitated the extension and optimisation of code related to curves and lead to a re-design of this code. Emphasis was put on efficiency and on dealing with numerical errors. The results will be integrated into the upcoming beta test release of GEANT4.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

CERN, the European Laboratory for Particle Physics, is one of the world's largest scientific research laboratories. An early European joint venture, CERN was founded in 1954 and straddles the French-Swiss border west of the city of Geneva. CERN's nineteen member states provide the budget (870.1 million Swiss francs in 1997) in proportion to their national revenues. Hungary has been a member state since 1992. Currently more than 6500 users, over half of the planet's experimental high-energy physicists, carry out fundamental research at CERN. This user community represents 500 universities and over 80 nationalities.

CERN's business is pure research — studying Nature's tiniest building blocks, the fundamental particles, to find out how our world and the Universe work. The energy densities reached in head-on collisions of particles accelerated in CERN's machines approach those which may have prevailed immediately after the 'Big Bang', and are sufficient to create the elementary particles which populated the early universe. Detectors, built around the collision points, record the brief existence of these particles, re-enacting moments in the evolution of the early universe.

CERN hosts the Large Electron Positron Collider (LEP), the largest particle collider in the world with an accelerator ring 27 km in circumference, buried about 100 m underground. The tunnel will be reused for the successor accelerator, the Large Hadron Collider (LHC), starting operation in 2005. The LHC will be the only machine capable of addressing problems far beyond today's frontiers of high energy physics. The technological challenges of the LHC demand breaking new ground in super-conductivity, high-speed electronics, cryogenics, computing and networking, vacuum technology, material science and many other disciplines.

Particle detectors are large and highly complicated devices (Figure 1.1). Each as high as a five-storey building, they are built like a Russian doll, with one module fitting snugly inside the other around the beam collision point at the centre. Each module, packed with state-of-the-art technology, is custom-built to do a special observation job before the particles fly outwards to the next layer. Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM) is used for engineering and manufacturing these complex machines.

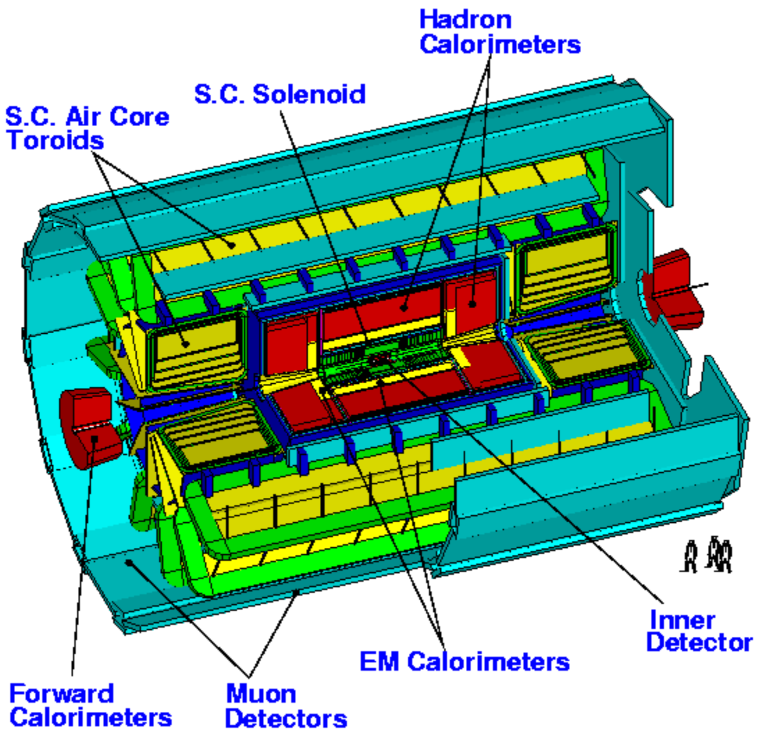


Figure 1.1: Layout of ATLAS detector, a proposed experiment on the LHC. Note the height of the persons around the detector.

The physics response of the detectors is normally estimated and optimised with the aid of simulation packages like GEANT3 [App93] or GEANT4 [G$^+$95, G$^+$97a]. GEANT4 is a world-wide research and development project, aiming at the creation of the successor to an old Fortran simulation framework (GEANT3), completely redesigned in C++, using an object oriented analysis and design methodology. Of course, one needs to describe detectors for GEANT4; the most straightforward way is importing models directly from CAD systems. This approach is hoped to speed up the overall detector design process and make it consistent on both the engineer's and the physicist's side. The ISO Standard for the Exchange of Product Models[STEP1] provides a common method for model data transfer.

Since most modern CAD systems use Boundary Representation (B-Rep), Geant4 has to include B-Rep data structures and implement efficient particle tracking in B-Rep models — efficiency is a crucial requirement of the whole of Geant4. Geometrical objects have to provide the tracking scheme with some key functionality. This includes bounding box and polyhedron calculation, and computing the intersection point of a ray and an arbitrary surface. The efficiency of these implementations has a very significant affect on the overall performance of tracking.

The development effort of this Geant4 sub-domain has been going on for three years[Sul95, Vuo96]. The results of extending and optimising it form the contents of this thesis. The main areas of development are as follows:

- Swept surfaces have been implemented. These are obtained by linear extrusion of a curve along an axis, or revolution of a curve about an axis. The curve being swept may be of an arbitrary STEP type, including the most complicated Non-Uniform Rational B-Splines (NURBS) class.
- In STEP, surfaces can be bounded by any collection of closed curves lying on the surface. One has to be able to find out whether a point on the surface lies within the surface boundary. This is an important step in deciding if the intersection of a particle track with the surface is a real intersection. Bounding curves and surfaces of every STEP type had to be handled.
- The above developments are the heaviest users of code dealing with curves in the B-Rep modelling part. Numerous extensions and optimisations were necessary. These involved the re-design of the curves code.

The rest of the thesis is structured the following way:

- Chapter 2 presents the STEP standard, mainly its parts related to solid modelling with Boundary Representation.
- Chapter 3 introduces the Geant4 framework, with emphasis on the geometry, especially B-Rep code.
- Chapters 4, 5 and 6 describe the process and the results of extending and optimising the B-Rep code. Curves come first, followed by swept surfaces and curve bounded surfaces, since the latter topics are based heavily on the former one.

- Chapter 7 revisits the main results and the status of implementation and integration into the beta release of GEANT4. It also highlights further extension directions and optimisation possibilities of the B-Rep modelling code.

I would like to express my sincere gratitude to Dr. András Pataricza and Jari Sulkimo, my thesis advisors, for their invaluable encouragement and guidance. I owe my best thanks to my colleagues, especially Anthony Osborne, Simone Giani and John Apostolakis for their valuable comments on preliminary versions of this thesis; special thanks to Tony for revising my English. Especially, I would like to thank to Simone and Tony for making my one year long stay at CERN possible.

Chapter 2

Boundary Representation with STEP

2.1 Structure of the standard

STEP, the STandard for the Exchange of Product model data[STEP1], is a new international standard for representing and exchanging product model information between CAD systems. Its advantages over its numerous predecessors[Vuo96] are that it uses an object definition language, EXPRESS [STEP11], for the representation of data, and that it is designed to cover all aspects and the whole life cycle of products: geometry, topology, tolerances, materials, production context, etc. . The first parts were approved by the International Standards Organisation in 1992, and development has continued ever since. All major CAD system vendors already support STEP, thus it was a straightforward choice for Geant4 to implement the exchange of particle detector setups via STEP.

The standard is organised as a collection of parts, each of which falls into one of the following categories:

Overview and fundamental principles Defines the basics[STEP1].

Description methods The EXPRESS object definition language[STEP11], used to represent product information. Automated translation to object oriented languages is possible[Lib93]. See Figure 2.1 for a code example.

Implementation methods The physical representation of exchange data.

```
ENTITY line
  SUBTYPE OF (curve);
  pnt : cartesian_point;
  dir : vector;
WHERE
  WR1: dir.dim  = pnt.dim;
END_ENTITY;
```

Figure 2.1: EXPRESS specification of the `line` STEP entity.

Data can be exchanged using ASCII files or via an application programming interface, which is accessible from C++, C and IDL.

Conformance testing A methodology and framework.

Integrated resources Product information common for different application areas. A few examples: Fundamentals of Product Description and Support, Geometric and Topological Representation[STEP42], Representation structures[STEP43], Materials and Shape Tolerances.

Application Protocols Product information specific for the application areas. An application protocol is a combination of a suitable part of integrated resources with additional constraints and application specific concepts. [STEP203] is the de facto standard, supported by all important CAD system vendors. At the time of writing, several German car manufacturers are pushing heavily for general acceptance of [STEP214], which includes a great deal more of the integrated resources.

Abstract test suites Test cases for an application protocol to support the conformance requirements.

2.2 Representation of solids

Integrated Generic Resources for geometric and topological representation are described in [STEP42]. Their primary application is for explicit representation of the shape or geometric form of a product model – in the context of physics simulation this means solids which are parts of a detector setup. The shape representation has been designed to facilitate stable and efficient communication when mapped to a physical file, and a general philosophical motivation behind is to support all the frequently used ways to describe geometry, in order not to restrict the choices of the designer. Appendix C contains class diagrams of the geometry and topology related part of [STEP203], i.e. the parts taken from [STEP42].

[STEP42] is divided into three main parts:

geometry This part is exclusively the geometry of parametric surfaces and curves, both two and three dimensional. The entities might even have, and often do have, characteristics undesirable for product description: they might be infinite, for example.

topology This part is concerned with connectivity relationships between objects rather than with their precise size and shape. Topological entities may have a geometry associated with them, and they may constrain the geometry, thereby making it suitable for product description.

solid model This part provides resources for the communication of data describing the precise size and shape of three dimensional solid objects. This complete representation of shapes in many cases includes both geometric and topological data.

Geometry and topology will be described in more detail in the subsequent sections. This section introduces the available solid models.

The solid model part includes the two classical types of solid models, *constructive solid geometry* (CSG) and *boundary representation* (B-Reps). Also included are a collection of entities providing less complete geometric and topological information than the *full* CSG or B-Rep models. The use of these entities is appropriate for communication with systems whose capabilities differs from that of solid modelling systems.

The CSG models are represented by their component primitives — cones, spheres and blocks, to name a few — and the sequence of Boolean operations *union*, *intersection* and *difference* used in their construction. CSG models do not form part of the application protocol[STEP203].

B-Rep models consist of *solids* represented by surfaces and their boundaries. Solids must be finite and connected; they might contain holes as well. They correspond to `manifold_solid_breps` in STEP. There are several other kinds of surface based models, dropping some restrictions of B-Rep. A more rigorous definition of B-Rep models and their elements is the subject of the two subsequent sections.

A variety of systems only allow boundary type solid representations in which all faces are planar polygons. Such a representation is often used for rendering 3D models, e.g. in the popular OpenGL library[WNDO97]. Such models may be represented by the general `manifold_solid_brep`, but STEP provides a more compact representation called `facetted_brep`.

Other surface based models are also available. These collections of surfaces do not necessarily form a proper solid. However, CAD systems often use these models for communication of proper solids. The STEP entities are called `shell_based_surface_model` and `face_based_surface_model`.

The remaining STEP models do not provide a complete description of three dimensional solids or make it difficult to get the necessary information out of the exchange file. Thus they are not suitable for describing particle detectors and need not be supported by GEANT4. Entities include different kinds of wire-frame models and `geometric_set`, geometric information without topology.

2.3 Geometry

The subject of the geometry schema is the geometry of parametric curves and surfaces. It is described in [STEP42], Clause 4.

For easier readability, STEP entity names will be typeset the following way throughout the remaining chapters: `step_entity`. Their EXPRESS names are used[STEP11].

Geometric entities are all defined in a *right-handed rectangular Cartesian coordinate system* with the same units on each axis. A common schema has been used for the definition of both *two dimensional* (2D) and *three dimensional* (3D) geometry. Detectors are 3D, hence so are GEANT4 models, but 2D entities are used for some computations and entities in the parameter space of a surface. `geometric_representation_item` is the base class to all geometric entities[STEP43]; the coordinate system and the dimensionality come from this class.

Many of the geometric entities provide the capability to define an item of geometry in more than one way, e.g. a curve on a surface can be defined both in the 2D parameter space of the curve and in 3D space. One of the representations must be nominated the *master representation*; all characteristics for an entity must be derived from this one. This rule acknowledges the impracticality of ensuring and checking if multiple representations are indeed identical; consider the previous example with a curve and a surface from the most complicated Non-Uniform Rational B-Spline (NURBS) class.

The alternative representations thus cannot be trusted, therefore they should not be used. The same holds for attributes which are there for information only, e.g. information if a B-Spline curve is self-intersecting or not.

The subsequent sections introduce *points*, *curves* and *surfaces*. The intent is to give an overview; precise definitions will follow in the chapters which need them. Only those entities occur which are important in practice or helpful in demonstrating a concept.

Mathematical notions such as length and connectedness will not be defined here; their common sense meaning is hoped to be enough for understanding. Definitions, which are intuitive rather than mathematically rigorous, can be found in [STEP42], Clause 3.

2.3.1 Points

A `point` is a location in 2D or 3D Cartesian coordinate space. It can be given in this space (`cartesian_point`) or in the parameter space of a curve (`point_on_curve`) or a surface (`point_on_surface`). Degenerate curves (e.g. `degenerate_pcurve`) are also subclasses of `point`, for a `curve` must have non-zero length.

The subtype `point_replica` is a replica of an existing point. The location is defined by a `cartesian_transformation_operator`, composed of *translation*, *rotation*, *mirroring* and *uniform scaling*. Similar entities exist for curves (`curve_replica`) and surfaces (`surface_replica`).

2.3.2 Curves

A `curve` can be envisioned as the path of a point moving in its coordinate space. A `curve` is connected and has an arc length greater than zero. `curve`s have a well-defined parametrisation which makes it possible to trim the curve or identify points on the curve by parameter value. Curves can be infinite or self-intersecting. See [Far93], a rather comprehensive guide to curves and surfaces.

A `line` is defined by one of its points and its direction. The line is the whole unbounded curve rather than a line segment or a half line. Half lines will be called rays for simplicity.

`conic` curves can be produced by intersecting a plane with a cone. They are defined in a placement coordinate system (`axis2_placement`) by their intrinsic geometric properties. An example for a definition: a `circle` consists of all the points satisfying $x^2 + y^2 - R^2 = 0$, where x and y are coordinates in the placement system and R is the radius. The other subtypes are `ellipse`, `parabola` and `hyperbola`.

The general purpose parametric curve is represented by the `b_spline_curve` entity. This is a general *Non-Uniform Rational B-Spline curve* (NURBS) [BBB87]. All the other basic curves, including conics, can be represented exactly with this entity. Special cases include *uniform and quasi uniform B-splines* and *Bézier curves.*

The other curve types refer to other geometric entities in their definitions:

trimmed_curve Created by taking a selected portion of the associated basis curve. Both end points must be specified, by parametric value and/or geometric position. `trimmed_curve` is analogous to the topological entity `edge`, to be introduced in the next section.

composite_curve A collection of bounded curves joined end to end. The parametrisation is an accumulation of the parametric ranges of the components. The sense of components can be reversed, so that the parameter range maps to an interval on the real line. `composite_curve` is analogous to the topological entity `path`, to be introduced in the next section.

pcurve Lies on a basis surface and is defined in the 2D parameter space of that surface.

intersection_curve The result of intersecting two surfaces. It *must* contain a description as a 3D or a parametric curve. This representation is used everywhere, thus the difficult task of surface-surface intersection need not be implemented in GEANT4.

seam_curve Refers to a surface where the range of a parameter is "wrapped around". On such a surface, the two extremes of that parameter range denote the same curve; e.g. the one parameter of a `cylindrical surface` goes from 0 degrees to 360 degrees, and these two parameter values refer to the same points. This curve is the `seam_curve` of the surface.

offset_curve A curve at a constant distance from a basis curve. The basis curve should have a well defined *tangent direction* at every point; the points of the `offset_curve` are at a fixed distance from that point, in a direction perpendicular to the tangent. In 3D space, a reference direction is needed to define a unique direction for offsetting.

2.3.3 Surfaces

A `surface` can be envisioned as a set of connected points in 3D space which is always locally 2-dimensional[Far93]. It can be infinite, self-intersecting and need not be embeddable into the plane. Surfaces are described in less detail than curves, since for the purposes of this thesis, curves are more important as building blocks.

An `elementary_surface` is one of (unbounded) `plane`, `cylindrical surface`, `conical_surface`, `spherical_surface` or `toroidal_surface`.

`swept_surface`s are constructed by sweeping a curve along another curve. The two subtypes are

surface_of_linear_extrusion A *generalised cylinder* obtained by sweeping a `curve` of any kind in a given direction. The result is an infinite surface.

surface_of_revolution Obtained by rotating a `curve` a *complete* revolution about an axis.

The `b_spline_surface` entity provides the most general capability for the communication of all types of polynomial and rational bi-parametric surfaces. In general, it is a *Non-Uniform Rational B-Spline Surface* (NURBS)[BBB87]. Subtypes represent special cases, such as uniform and quasi uniform B-spline surfaces and Bézier surfaces. All the other basic surfaces can be represented with this entity.

The other surface types refer to other geometric entities in their definitions:

curve_bounded_surface A parametric surface with a curved boundary defined by one or more *boundary curves.* These curves must be closed and might be classified as an *outer bounding curve* and some *inner bounding curves.* They are analogous to the topological entity `face`; a more detailed definition will be given there.

rectangular_trimmed_surface A special case where a rectangle in the *2D parameter space* forms the boundary.

rectangular_composite_surface A rectangular array of surface patches, tiling a connected surface without holes. The parametrisation is an accumulation of the parametric ranges of the components. The patches can be linearly re-parametrised, so that the parameter range maps to a rectangle on the real plane. `rectangular_composite_surface` is a special case of the topological entity `shell`, to be introduced in the next section.

offset_surface This is a surface offset at a constant distance from a basis `surface`. The basis surface should have a well defined *normal direction* at every point; the distance is to be measured along this normal.

2.4 Topology

The topology part is the subject of Clause 5 of [STEP42]. It has its roots in B-Rep solid modelling but can be used in any other application where an explicit method is required to represent connectivity. The focus in this section is on what constraints the topology of a product model imposes on its geometry and what information is added, rather than on presenting the whole of the topology part, along with all its mathematics.

The topological entities have been defined in a hierarchical manner. The basic topological entities in order of increasing complexity are `vertex`, `edge`, `path`, `loop`, `face` and `shell`. The entities will be described below in more detail. Each entity has its own set of constraints and the higher level entity may impose constraints on the lower level entity.

Many of the topological entities have a specialised subtype which enables them to be associated with geometric data; see Table 2.1. The key concept relating geometry to topology is the *domain*. The domain of geometric entities are themselves viewed as sets of points. For topological entities, the domain is the union of domains of the associated geometry — the association is either direct or through the topological components of the entity. E.g. for a `loop`, a sequence of `edges` joined end to end, the domain is the union of the domains of all the `vertexes` and `edges` in the loop, i.e. the union of the associated `points` and `curves`. Whenever a geometrical concept — connectedness, finiteness, etc. — is discussed in relation to an entity, it is understood that the concept applies to the domain of that entity.

Constraints are formulated in terms of domains. Recall that curves and surfaces can have some undesirable properties for solid modelling. E.g. they can be infinite, whereas all manufacturable products are finite, or they can self-intersect, which makes certain computations difficult. Topology can restrict

Topological		**Geometric**
supertype	**subtype**	**entity**
`vertex`	`vertex_point`	`point`
`edge`	`edge_curve`	`curve`
`face`	`face_surface`	`surface`

Table 2.1: Association of topological entities with geometric data in STEP

the curve of surface by bounding it with points and curves, respectively, so that they have none of the undesirable properties.[1] This is the one major "added value" coming from topology.

The other is related to *orientation*. All curves and surfaces have a *tangent direction* and a *surface normal* assigned to almost every point. The tangent direction agrees with the direction of the derivative with respect to the parameter, i.e. the direction of movement towards greater parameter values. The surface normal is the cross product of two partial derivatives with respect to the parameter variables. The closed surfaces used to bound a solid (`closed_shell`) are always orientable; so is the curve boundary of the `face`s of a solid. Orientation defines inside – outside relations, vital for particle tracking with these solids. The majority of topological entities can reverse the directions of tangents and normals of their domains: the *sense* of the curve or surface. Some are only there for this sole purpose.

The most important topological entities are:

vertex The topological construct corresponding to a point. Its domain, if present, is a point in 2D or 3D space. Recall that the domain is represented by the `vertex_point` subtype.

edge corresponds to the connection between two vertices. Its domain is a finite, non-self-intersecting open curve. The bounds of the `edge` are two `vertex`es, which need not be distinct. The edge is oriented by choosing its traversal direction to run from the first to the second `vertex`. The length of the curve segment is greater than zero.

For a lot of topological entities, there exists an oriented version, here `oriented_edge`. These serve the sole purpose to reverse the sense of another topological entity, here an `edge`.

[1]Note that the same can be achieved by using geometric entities like `trimmed_curve` and `curve_bounded_surface`. Another example showing that there are multiple ways to describe a concept in STEP.

path is an ordered collection of edges, such that the start vertex of each edge coincides with the end vertex of its predecessor. Thus paths are connected. The edges do not intersect, except at common vertices.

loop is the topological entity typically used to bound a face lying on a surface. It might be a single point (vertex_loop) or can be a closed, but not necessarily non-self-intersecting curve (edge_loop). Its domain includes its bounds.

face is a topological entity of dimensionality 2 corresponding to the intuitive notion of a piece of surface bounded by loops. Its domain is an oriented, connected, finite surface. The domain shall not have any *handles*, but *holes* are permissible. Hence, a face is embeddable into the plane.

The bounding loops divide the underlying surface into one region inside the face and possibly several regions outside. Consider a point on one of the bounding loops and denote the *topological normal* of the surface by $\mathbf{n}$ and the tangent to the loop by $\mathbf{t}$. The cross product $\mathbf{n} \times \mathbf{t}$ always points towards the *interior* of the face; see Figure 6.1.

A face shall have at least one bound, and the loops shall be distinct and non-intersecting. One of the loops is optionally distinguished as the *outer loop* of the face. If so, it establishes a preferred way of embedding the face domain into the plane. Note that for some types of closed or partially closed surface it might not be possible to identify such an outer bound.

shell is a connected object of dimensionality 2, typically used to bound a region of 3D space.[2] The shell is constructed by joining faces along edges. Its domain is a connected, oriented, finite, non-self-intersecting surface, which may be *closed* or *open*.

A closed_shell serves as a bound for a region in 3D space; can be used to model detector elements. It divides the 3D space into a finite, connected *interior* and an infinite, connected *exterior* region. The topological normal is directed from the interior to the exterior.

An open_shell can be thought of as a closed_shell with one or more holes punched in it. It is more general as a face because its domain can have handles.

[2]A 0 dimensional shell is a point and a 1 dimensional shell is a wire-frame. As mentioned before, wire-frame models are of no interest to us.

Chapter 3

The Geant4 simulation framework

3.1 Overview

The GEANT4 project has the objective of producing an object oriented framework for the simulation of future and LHC experiments. It was started in 1994 by a few tens of scientists, from different laboratories and experiments, aiming to apply a rigorous approach to High Energy Physics simulation software construction. Today GEANT4 is a world-wide collaboration, consisting of about 100 researchers including physicists, engineers and computer scientists, working in different areas of expertise. Strong importance is given to the study of the software process in such a large, geographically dispersed, development effort; the use of design and Quality Assurance tools has proven to be a decisive factor during the lifetime of the project. The connection between open/transparent software and the scientific validation of the produced results has been emphasised. Development is nearing its completion, with the beta testing phase starting in July 1998 and the production phase in January 1999.

The aim is to provide a flexible and extendible software toolkit. These requirements are fulfilled by exploiting object oriented methodologies and C++ implementation technology. The Booch analysis and design method has been used[Boo93], but transition to the Unified Modeling Language, the emerging standard [UML], will occur in the next few months; the diagrams in this thesis use UML notation.

Another crucial issue is runtime performance. GEANT4 is required to match the speed of its Fortran predecessor[App93] which offered much less functionality.

GEANT4 is managed in close collaboration with RD45, an R&D project which is demonstrating the suitability of Object Data Base Management Systems for high energy physics data storage and selection, and with analogous projects going on in the experiments to create an object oriented framework for reconstruction and analysis. External packages are used for implementing basic building blocks, such as container classes.

The structure of the framework is shown in Figure A.1. The categories are explained in only a few words; see [G+95], [G+97a] and [G+97b] for more details (in this order). Let us follow a particle through the detector and observe which categories come into play:

- The particle is generated in the category `Event Generator` and passed to `Event Management`.
- The particle starts traversing the detector. `Track Management` propagates it step by step.
- `Physics Process`es may change the speed and direction of the particle. They can also destroy it or generate new particles.
- Particle tracking interacts with `Geometry` to ascertain in which component of the detector the particle is moving. One way to describe the detector setup is to import via the `CAD interface`.
- Components are built of different `Materials`. Physics processes act differently depending on the material.
- `Magnetic Fields` bend the trajectory of the particle; this must be taken into account by tracking.
- Sensitive detector elements respond to particles passing through them, generating `Hits`.
- The generated signals are processed and `Digi`tised.
- `Visualisation` shows tracks and digitised hits in the detector setup. See Figure 3.1 for an example.
- These simulation results can also be stored in an Object Database Management System via the `ODMBS Interface`.

- The whole process can be controlled via command line or graphical user interfaces (`UI GUI`).

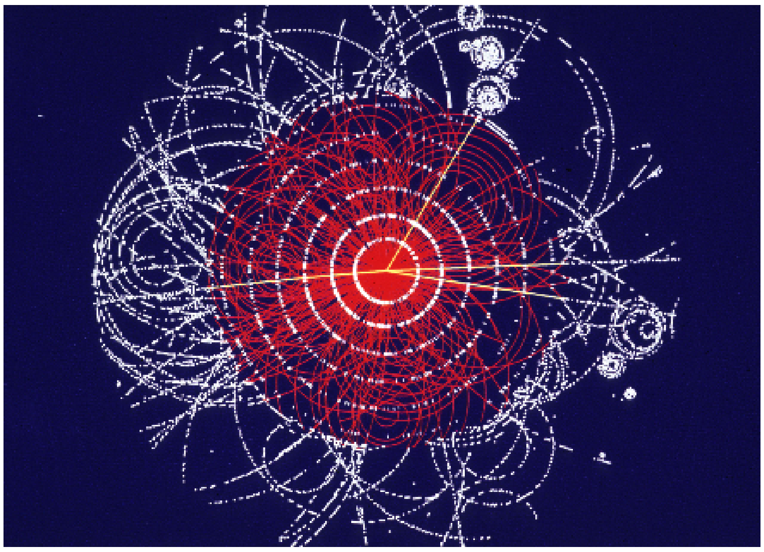


Figure 3.1: Simulation of the decay of a Higgs boson into 4 muons (straight tracks). LHC experiments will be looking for this event, predicted by theory.

This thesis presents developments made in the `Geometry` category, which is the subject of the next section.

3.2 Geometry in Geant4

The role of the geometry in GEANT4 is to give the simulation user the ability to describe the geometrical structure of a detector and to allow the simulation system to propagate particles efficiently in this model detector. The geometrical model of GEANT4 borrows concepts from previous simulation packages and adds advances in key areas as well as refinements.

Important considerations in the design were the need for performant navigation and the requirement to exchange detector geometries with Computer Aided Design (CAD) systems via the ISO Standard for the Exchange of Product Model Data (STEP). The STEP related parts are discussed in Chapter 2. The purpose of this section is to give an overview and highlight what services the STEP related part should offer to the geometry. The reader should refer to [G^+97b] and [Ken95b] for details.

3.2.1 Detector representation

The new feature of tracking in CAD models allows, for the first time, the comparison between simulations of the "exact" engineering descriptions of detectors and the more traditional, simpler, geometry descriptions used today.

A detector's geometry is described by listing the different elements it contains and specifying their positions and orientations. The concepts employed include:

logical volume An un-positioned detector element of a certain shape which can incorporate other volumes and which has attributes, e.g. information about the material.

physical volume An element positioned with respect to an enclosing logical volume. It can be a simple *placement*, given by a transformation in the reference coordinate system of the logical volume, or a *replica*, describing several placements with some parametrisation function. The latter results in considerable memory savings for typical detector setups, as these contain a lot of identical components.

touchable volume A uniquely defined detector element. Strictly speaking, it is not part of the geometry description, but is created and destroyed for e.g. physics processes which have to know about the surroundings of the tracked particle.

A detector can be described by utilising a hierarchy of volumes, each of which can contain smaller volumes. Or it can be specified by listing all the volumes one by one, with no volume containing another[1]. The former way has been utilised successfully to provide performant simulations, for example in GEANT3 [App93]. The latter type of "flat" geometry structure is required to interface to CAD systems where a hierarchy is not possible.

The shape and dimensions of volumes is placed in a separate entity, the *solid*. Services which have to be provided by solids are described in Section 3.2.3, after the motivations have been described.

Solids of simple shapes, including rectilinear boxes, trapezoids and spherical and cylindrical sections and shells, are available directly, allowing Constructed Solid Geometry (CSG) style modelling. More complex solids are defined by their bounding surfaces, e.g. planes, second order surfaces or

[1]More precisely, with each volume placed in a single "world volume".

Non-Uniform Rational B-Spline (NURBS) surfaces. This style of modelling is called Boundary Representation (B-Rep).

Modern CAD modellers use the latter approach, as they need an accurate representation for engineering analysis and manufacturing purposes. Former simulation packages like GEANT3 employed the more compact CSG approach, as such packages have to represent a large number of solids (up to $\approx 10,000,000$ in future experiments) and perform efficient particle tracking in them.

Conversion of B-Rep models to CSG necessitates simplifications with a loss of information, which are no longer acceptable for certain newly developed detector types; precise conversion is impossible in the general case. In addition, today's hardware and computer graphics algorithms are able to cope with the higher memory and computation time requirements of particle tracking in B-Rep models. Thus GEANT4 has a *hybrid modeller*, including *both* B-Rep and CSG models, the latter for reusing existing detector descriptions and for simulation tasks not demanding extremely high precision. Utilities converting old models to new ones are provided as well.

3.2.2 Particle propagation in detector model

The second task of geometry is to handle the geometrical propagation of particle tracks. In addition, geometry has to be aware which volume each particle is moving in.

Particles are propagated in discrete *steps*. Physics processes and geometry compete to determine the step length:

- Each of the physics processes models a certain physical phenomenon, e.g. scattering of particles on atomic nuclei. The process proposes a step length which it considers to be small enough to model the physical phenomenon at the desired accuracy.
- Geometry proposes a step length such that the step takes the particle to the next volume boundary. Boundaries are important; the particle may be just about to enter a sensitive detector element, and physics processes may change their behaviour if the particle enters a different material. Some, e.g. light refraction, may even occur on the boundary.

The smallest of the proposed step sizes is taken. Once the step length is known, physics processes may modify the direction, momentum and energy of the particle, and can also destroy the particle or create new ones. This process is repeated as often as necessary, i.e. until the particle exits the detector, is destroyed or simply becomes uninteresting for the simulation user.

Determination of the geometric step length, the distance to the next volume boundary, involves the following:

1. The solid containing the particle is identified, unless this information is known already.
2. The intersection distance to volumes inside of the current solid, and the distance to the boundary of the current solid are calculated.
3. The first intersection, i.e. the smallest intersection distance is taken.

It is important to note the similarity between *ray tracing*[Gla89] and particle tracking. In both cases, straight lines — called *light rays* or *viewing rays* and particle tracks, respectively — are followed to the next volume boundary. The basic operation, intersecting a ray with a volume or a surface, is also the same. In fact, all of GEANT4 geometry has profited greatly from techniques developed for ray tracing and such techniques are omnipresent in this thesis.

An important class of algorithms exists for optimising ray tracing, those exploiting the so called *ray coherence*, the fact that a lot of rays are nearly parallel and close to each other. Unfortunately, particle tracks in a detector are scattered too much, thus ray coherence techniques provide no benefits.

Until now, it was tacitly assumed that particle tracks are straight lines. However, charged particles moving in an electro-magnetic field do not follow linear trajectories between interactions. In a uniform field their trajectories are helical, while in non-uniform fields the curves have no analytic description. The problem is handled by numerical integration of the equations of motion. The curve thus obtained is approximated by *chords* such that the approximation is never farther from the curve than a given tolerance. Thus the task is solved by using smaller but straight trajectory pieces. See [AGGon] for details.

In order to efficiently traverse a detector model geometry, it is critical to reduce the number of intersections of candidate volumes made in each step. As intersections are potentially costly operations, an optimisation method should minimise the number of candidate volumes. Different methods can be used for this — some of them inspired by techniques used in ray tracing. A space subdivision method was developed to organise volumes into an efficiently searchable hierarchy: the "smart voxels" method[Ken95b]. Thanks to its adaptive nature, this copes well with detector geometries where the complexity of the parts varies to a great extent.

3.2.3 Requirements for solids

The geometry requires solids to offer some basic services. These services constitute the interface of an abstract class. This interface is described here. It is particularly important, for it is the only link between the STEP related code and the rest of the geometry category.

B-Rep solids are described in terms of their bounding surfaces, thus it is indicated for every operation which surface operations it is based on.

bounding box A simple shape surrounding the solid. Computation uses the bounding boxes of the surfaces.

inside A point is classified as being inside or outside the solid. A ray is shot from the point and its intersections with the surfaces are examined.

surface normal The normal vector pointing outside the solid. It is the normal vector of one of the surfaces at the given point.

distance from a point The distance from the closest surface. It can be an underestimate, as this operation is not essential for tracking, it just makes optimisations possible. It is often implemented as the distance to the bounding box of the solid.

distance from a point along a ray Essential. The same operation exists for surfaces.

3.2.4 Tolerances

GEANT4's propagation methods were designed to provide accuracy and efficiency. To minimise the number of geometrical calculations, tracks are propagated exactly to boundaries and their state is stored: whether they are on a boundary, whether they are exiting the current volume, etc.

Tolerances in GEANT4 geometry[Ken95a] were introduced to avoid problems when the track crosses a volume boundary. Due to precision problems, the intersection point does not lie precisely on the boundary, and this could result in a boundary being intersected several times or even incorrect physics, e.g.

no reflection when a particle should be reflected, because it was reversed an even number of times. The solution adopted is the following:

Boundaries are effectively given a "thickness", called `kCarTolerance`. The tolerance is chosen to be very small compared to detector features but much larger than the expected arithmetic errors. Intersection operations are allowed to return any point within this boundary. Boolean operations, such as classifying a point being inside or outside, are required to return "on boundary" whenever the point is within the thick boundary.

This concept of tolerances is generalised and is present throughout the code. Whenever a geometric object is returned by an operation, it need not be precise; locations and lengths have a tolerance of `kCarTolerance` and angles have one of `kAngTolerance`. In conditions formulated with such objects, one must take the tolerances into account. Special cases might need to be handled — an "if" with two branches can become one with three branches. An example: when a decision depends on the sign of a rotation angle ϕ, the special case $\phi \approx 0$, i.e. $|\phi| <$ `kAngTolerance` must also be investigated.

Precision problems are highlighted whenever algorithms are presented in this thesis. Their handling usually involves these tolerances.

3.2.5 Overview of the B-Rep modeller

This section sketches an outline of how STEP geometric and topological entities are mapped to classes in GEANT4.

The STEP standard was created to provide a solid modeller with all the convenient notions in modelling. As such, a geometry described in STEP tends to be verbose and often, there are multiple ways to describe the same concept.

GEANT4, on the other hand, must be able to perform particle tracking in the detector models. Verbosity is undesirable because it compromises efficiency, a crucial requirement, and multiple ways of describing the object tend to increase code size and implementation effort unnecessarily. Thus the internal object structure in GEANT4 will differ significantly from the object structure in the STEP exchange file.

In fact, there is another class tree which roughly corresponds to the tree of STEP entities. Objects of these classes takes care of STEP input/output, creating the objects which perform tracking and writing them out. This architecture minimises the amount of coupling between tracking and STEP input/output.

The geometric entities are implemented to the full extent. There are entities which can be represented with other entities, e.g. any finite part of a *parabola* can be represented with a NURBS curve (`b_spline_curve`), or a `rectangular_trimmed_surface` is equivalent to a `curve_bounded_surface` with `pcurve` bounds, but the conversion is never worthwhile, for reasons of efficiency. Simplifications have to come from the topology.

Note that `surface`s only appear as domains of `face`s. Therefore they are trimmed either explicitly by geometry, or implicitly by topology. There exist no `surface`s without bounds, thus something equivalent to a collection of `loop`s can be included in the most basic `curve` class. The same holds for `curve`s and their two end `point`s. With this implementation strategy, trimming geometric entities, and thus one of the major functionalities of topology, is handled.

The other major functionality of topology, reversing the senses of surfaces and curves, can also be put into the `curve` base class. One just has to traverse the topology structure down to the geometric entity, thereby collecting all the reversal flags, and in the end, the sense of the geometric entity is either reversed or not. All this can be done while building the GEANT4 object structure. Nothing will remain of the topology structure except the information which `face`s belong to the same solid.

3.2.6 From alpha to beta

The beta release of GEANT4, scheduled for July 1998, is going to include the developments presented in this thesis. Let us summarise these developments once again, this time more precisely with the aid of STEP terms.

- Swept surfaces, surfaces of linear extrusion and surfaces of revolution, have been implemented. This is the topic of Chapter 5.
- One has to be able to find out whether a point on a surface lies within the bounding loops of the associated face. This is vital for the correct operation and the efficiency of tracking. General methods are specialised whenever this enables improvements in performance. Performance is further improved by optimising the special cases implemented in the alpha versions. This is presented in Chapter 6.

- The above developments are the heaviest users of curves code in the B-Rep modelling part. As a result, the code of the alpha versions has been extended, optimised and re-designed. This is presented first, in Chapter 4, for it forms the basis of the other chapters.

Chapter 4

Curves

4.1 Overview

Curves in STEP are used to form swept surfaces (Chapter 5) and the boundary for faces of solids (Chapter 6). This chapter defines their basic operations and discusses motivation for them, their design from an object oriented point of view, and also presents the algorithms used.

Each section presents a basic operation. These are listed below, along with at least one reason why they are necessary. Chapters 5 and 6 try to reuse these operations for their purposes, rather than introduce new ones. Detailed motivation for the operations is given there.

- Curves must be constructed in the first place (Section 4.3).
- Each point on a curve has a representation as a 3D point and one as a parameter value. STEP has both representations, and for certain operations the missing representation might be more convenient, therefore conversions are implemented (Section 4.2).

  Functions for handling the start and end point are provided. Recall (Section 3.2.5) that almost all curves are bounded by points; this is why they are curve operations rather than a separate class.
- The basic operation of ray tracing, surface intersection with a ray, must be implemented for swept surfaces as well. These are specified in terms of a general curve, therefore ray intersection must be implemented for all types of curves. Intersection of these 1D entities in 3D is best done by intersecting them in 2D (Section 4.6); the 3D intersection point is

easily computed from the 2D one. This is in accordance with an important principle in computer graphics, namely that the dimensionality of any problem should be reduced whenever this is possible.

- The simplest way of transforming curves to 2D is parallel projection to an arbitrary plane (Section 4.5).
- As with solids (Section 3.2.3), intersection with a ray can be optimised by intersecting with a bounding box first (Section 4.4).
- The definition of the interior of a face (Sections 2.4 and 6.1) involves the tangent to a curve at a given point (Section 4.7).

The operations are normally pure virtual functions in the abstract curve base class `G4Curve`. Figure B.1 shows the interface of this class.

When describing an operation, special attention is given to

- whether the operation is *time critical.* Inefficient implementation of a time critical operation is likely to affect the overall performance of geometry significantly.
- precision problems. The tolerance constants (Section 3.2.4) are used extensively throughout the chapter.
- generic testing procedures for the operations.

The sections are divided into parts specific for the curve types: `line`, `circle`, `ellipse`, `parabola`, `hyperbola`, `b_spline_curve`, `composite_curve`, `pcurve` and `offset_curve`. The first six are termed *curve primitives.* Operations for the remaining curve types have a trivial implementation and therefore these are not described in the subsequent sections:

intersection_curve always has a 3D representation. The operations are delegated to that representation.

seam_curve is also available as a `pcurve`, of which it is a special case.

4.2 Parametrisation

Each point on the curve has two representations: one as a 3D point and one as a parameter value. Conversions between them are implemented. The conversion from 3D is only guaranteed to work when the point is sufficiently close to the curve, i.e. within a distance of `kCarTolerance`. Member functions related to any point and the start and end point are shown in Figure 4.1.

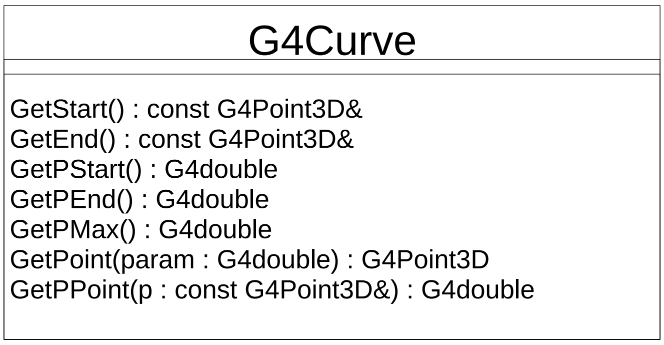

G4Curve
GetStart() : const G4Point3D&
GetEnd() : const G4Point3D&
GetPStart() : G4double
GetPEnd() : G4double
GetPMax() : G4double
GetPoint(param : G4double) : G4Point3D
GetPPoint(p : const G4Point3D&) : G4double

Figure 4.1: Conversions between the two representations of a point on a curve.

In time critical operations, the number of conversions should be minimised, to avoid degrading performance. Consider a NURBS curve: in the one direction, one needs evaluations of polynomials, and in the other, the solution of a non-linear equation! However, it is inconvenient for a function to remember which representations it has already calculated. This functionality can be moved to a class holding both representations and evaluating the one from the other when needed. This class with "lazy evaluation" is called `G4CurvePoint` (Figure B.2).

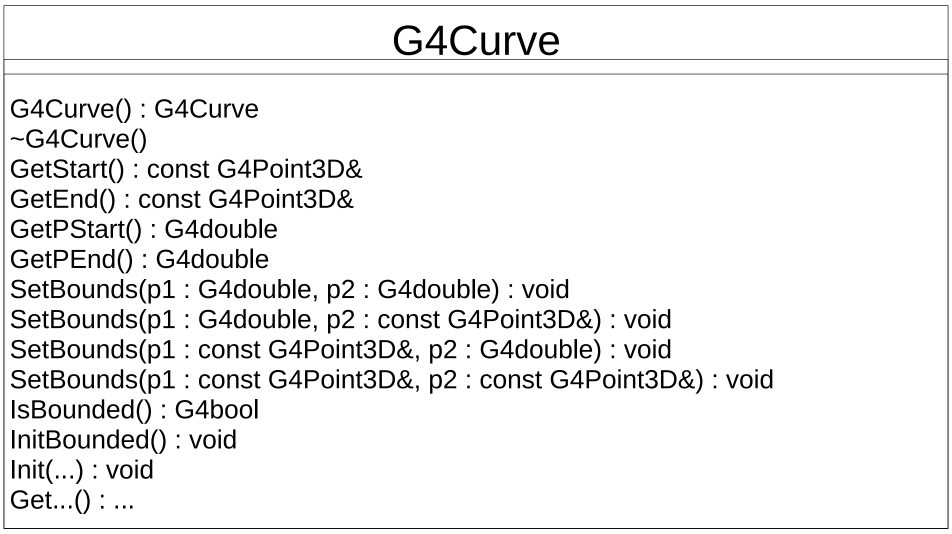
In other operations, all representations can be computed in advance. This is done for the end points.

4.3 Construction

The construction of curve objects takes place in three phases:

1. A call to the constructor (without parameters).
2. Passing the shape information. The parameters correspond directly to the attributes in the EXPRESS description of the curve; this makes STEP input/output and documentation easier.
3. Passing the boundary information, i.e. the two end points. They can be given either as parameter values or as 3D points.

See Figure 4.2 for the function signatures and Figure 4.3 for an example. Motivations for this scheme are given below.



```
G4Curve

G4Curve() : G4Curve
~G4Curve()
GetStart() : const G4Point3D&
GetEnd() : const G4Point3D&
GetPStart() : G4double
GetPEnd() : G4double
SetBounds(p1 : G4double, p2 : G4double) : void
SetBounds(p1 : G4double, p2 : const G4Point3D&) : void
SetBounds(p1 : const G4Point3D&, p2 : G4double) : void
SetBounds(p1 : const G4Point3D&, p2 : const G4Point3D&) : void
IsBounded() : G4bool
InitBounded() : void
Init(...) : void
Get...() : ...
```

Figure 4.2: Member functions for curve creation. Note that `SetBounds` has 2×2 different signatures, for the two kinds of representation of two points.

Note that not all curves have boundary information. The curve might be closed or finite, just ready for use after its shape is known. Also, infinite curves can be used in the definition of swept surfaces; see Chapter 5.

Curve objects are usually constructed when the STEP interaction part reads a STEP format file. In general, the shape of the curve and its endpoints are not available at the same time: the shape always comes from the geometry schema, whereas the endpoints might come from topology. A one phase construction would require that STEP creator classes store one half of the information until the second half becomes available. This would complicate their implementation unnecessarily, whereas curve objects are required to store all this information — no matter if it comes in pieces.

The question remains why the shape information is not passed to the constructor. The short answer is: flexibility. Properties of curves might have to be changed after construction; it is foreseen that simulation users modify the geometry inside Geant4. The approach that users have to destroy a given object, construct another and update all the references to the object is not feasible. Moreover, construction is not time critical, extra function calls are affordable.

```
G4Ellipse e;

// the standard coordinate system (1,0,0) (0,1,0) (0,0,1)
G4Axis2Placement3D pl;
pl.Init(G4Point3D(0,0,0), G4Vector3D(0,0,1), G4Vector3D(1,0,0));

// semi-axes: 5 and 8
e.Init(pl, 5, 8);

// a quarter of the ellipse
G4Point3D p(0, 8, 0);
e.SetBounds(p, pi/2);
```

Figure 4.3: Example for curve construction.

The `Init` function is, naturally, different for each class, whereas `SetBounds` can be generic. However, a part of the intersection and bounding box calculations can be moved here. These tasks must be implemented in the pure virtual `InitBounded` member.

The STEP interaction part also outputs curves, beside reading and constructing them. Read only access — “`Get`” functions — must be implemented for all attributes corresponding to the non-derived EXPRESS attributes, i.e. the attributes used for construction. These are also used for interacting with the visualisation category.

4.4 Bounding box calculation

Bounding boxes are used to optimise intersection algorithms. Frequently, a negative answer can be obtained by intersecting with the box first, which is an extremely fast operation. The main requirements for bounding box calculation — not necessarily just that of curves! — are the following, in decreasing order of importance:

- Some bounding box should always be calculated if one exists. See Section 5.1.4 for a case when even this is a rather complicated problem.
- The bounding box should wrap the object as tightly as possible. Ideally, the bounding box is the smallest box still containing the object.

- Computation should be reasonably fast. In fact, the bounding box can and should be pre-computed during construction, i.e. in `InitBounded` at latest; this is why the speed of computation is not critical issue.

A `G4BoundingBox` object (Figure B.3) takes care of storing the bounding box, and provides support for a frequent method of construction:

1. start from a box specified by two points, or from a point, a "box" of 0 extent;
2. specify points and/or boxes which must lie inside. The box grows as necessary.

4.4.1 Curve primitives

Circle and ellipse

As the `circle` is a special case of `ellipse`, no separate implementations exist. This is the case with all operations but the construction and the time critical operations.

The bounding box is the tightest box including

- the start and end points;
- those points of the *full* ellipse which have the minimum or the maximum x, y or z coordinate. Of course, only those of the six points are taken which happen to lie on the curve.

The points of the ellipse are given by

$$\boldsymbol{\lambda}(u) = \mathbf{C} + (R_1 \cos u)\mathbf{x} + (R_2 \sin u)\mathbf{y} \tag{4.1}$$

where u is the parameter value, $\mathbf{C}$ is the centre of the ellipse and $\mathbf{x}$ and $\mathbf{y}$ are coordinate axes of the placement coordinate system.

We will look for the parameter values of the extreme points. One can exploit the fact that extrema lie on opposite sides of the centre, i.e. if the one extreme value is at parameter value u, the other is at $u + \pi$.

Computing the extreme point is only shown for x coordinates (subscript 1); for the other coordinates it is analogous. Extrema are computed by setting the derivative of $\boldsymbol{\lambda}_1(u)$ equal 0.

$$\begin{aligned}
\boldsymbol{\lambda}_1'(u) &= 0 \\
(\mathbf{C}_1 + (R_1 \cos u)\mathbf{x}_1 + (R_2 \sin u)\mathbf{y}_1)' &= 0 \\
-R_1\mathbf{x}_1 \sin u + R_2\mathbf{y}_1 \cos u &= 0 \\
\tan u &= \frac{R_2\mathbf{y}_1}{R_1\mathbf{x}_1}
\end{aligned}$$

The inverse tangent of a quotient is always computed with the Standard C function `atan2`, which handles the special case when the denominator is 0.

Parabola

The method of computation is similar here, except that only one extreme point exists for each coordinate. The parametric representation is

$$\boldsymbol{\lambda}(u) = \mathbf{C} + F(u^2\mathbf{x} + 2u\mathbf{y}) \tag{4.2}$$

where F is the focal distance of the parabola. Once again, the derivative is set to 0:

$$\begin{aligned}
\left(\mathbf{C} + F(u^2\mathbf{x} + 2u\mathbf{y})\right)' &= 0 \\
2F\mathbf{x}_1 u + 2F\mathbf{y}_1 = 0 &= 0 \\
u &= -\frac{\mathbf{y}_1}{\mathbf{x}_1}
\end{aligned}$$

The case $\mathbf{x}_1 = 0$ is uninteresting, the bounding box need not be extended. If $\mathbf{y}_1 = 0$, the the parabola is in the yz plane (of the global coordinate system) and all points are extreme, thus so is the starting point, which is used to extend the bounding box anyway. If $\mathbf{y}_1 \neq 0$, the parabola is symmetric to an axis in the yz plane and there is no extreme point.

Also small values for denominators can cause numerical problems; and a 0 stored in a variable is almost never exactly 0. It is better to have the comparison

$$|\mathbf{x}_1| < \texttt{kAngTolerance}$$

The use of the angle tolerance is appropriate here because small $\mathbf{x}_1$ values are almost identical to the angle of $\mathbf{x}$ to the yz plane, as $|\mathbf{x}| = 1$.

Hyperbola

The parametric representation is

$$\boldsymbol{\lambda}(u) = \mathbf{C} + (R_1 \cosh u)\mathbf{x} + (R_2 \sinh u)\mathbf{y} \tag{4.3}$$

where R_1 is the real and R_2 is the imaginary semi-axis. The parameter value for the extreme point is computed as follows:

$$\begin{aligned}
\boldsymbol{\lambda}_1(u) &= 0 \\
(\mathbf{C}_1 + (R_1 \cosh u)\mathbf{x}_1 + (R_2 \sinh u)\mathbf{y}_1)' &= 0 \\
R_1\mathbf{x}_1 \sinh u + R_2\mathbf{y}_1 \cosh u &= 0 \\
\tanh u &= -\frac{R_2\mathbf{y}_1}{R_1\mathbf{x}_1}
\end{aligned}$$

If $\mathbf{x}_1 \approx 0$, the bounding box need not be extended, the hyperbola is symmetric to an axis in the yz plane. The same holds when the right hand side is outside the range of tanh, $]-1;+1[$; the equation has no solution.

NURBS curve

A NURBS curve[Far93] is a parametric rational curve described in terms of control points and basis functions. It is given by

$$\frac{\sum_{i=0}^{k} w_i \mathbf{P}_i N_i^d(u)}{\sum_{i=0}^{k} w_i N_i^d(u)} \tag{4.4}$$

where $w_i > 0$ are the *weights*, $\mathbf{P}_i$ are the *control points*, u is the parameter value and d the *degree* of the NURBS curve. $N_i^d(u)$ are called *basis functions* and are defined in terms of values called knots:

$$u_{-d} \leq \ldots \leq u_{-1} \leq u_0 \leq u_1 \ldots u_{k+1}$$

and the recurrence

$$N_i^1(u) = \begin{cases} 1 & \text{if } u_i \le u < u_{i+1} \\ 0 & otherwise \end{cases}$$

$$N_i^d(u) = \frac{u_{i+d} - u}{u_{i+d} - u_{i+1}} N_{i+1}^{d-1}(u) + \frac{u - u_i}{u_{i+d-1} - u_i} N_i^{d-1}(u)$$

After the definition, let us return to the problem of the bounding box. The *convexity property* of NURBS curves (and surfaces) states that the curve lies within the convex hull of its control points. A bounding box of the control points is thus a bounding box of the curve, and a reasonably tight one.

4.4.2 Pcurve

A `pcurve` is described in 2D, in the parametric space of an associated surface. The bounding box is a notion in 3D space. In order to compute it, data from both the surface and the 2D curve are needed. The 2D bounding box of the `pcurve` determines a surface patch. This patch contains the curve, therefore its (3D) bounding box is a bounding box of the `pcurve`. This simple solution avoids the complicated conversion into 3D space.

4.4.3 Other curves

Line

The two end points specify the best possible bounding box.

Composite curve

The bounding box must contain the bounding boxes of all components. One box has to be extended repeatedly by each other box.

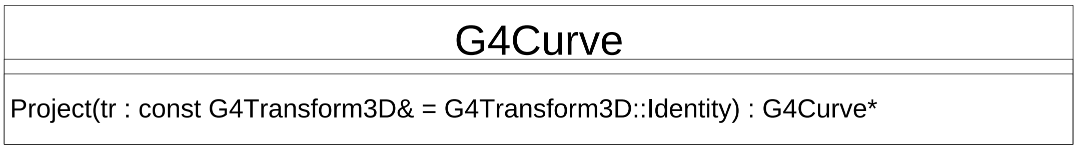
Offset curve

The bounding box should be extended by the absolute value of the offsetting distance, as the latter can be negative as well.

There is no straightforward implementation for the other operations. Offset entities are rarely used in modelling detectors; none of the detector descriptions available at the time of writing contain them. Therefore their implementation is postponed for later.

4.5 Parallel projection

The parameter of this operation is the plane to which the curve is parallel projected: the *image plane*. The image plane is specified with a placement coordinate system; it is the xy plane of that coordinate system. The result is a 2D curve. See Figure 4.4.



```
Project(tr : const G4Transform3D& = G4Transform3D::Identity) : G4Curve*
```

Figure 4.4: Parallel projection of a curve.

Parallel projection to an arbitrary plane is an affine transformation, preserving many important properties of transformed objects[FvDFH90]. Hence the result can always be one of the STEP curve types, or a slight modification of those; hardly any new curve types had to be created. The fact that the result is 2D is normally not shown explicitly in the curve object. All kinds of curves can be conceived as 2D curves, if all their points lie in the xy plane.

Time critical operations need parallel projections, but in most cases the projection is known in advance, thus the result can be pre-computed. Therefore parallel projection itself is not time critical; one can strive for a simple and understandable implementation.

Parallel projection must preserve the parametrisation of curves: the image of the point with parameter value u must have the parameter u in the projected curve. Problems might arise as parallel projection is a many to one mapping, thus a point on the projected curve might not have a unique parameter value. This problem and its solution is presented in Section 4.5.4.

4.5.1 Conics

The implementations for conics have a lot a features in common. These are described first in this section.

It is tempting to try implementing a generic parallel projection operation for all conics. This is possible for the shape when one uses the generic equation for conics, valid in *projective* 3D space[PP86], but the parametrisations of the curves must be preserved as well, and these differ from the parametrisation which can be handled naturally in projective geometry; recall that the parametrisations for the ellipse and the hyperbola (Equations (4.1) and (4.3)) involve trigonometric or hyperbolic functions. Thus parallel projection has a different implementation for each of the three types of conics.

Parallel projection of a conic — ellipse, parabola or hyperbola — produces a conic of the same type. The parametrisation, however, differs in general: the point with parameter value 0 does not coincide with a vertex of the conic, as it is the case in STEP. The solution is to introduce a *shift* in the parametrisation. This shift is managed by the conics base class `G4Conic`, shown in Figure B.4.

Note that, besides shifting, both the original and the projected conics must have the same sense when viewed along the direction of projection. The z axis direction of the placement coordinate system ($\mathbf{z}_{new}$) must be set to $(0, 0, 1)$ or $(0, 0, -1)$ for the projection, accordingly. The sense is given by transforming the z axis of the placement coordinate system of the conic into the coordinate system specifying the projection. This vector ($\mathbf{z}_{transformed}$) and $\mathbf{z}_{new}$ must point to the same side of the image plane.

The special case when $\mathbf{z}_{transformed}$ lies in the image plane must be handled. It is characterised by

$$|\mathbf{z}_{transformed_3}| < \texttt{kAngTolerance}$$

the angle of the vector to the plane being smaller than the tolerance. In this case, the conics map to a line. See Section 4.5.4 for how this case is handled.

The intersection fails in this case and no 2D curve is returned.

Note that this special case ensures that all conics map to non-degenerate conics, thus no similar “special cases” can occur when computing the shift.

The projection algorithm sets z_{new} first, then finds the shift by computing which parametric value maps to the point which would have parametric value 0 if the projection was parametrised the STEP way, i.e. a vertex of the conic. Given the shift, attributes of the projected image are computed.

Circle

In general, the result is a 2D ellipse (and not a 2D circle!). To simplify the implementation, the operation is just delegated to `ellipse`.

Ellipse

The shift for the ellipse, u, is characterised by the following: the point of the original curve corresponding to u becomes an endpoint of one of the axes of the projection when projected to the image plane. Therefore its distance from the centre is an extreme value:

$$\left\{[(R_1 \cos u)\mathbf{x}' + (R_2 \sin u)\mathbf{y}']^2\right\}' = 0 \tag{4.5}$$

where $\mathbf{x}'$ and $\mathbf{y}'$ are the projections of $\mathbf{x}$ and $\mathbf{y}$; see also Equation 4.1.

With the shorthands $\mathbf{a} \overset{def}{=} R_1\mathbf{x}'$ and $\mathbf{b} \overset{def}{=} R_2\mathbf{y}'$ this becomes

$$\begin{aligned}
\left\{\mathbf{a}^2 \cos^2 u + \mathbf{b}^2 \sin^2 u + 2\mathbf{ab} \sin u \cos u\right\}' &= 0 \\
\left\{\frac{\mathbf{a}^2 + \mathbf{b}^2}{2} + \frac{\mathbf{a}^2 - \mathbf{b}^2}{2} \cos 2u + \mathbf{ab} \sin 2u\right\}' &= 0 \\
-\left(\mathbf{a}^2 - \mathbf{b}^2\right) \sin 2u + 2\mathbf{ab} \cos 2u &= 0 \\
\frac{2\mathbf{ab}}{\mathbf{a}^2 - \mathbf{b}^2} &= \tan 2u
\end{aligned}$$

The first and the second semi-axis are the distances of the parametric point u and $u + \pi/2$ from the centre, respectively.

Parabola

The shift is determined with the aid of the *derivative* vector of the curve (with respect to the parameter). The length of the derivative is maximised at the vertex of the parabola, i.e. at $u = 0$, as is shown below.

The derivative vector is (see Equation 4.2)

$$\Big(\mathbf{C} + F(u^2\mathbf{x} + 2u\mathbf{y})\Big)' = 2F(\mathbf{x}u + \mathbf{y}) \tag{4.6}$$

The maximum length is found by setting the square length of the derivative to 0.

$$\begin{aligned}
\left\{[2F(\mathbf{x}u + \mathbf{y})]^2\right\}' &= 0 \\
\left\{(\mathbf{x}u + \mathbf{y})^2\right\}' &= 0 \\
\left\{\mathbf{x}^2u^2 + 2\mathbf{x}\mathbf{y}u + \mathbf{y}^2\right\}' &= 0 \\
2\mathbf{x}^2u + 2\mathbf{x}\mathbf{y} &= 0 \\
u &= \frac{-\mathbf{x}\mathbf{y}}{\mathbf{x}^2}
\end{aligned}$$

As $\mathbf{x}$ and $\mathbf{y}$ are perpendicular, this becomes $u = 0$.

The similar derivation can be used to find the u value for the vertex of the projected parabola, with $\mathbf{x}'$, $\mathbf{y}'$ and $\mathbf{C}'$ instead of $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{C}$:

$$u = \frac{-\mathbf{x}'\mathbf{y}'}{\mathbf{x}'^2}$$

This is exactly the value of the shift.

Hyperbola

The derivation of the shift is analogous to the one for ellipses, except that hyperbolic functions are used:

$$\begin{aligned}
\left\{[(R_1 \cosh u)\mathbf{x}' + (R_2 \sinh u)\mathbf{y}']^2\right\}' &= 0 \\
\left\{\mathbf{a}^2 \cosh^2 u + \mathbf{b}^2 \sinh^2 u + 2\mathbf{a}\mathbf{b} \sinh u \cosh u\right\}' &= 0 \\
\left\{\frac{\mathbf{a}^2 - \mathbf{b}^2}{2} + \frac{\mathbf{a}^2 + \mathbf{b}^2}{2} \cosh 2u + \mathbf{a}\mathbf{b} \sinh 2u\right\}' &= 0 \\
\left(\mathbf{a}^2 + \mathbf{b}^2\right) \sinh 2u + 2\mathbf{a}\mathbf{b} \cosh 2u &= 0 \\
-\frac{2\mathbf{a}\mathbf{b}}{\mathbf{a}^2 + \mathbf{b}^2} &= \tanh 2u
\end{aligned}$$

4.5.2 Pcurve

Projection of `pcurve`s is difficult; so is their intersection in 2D (Section 4.6). The problem is that projection is a 3D operation and does not make sense in the parameter space of a surface. One may think of solutions like converting the `pcurve` into 3D; this is a difficult task, as the result depends on both the surface and the `pcurve` itself.

Fortunately, there is hardly any need for implementing these operations: the problems arising in Chapter 6 only deal with geometric entities on a surface, and can be more naturally formulated and solved in the parametric space of the surface.

4.5.3 Other curves

Line

Projecting the two end points is sufficient here.

NURBS curve

It is sufficient to project the control points, as parallel projection is an affine transformation.

Composite curve

The components must be projected one by one.

4.5.4 Problems of parametrisation

Parallel projection might map several points onto one. One of the uses of parallel projection is intersecting two 3D curves (Section 6.2): the curves are projected, intersected in 2D, and the parameter value of the intersection point is used to find the intersection in 3D. Clearly, this might fail if multiple parameter values are mapped to the same 2D point.

The set of points which maps to the same point is a union of single points and line segments. One can ensure that line segments never occur, as STEP curves only contain a finite number of line segments, and we have at least one degree of freedom when choosing the projection in all our algorithms (see Chapter 6). Projections which would map a line segment to a point must signal this fact by returning no 2D curve. Detection is easy:

- Conics have no line segments.
- A line must check if its two end points map to the same point, or, considering precision problems, if the end points are closer to each other than `kCarTolerance`.
- Line segments in NURBS curves are always between control points. Error must be signalled if these control points map to the same point.

Self intersecting projections cannot be avoided. After eliminating line segments, the 2D space – parameter value conversion should still return a finite number of values. To simplify the implementation, 2D curves are only required the return one of these parameter values, but the choice has to be deterministic and close parameter values must map to close points, as it is the case with all STEP curves. E.g. if two segments of the map onto the same curve, the implementation must discard one of them, rather than choose one point / segment from the one and another from the other.

Now the special case of projecting conics when the image is a line segment can be handled. Modelling the image with a line results in incorrect parametrisation. Using conics with degenerate (0) attributes would cause numerical problems in computations and the 2D space – parameter value conversion would be erratic. The solution is to implement a new type of STEP curve, called `G4DegenerateConic2D`. This is generic to all conics, therefore a member of `G4Conic` takes care of its creation.

This curve type does not need to implement its operations efficiently, for it is expected to be created very rarely. It is solely there to protect against numerical problems.

The conic is projected to $\mathcal{P}'$, the plane perpendicular to $\mathcal{P}$, the image plane of the original projection, containing $\mathcal{L}$, the line resulting from the original projection. The projection to $\mathcal{P}'$ as well as $\mathcal{L}$ are stored. The implementation of the intersection operation (Section 4.6) is interesting:

The ray, which lies in $\mathcal{P}$, is intersected with $\mathcal{L}$. A ray going through the intersection and perpendicular to $\mathcal{P}$ is shot on $\mathcal{P}'$. This ray might intersect the projected image on $\mathcal{P}'$; the parameter value for this intersection is returned.

If the ray in $\mathcal{P}'$ is started from far enough, the choice of the parameter value is consistent, close values map to close points.

4.6 Intersection

This operation should only be called for 2D curves — recall that most curve types can represent either a 2D curve or a 3D curve. The 2D curve is intersected with a 2D ray. The result is the intersection closest to the ray starting point, if the ray and the curve intersect at all. The intersection must be performed efficiently.

The points of the ray are given by

$$\mathbf{S} + i\mathbf{d} \quad \text{where} \quad i \geq 0 \quad \text{and} \quad |\mathbf{d}| = 1$$

in this section. Note that i is the intersection distance from the starting point $\mathbf{S}$ of the ray.

An intersection point has three possibly useful representations:

- a 3D point;
- a parameter value, as the intersection lies on the curve;
- the intersection distance i.

Most intersection routines have some functionality in common. It is often more convenient to intersect with the line containing the ray rather than the ray. In this case, the intersection might lie behind the starting point, and all intersections must be computed to subsequently take the one closest to the starting point. Moreover, most algorithms intersect with the unbounded curve, i.e. they use no information about the end points intrinsically. Hence, each intersection must be checked to see if it lies on the bounded curve.

All this functionality and the management of the three representations, with lazy evaluation for efficiency, is moved to the `G4CurveRayIntersection` class. Each intersection operation initialises such an object with the intersection distance set to `kInfinity`, a very large number, meaning no intersection. Candidate intersections are passed to the `Update` methods as they are found, and if they are real intersections and are closer to the starting point than the best candidate so far, the intersection object is updated. The intersection object is returned by the operation (Figure 4.5).

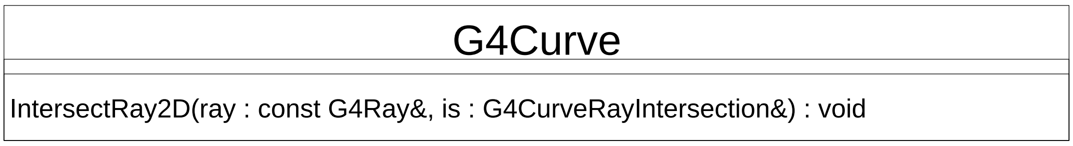

G4Curve
IntersectRay2D(ray : const G4Ray&, is : G4CurveRayIntersection&) : void

Figure 4.5: The curve – ray intersection operation.

Some of the intersection calculations are rather basic. Despite this fact, they are present in order to demonstrate how precision problems are handled.

4.6.1 Curve primitives

Line

A `line` is parametrised as follows:

$$\mathbf{P} + u\mathbf{V}$$

A point $\mathbf{X}$ lies on the line if and only if

$$\det\begin{pmatrix} \mathbf{X} - \mathbf{P} & \mathbf{V} \end{pmatrix} = 0$$

Let us use a unit vector $\mathbf{v} = \mathbf{V}/|\mathbf{V}|$ with the same direction as $\mathbf{V}$; reasons will be given later on.

Let us substitute the equation of the ray into $\mathbf{X}$:

$$\begin{aligned} \det\begin{pmatrix} \mathbf{S} - \mathbf{P} + i\mathbf{d} & \mathbf{V} \end{pmatrix} &= 0 \\ (\mathbf{S}_1 - \mathbf{P}_1 + i\mathbf{d}_1)\,\mathbf{v}_2 - (\mathbf{S}_2 - \mathbf{P}_2 + i\mathbf{d}_2)\,\mathbf{v}_1 &= 0 \\ i &= \frac{(\mathbf{S}_1 - \mathbf{P}_1)\,\mathbf{v}_2 - (\mathbf{S}_2 - \mathbf{P}_2)\,\mathbf{v}_1}{\mathbf{d}_2\mathbf{v}_1 - \mathbf{d}_1\mathbf{v}_2} \end{aligned}$$

This candidate intersection can then be passed to `G4CurveRayIntersection`.

Care must be taken to handle special cases. The denominator might be close to 0; this corresponds to parallel or identical lines. As the denominator is the angle between the vectors $\mathbf{d}$ and $\mathbf{v}$ — recall that these are of unit length — a test against `kAngTolerance` is appropriate.

The ray falls on the line if and only if both the numerator and the denominator are 0. The numerator is a length, to be tested against `kCarTolerance`. This special case is handled by providing three candidate intersections: the starting point of the ray and the two end points of the line.

Circle

2D circles are represented as ellipses. Real 2D circle objects can be added when this proves to be important for efficiency. Note that also projection code, the creator of 2D curves, must be modified (Section 4.5).

Ellipse

Once again, separate algorithms are implemented for all types of conics. The generic representation would lead to a performance loss.

An affine transformation is pre-computed which maps the ellipse into the unit circle centred at the origin. This is composed of the transformation to the placement coordinate system and scaling with the reciprocals of the two semi-axes in x and y direction.

This transformation is applied to the starting point **S** and the direction **d** of the ray, resulting in $\mathbf{S}'$ and $\mathbf{d}'$. For the intersection distance i, the following must hold:

$$
\begin{aligned}
1 &= \left(\mathbf{S}' + i\mathbf{d}'\right)^2 && (4.7)\\
0 &= \left(\mathbf{d}'^2\right) i^2 + \left(2\mathbf{S}'\mathbf{d}'\right) i + \left(\mathbf{S}'^2 - 1\right) && (4.8)\\
i &= \frac{-\mathbf{S}'\mathbf{d}' \pm \sqrt{\left(\mathbf{S}'\mathbf{d}'\right)^2 - \mathbf{d}'^2\left(\mathbf{S}'^2 - 1\right)}}{\mathbf{d}'^2} && (4.9)
\end{aligned}
$$

The formula for i gives 0, 1 or 2 candidate intersections. The case of one intersection is expected to occur rarely, therefore it is handled by the code for two intersections.

Hyperbola

The basic idea is the same as with the ellipse: let us transform the hyperbola into the hyperbola $x^2 - y^2 = 1$. If we take into account the y coordinates of

$\mathbf{S}'$ and $\mathbf{d}'$ as *imaginary* coordinates, the equation to be solved is Equation 4.7.

There is, however, no need to perform inefficient complete arithmetic. The result (Equation 4.9) only contains

$$\begin{array}{rcl} \mathbf{S}'^2 & = & \mathbf{S}_1'^2 - \mathbf{S}_2'^2 \\ \mathbf{d}'^2 & = & \mathbf{d}_1'^2 - \mathbf{d}_2'^2 \\ \mathbf{S}'\mathbf{d}' & = & \mathbf{S}_1'\mathbf{d}_1' - \mathbf{S}_2'\mathbf{d}_2' \end{array}$$

The special case $\mathbf{d}'^2 = 0$ must be handled. Note that this special case never occurs for the ellipse, with a $\mathbf{d}'$ having real coordinates. Of course, testing against 0 means testing the absolute value $<$ `kCarTolerance`^{2}.

In this special case, Equation 4.8 becomes a first order equation, thus

$$i = \frac{\mathbf{S}'^2 - 1}{2\mathbf{S}'\mathbf{d}'}$$

It might occur that also $\mathbf{S}'\mathbf{d}' = 0$, becoming $|\mathbf{S}'\mathbf{d}'| <$ `kCarTolerance`^{2} in the code. Geometrically, this means that the transformed ray is either $x = y$ or $x = -y$, i.e. an asymptote to the transformed hyperbola. No intersection exists in this case.

Parabola

A more direct geometric solution is taken for the parabola (Equation 4.2). A point $\mathbf{X}$ on the parabola is equidistant from its focus $\mathbf{F}$ and its directrix $\mathbf{L}_0 + \lambda\mathbf{l}$:

$$\det\begin{pmatrix} \mathbf{X} - \mathbf{L}_0 & \mathbf{l} \end{pmatrix} = (\mathbf{X} - \mathbf{F})^2$$

For a point on the ray this becomes

$$\begin{array}{rcl} \det\begin{pmatrix} \mathbf{S} - \mathbf{L}_0 + i\mathbf{d} & \mathbf{l} \end{pmatrix} & = & (\mathbf{S} - \mathbf{F})^2 + 2(\mathbf{S} - \mathbf{F})\mathbf{d}i + \mathbf{d}^2 i^2 \\ i^2 & + & \\ i\left\{2(\mathbf{S} - \mathbf{F})\mathbf{d} - \det\begin{pmatrix} \mathbf{d} & \mathbf{l} \end{pmatrix}\right\} & + & \\ \left\{(\mathbf{S} - \mathbf{F})^2 + \det\begin{pmatrix} \mathbf{S} - \mathbf{L}_0 & \mathbf{l} \end{pmatrix}\right\} & = & 0 \end{array}$$

This equation is solved for i, giving the candidate intersections. The missing quantities are

$$\begin{aligned}\mathbf{F} &= \mathbf{C} + F\mathbf{x} \\ \mathbf{L}_0 &= \mathbf{C} - F\mathbf{x} \\ \mathbf{l} &= \mathbf{y}\end{aligned}$$

4.6.2 NURBS curve

The ray – NURBS surface intersection method is reused[Sul95]. At first, a conversion from NURBS to Bézier curves is performed[RHD89]. The Bézier curve – ray intersection problem is solved with a technique called *Bézier clipping*, described in [NSK90]. The technique uses the convex hull property of Bézier curves to identify regions of the parameter axis where no intersection with the ray can occur. These regions are discarded, and the remaining region is clipped repeatedly until the region is sufficiently small. The intersection point is any point of the remaining region. The method normally has stronger than linear convergence to the intersection point, outperforming most other subdivision methods.

For the ray – surface intersection, clipping was performed first in u parametric direction, then in v direction, then again in u direction, always alternating the direction. There is no need for this in the ray – surface algorithm.

4.6.3 Composite curve

It is possible to intersect the ray with each of the constituting curves but this might be inefficient when the number of these is large. Therefore the bounding boxes of the curves are intersected first and only the curves passing this test are examined further. Several optimisations are possible; they are worthwhile to consider for an extremely large number of curves:

- The bounding boxes could be sorted by their distance along the ray.
- Once an intersection is found on a nearby curve, curves whose bounding boxes do not contain the intersection can be eliminated.
- Even sorting bounding boxes can be expensive. A space subdivision method, with pre-computed subdivision, can be used instead.

Type of conic	$C(x,y,z)$	**Tangent**
Circle	$x^2+y^2-R^2$	$(-y,x,0)$
Ellipse	$x^2/R_1^2+y^2/R_2^2-1$	$(-R_1^2/R_2^2\cdot y,x,0)$
Parabola	$4Fx-y^2$	$(y,2F,0)$
Hyperbola	$x^2/R_1^2-y^2/R_2^2-1$	$(R_1^2/R_2^2\cdot y,x,0)$

Table 4.1: Conic tangents in the placement coordinate system.

4.7 Tangent calculation

The tangent to a `curve` is a vector. Only the direction of the vector is interesting; implementations are free to choose the magnitude. The tangent is defined everywhere, except at singular points. By convention, the vector $(0,0,0)$ is returned at these.

Computation of the tangent may involve derivation of the parametric equation of the curve and taking the value of the derivative at the given point, specified as a parameter value. In other cases, using the 3D representation of the given point might be more convenient or more efficient. For this reason, the point is represented by a `G4CurvePoint` object; see Section 4.2 and Figure 4.6.

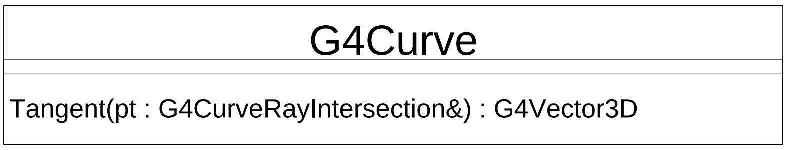


Figure 4.6: Taking the tangent at a given point of the curve.

4.7.1 Conics

If a 3D point is given, the tangent is easily computed in the placement coordinate system. Let (x,y,z) be a point. [STEP42] specifies a function $C(x,y,z)$ for each conic; the points with $C=0$ are on the conic. The tangent is always

$$\left(-\frac{\mathrm{d}C}{\mathrm{d}y},\frac{\mathrm{d}C}{\mathrm{d}x},0\right)$$

Table 4.1 shows C and the tangent for each type of conics.

Type of conic	**Derivative**
Circle	$-\sin u \cdot \mathbf{x} + \cos u \cdot \mathbf{y}$
Ellipse	$-\frac{R_1}{R_2} \sin u \cdot \mathbf{x} + \cos u \cdot \mathbf{y}$
Parabola	$u\mathbf{x} + \mathbf{y}$
Hyperbola	$\frac{R_1}{R_2} \sinh u \cdot \mathbf{x} + \cosh u \cdot \mathbf{y}$

Table 4.2: Conic tangents at a parametric point.

The tangent to all conics is $(-\mathbf{C}_2, \mathbf{C}_1, 0)$, where $\mathbf{C}$ is the location of the point in the placement coordinate system. The cost of the operation is just a conversion to the placement coordinate system.

If a parameter value is given, it is more efficient to compute the derivative than to convert the parameter into a 3D point: the conversion to the placement coordinate system is avoided. Table 4.2 shows the derivatives, sometimes scaled by a constant. See Section 4.4.1 for the parametric equations.

4.7.2 NURBS curve

For NURBS curves, the tangent is the derivative of the parametric equation (4.4). The resulting expression is a polynomial of the parameter. The coefficients can be pre-computed for each parametric range determined by the knots of the curve.

Normally, tangents at intersection points are determined. The intersection algorithm computes the parameter value of the intersection, thus there is no need for the expensive 2D point – parameter value comparison.

4.7.3 Other curves

Line

The direction of the line is taken.

Pcurve

Each surface class provides tangent vectors: one in positive u direction and one in positive v direction. However, these are undefined if no surface normal exists at the given point. If this occurs, the `pcurve` tangent is undefined as well.

Composite curve

The composite curve delegates this operation to the component curve on which the given point lies. Only the end points of the components are special. The tangent may become undefined for the following reasons:

- Two curves join there and their tangents are different.
- More than two curves join there, i.e. there is a self intersection. Section 6.3 explains why self intersections are permitted.

Chapter 5

Swept surfaces

Swept surfaces are obtained by moving a curve along a trajectory in 3D space. The curve may change its shape or remain the same during sweeping. Also, its orientation may be unchanged or follow the trajectory. Swept surfaces where a 2D contour is swept along a 3D curve are also called *generalised cylinders*. STEP only defines the *surface of linear extrusion*, where a curve is swept along a line, and the *surface of revolution*, where a curve is revolved about an axis.

A key operation of surfaces is the intersection with a ray. Literature exists for ray tracing swept surfaces. [BK85] handles the case where the trajectory is a continuous 3D curve; this method is too expensive for the special STEP swept surfaces. The essence of the specialised methods in [Kaj83] and [vW84] is the same: the problem of intersection is reduced to the intersection of two 2D curves, using the properties of the construction of swept surfaces. The same principle is followed here. The papers differ in their curve – curve intersection methods. A discussion follows in the subsequent sections.

The section of each surface type presents the implementation of four basic operations for surfaces (Section 3.2.3): intersection with a ray, parametric point – 3D point conversion and tangent and bounding box computation. The problem of bounding boxes is difficult due to the numerous possibilities offered by STEP to trim surfaces.

For a `pcurve`, the 3D curve itself is swept, rather than the 2D curve in the parametric space of the basis surface. The type of the 3D curve depends on both the type of the 2D curve and the type of the surface, and there is no simple relation between these three types in general. The only feasible approach is computing the 3D representation. It is proposed that this is not implemented for each 2D curve – surface type. The 2D curve should be converted into NURBS curve segments, and the surface into NURBS surface patches. Computing the 3D representation then has to be implemented only for 2D NURBS curves in the parametric space of NURBS surfaces. Using this approach, $\approx c + s$ rather than $\approx c \cdot s$ operations are written, where c and s are the number of curve and surface types, respectively.

5.1 Surfaces of linear extrusion

A `surface_of_linear_extrusion` is a simple swept surface or a generalised cylinder obtained by sweeping a curve in a given direction. The resulting surface shall not self intersect. The parametrisation is as follows:

$$\boldsymbol{\sigma}(u, v) = \boldsymbol{\lambda}(u) + v\mathbf{V} \tag{5.1}$$

where $\boldsymbol{\lambda}(u)$ is the parametrisation of the swept curve and $\mathbf{V}$is the extrusion direction. The parametrisation range for v is $-\infty < v < +\infty$ and for u is defined by the curve parametrisation.

Note that nothing prevents the swept curve from being infinite.

5.1.1 Intersection

The task is to find the first intersection of the surface with a ray. As with the curves (Section 4.6), it is sufficient to find one representation out of the following three:

- point in 3D space;
- point in the 2D parameter space of the surface;
- the intersection distance.

The surface and the ray are projected along the extrusion direction to a perpendicular plane. The image of a swept point is thus single point, therefore the image of the surface is the image of the swept curve. The swept curve can be any 3D curve, in contrast to [Kaj83] and [vW84] where 2D contours are swept. Of course, the resulting surface is the same as the one obtained by sweeping the 2D projected image,but parametrisation is different. Parallel projection of curves is discussed in Section 4.5. Note that the projection of the swept curve can be pre-computed.

The image of the ray is either a ray or a point. If the image is a point, the original ray is either non-intersecting or a tangent to the surface; in the latter case, tolerances allow us to choose any point within the boundary of the surface as the intersection. Otherwise, if the image is a ray, a 2D ray – curve intersection must be determined; see Section 4.6 for a discussion. Subsequently, the 2D intersection distance is mapped easily to a 3D intersection distance, given the direction of the 3D ray.

The resulting intersection might lie outside the boundary of the surface; see Section 6.2.3 for methods to perform the test. This does not mean that there is no intersection at all: there may be multiple intersections of the projected curve and ray, one of which might lie within the boundary. In order to determine further intersections, the projected ray is re-started, this time from the most recent intersection. This procedure repeats until there is no (2D) intersection or a 3D intersection inside the boundary is found.

5.1.2 Parametrisation

Evaluating Equation 5.1 is straightforward. The other direction, converting a 3D point **P** to a 2D parametric point, is more interesting. The parallel projection and 2D intersection of the previous section is used to find the u parameter — recall that the definition of the surface guarantees that the projection of the swept curve is a one-to-one mapping, thus the u value is unique. The result is substituted into Equation 5.1:

$$\begin{aligned}
\mathbf{P} &= \boldsymbol{\lambda}(u) + v\mathbf{V} \\
\mathbf{P} - \boldsymbol{\lambda}(u) &= v\mathbf{V} \\
|\mathbf{P} - \boldsymbol{\lambda}(u)| &= v|\mathbf{V}| \\
|\mathbf{P} - \boldsymbol{\lambda}(u)| \cdot |\mathbf{V}| &= v|\mathbf{V}|^2 \\
v &= \frac{(\mathbf{P} - \boldsymbol{\lambda})\mathbf{V}}{\mathbf{V}^2}
\end{aligned}$$

5.1.3 Normal direction

The normal direction of the surface at a given point **P** is computed here. Recall that the normal at **P**, parametrised as $\mathbf{P}(u_0, v_0)$, has the same direction as the cross product of the tangent vectors to the parametric curves $u = u_0$ and $v = v_0$. In our case, $u = u_0$ is a line with direction $\mathbf{V}$; the tangent is $\mathbf{V}$. $v = v_0$ is the swept curve with tangent $\mathbf{t}(\mathbf{P})$. The resulting expression is $\mathbf{V} \times \mathbf{t}(\mathbf{P})$. The normal is undefined if and only if the tangent to the swept curve is undefined.

5.1.4 Bounding box calculation

The bounding box calculation is difficult, for it must involve the boundary of the surface, as the unbounded surface of linear extrusion is infinite in v direction and may be infinite in u direction as well. The goal is to find a correct and reasonably tight bounding box; see Section 4.4 for the detailed requirements on bounding box calculation.

The bounding box calculation is performed in a coordinate system whose z axis is the extrusion axis. Both the swept curve and the surface boundary are transformed to the coordinate system. A box containing the bounding box computed in that coordinate system is returned.

Let us ensure first that the surface is bounded in u direction. To achieve this, the swept curve must be trimmed if it is unbounded. Also, trimming of bounded curves might prove useful, if the curve is in fact trimmed, because this may lead to a smaller bounding box.

The curves which constitute the boundary might be described in the parameter space of the surface or in 3D space. Curves of the former type, i.e. `pcurves`, provide bounds on u in a straightforward manner. A box which includes all their 2D bounding boxes must be taken; its u range determines the part of the swept curve whose extrusion the `pcurves` lie on.

A similar strategy is chosen for 3D bounding curves. They are projected onto a plane normal to the extrusion direction and a (2D) box including the bounding box of each image is computed. The boundary of this box, four line segments, are intersected with the projected image of the swept curve. A range including the parameter value of each intersection, as well as the u range of `pcurves` gives correct bounds on u. The swept curve is trimmed with this range.

Now we are ready to compute the bounding box. The bounding boxes of `pcurve` bounding loops specify a v range $[v_{min}; v_{max}]$. The bounding box must include the (3D) bounding box of the trimmed swept curve, translated once by $v_{min}\mathbf{V}$ and once by $v_{max}\mathbf{V}$. The bounding box of all 3D bounding loops is computed, denoted by $\mathcal{B}$. The bounding box must include the bounding box of the swept curve, once put on top of $\mathcal{B}$ — i.e. the maximum z coordinate of $\mathcal{B}$ is the minimum z coordinate of the other box — and once put below $\mathcal{B}$.

5.2 Surfaces of revolution

This surface is obtained by rotating a curve one complete revolution about an axis. The parametrisation is as follows:

$$\begin{aligned} \boldsymbol{\sigma}(u,v) &= \mathbf{C} + (\boldsymbol{\lambda}(v) - \mathbf{C})\cos u \\ &+ ((\boldsymbol{\lambda}(v) - \mathbf{C}) \cdot \mathbf{V})\mathbf{V}(1 - \cos u) \\ &+ V \times (\boldsymbol{\lambda}(v) - \mathbf{C})\sin u \end{aligned} \tag{5.2}$$

where $\mathbf{V}$ is the direction of the axis, $\mathbf{P}$ is a point on the axis and $\boldsymbol{\lambda}(v)$ is the parametrisation of the swept curve. The parametric range for u is $0 \le u \le 360°$ and for v is defined by the curve parametrisation.

Many the algorithms presented are parallel to the ones for the surface of linear extrusion in Section 5.1. This fact will be often referred to, explicitly or implicitly.

5.2.1 Intersection

It is natural to view the surface in the *cylindrical coordinate system* whose axis is the axis of revolution. The radius, angular position and the position along the axis will be denoted by r, ϕ and z. We will also utilise a Cartesian coordinate system with the same z axis and $x = r\sin\phi$, $y = r\cos\phi$. Thus $x^2 + y^2 = r^2$.

In the cylindrical coordinate system, sweeping a point (r_0, ϕ_0, z_0) produces (r_0, ϕ, z_0) with ϕ going from 0 to 2π. Thus the projection produced by considering only r and z coordinates takes the surface and the swept curve into the same 2D curve on the rz plane — the dimensionality of the problem is reduced.

This projection is *not* a projection in its usual meaning — it does not preserve cross ratios[PP86], for instance — but the word will be used nevertheless, as the operation is analogous to the parallel projection used for surfaces of linear extrusion.

The ray is also projected to the rz plane — half plane, to be precise, for $r \geq 0$ — and is intersected with the image of the swept curve. The intersection point(s) are transformed back into 3D easily — the z coordinate is known. They are checked for lying within the boundary of the surface, similarly as with the surface of linear extrusion.

Problems are twofold:

- The image of the ray is not a ray but, as we will see later, a hyperbola. The intersection algorithms of Section 4.6 cannot be used.
- The images of the curves are not STEP curve entities, or representing them with STEP curves is inefficient. E.g. conics become fourth order curves.

For these reasons, the design of the projection and intersection operations is different from parallel projection and 2D ray – curve intersection; see Figure B.7. Projection produces an object which implements intersection, but this object is not a `G4Curve`. This has the advantage that fundamentally different approaches to surface of revolution – ray intersection can be implemented: "intersection" implements the parts of the algorithm depending on the actual ray and projection implements parts that can be computed in advance, by just knowing the surface.

Most curves implement the above projection approach, coming from [Kaj83] and [vW84]. The articles present different curve – curve intersection methods: [Kaj83] has a recursive subdivision method, whereas [vW84] deals with curve equations directly. Also, the usage of the (r^2, z) coordinate system is preferred to (r, z). The right choice of these methods depends on the type of the curve; details follow in the subsequent sections.

It must be noted that the above articles only consider curves lying in a plane containing the revolution axis. Thus the problem of projecting the swept curve is avoided.

Some general remarks on the algorithms:

- No effort was made to preserve the parametrisation. This would have introduced additional complications.
- As with the 2D ray – curve intersection, there are common tasks to intersection algorithms. The `G4CurveRayIntersection` class mentioned

in Section 4.6 is reused. Intersections with the line containing the ray are computed, not just the ones with the ray.

- The definition of surfaces of revolution ensures that projection is a one to one mapping; no problems similar to those in Section 4.5.4 arise.

Sweeping lines

A line can be described as

$$x = a \cdot z + b, \quad y = c \cdot z + d, \quad z \in \mathbb{R} \tag{5.3}$$

The special case where the line is in the xy plane is handled later.

Let us express the projection of a line in terms of r and z:

$$\begin{aligned} x^2 + y^2 &= r^2 \\ (az+b)^2 + (cz+d)^2 &= r^2 \\ (a+c)z^2 + 2(ab+cd)z + (b^2+d^2) &= r^2 \end{aligned} \tag{5.4}$$

The coefficients of z will simply be denoted by K, L and M. For an intersection point (r, z) of two lines their equations of hold simultaneously:

$$K_1 z^2 + L_1 z + M_1 = K_2 z^2 + L_2 z + M_2$$

Solving this quadratic equation yields at most 2 z values. r is obtained by substitution. Of course, only z values yielding $r^2 \geq 0$ are valid.

Sweeping conics

Let us project the conic onto the xy plane. The image is a conic. The general form of its equation is

$$Ax^2 + by^2 + Cxy + Dx + Ey + F = 0 \tag{5.5}$$

All points of the conic lie in the same plane. This yields the equation

$$Gx + Hy + Iz + J = 0 \tag{5.6}$$

As with the line, we would like to have Equation 5.5 expressed in terms of x and r — recall that $r^2 = x^2 + y^2$. This is possible and yields a fourth order polynomial of r and z, such that r is always at an even exponent, i.e. $r^0 = 1$, r^2 and r^4.

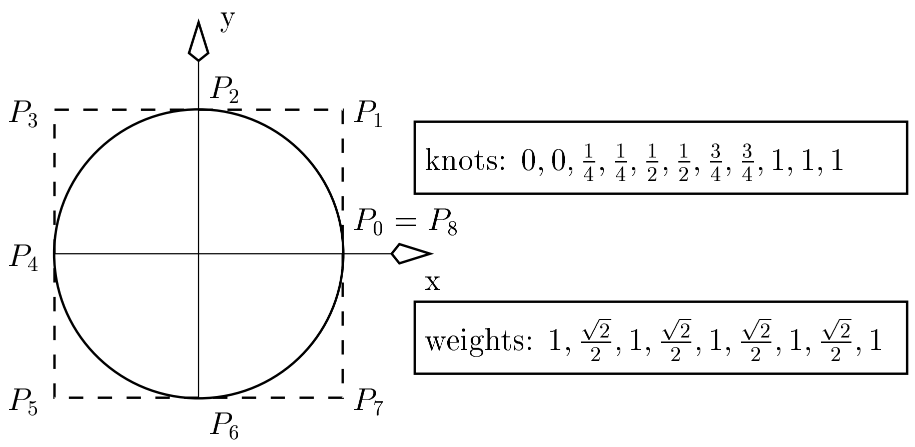
Let us now substitute $r^2 = Kz^2 + Lz + M$, the equation of the ray, into the fourth order polynomial. The substitution results in a fourth order equation of z. This can be solved analytically, but this is not worthwhile. A numerical method is better at finding zeroes. The choice of numerical methods in general involves a trade-off between efficiency and safe convergence behaviour. Laguerre's method is used, with subsequent polishing of the roots, as implemented in the popular Numerical Recipes package[PTVF92]. This method is not fast but it always converges for practical problems. Note that some of the roots might be wrong z values; this must be checked for.

Sweeping NURBS curves

The projection – 2D intersection approach involves serious difficulties: the projection of a NURBS curve to the rz half plane is not a NURBS curve. It does not seem worthwhile to implement a curve type even more complicated than NURBS just for the sake of this operation. Instead, the surface of revolution is constructed as a NURBS surface with the following method:

1. A NURBS representation is chosen for the unit circle. [PT89] contains a review of possible circle representations. Figure 5.1 shows an example.
2. The u direction knot vector and weights are taken from the unit circle representation.
3. The v direction knot vector and weights are taken from the swept curve.
4. The control points of the swept curve give the first column of the control points array.
5. A row of the control points array is obtained as follows: the control points of the unit circle are transformed to a circle symmetric to the revolution axis, such that the first control point of the circle falls on the

first control point of the row. The thus obtained control points form the row.



y
P_2
P_3
P_1
knots: $0, 0, \frac{1}{4}, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{3}{4}, \frac{3}{4}, 1, 1, 1$
$P_0 = P_8$
P_4
x
weights: $1, \frac{\sqrt{2}}{2}, 1, \frac{\sqrt{2}}{2}, 1, \frac{\sqrt{2}}{2}, 1, \frac{\sqrt{2}}{2}, 1$
P_5
P_7
P_6

Figure 5.1: Nine control point NURBS circle.

Sweeping composite curves

Intersection should be attempted with each component curve. Testing the bounding boxes of the curves usually speeds up this process.

5.2.2 Parametrisation

The direction parametric point to 3D point is trivial substitution into Equation 5.2. We will focus on the other direction, even though that conversion is not needed very frequently. Usually, the conversion is done at the intersection point, in order to obtain the tangent to the swept curve. But as we will see in the next section, tangent computation is a by-product of the intersection calculation.

v, the parameter along the swept curve is determined first. Projected curves lose their parametrisations, thus projection and the sequence of conversions 2D point – parameter – 3D point cannot be applied now.

Let the 3D point have r_0 and z_0 as r and z coordinates. The task can be formulated as looking for a point on the swept curve with $r = r_0$ and $z = z_0$. Let us project the swept curve to a plane containing the axis of revolution, an ordinary parallel projection this time. The plane determined by $z = z_0$

becomes a line in this projection. The intersection operation of Section 4.6 can thus be used to determine the set of points and curve segments with $z = z_0$.; these are checked one by one for intersections.

The case of sweeping NURBS curves is different. v can be obtained by performing the 3D point – parametric point conversion on the NURBS surface representing the surface of revolution. The u parameter resulting from that conversion must be discarded, since there is no NURBS representation of a circle having the desired uniform representation.

5.2.3 Normal direction

Once again, the normal is the cross product of the tangent to the curve $u = u_0$ and the tangent to $v = v_0$ where (u_0, v_0) is the parametric representation of the given point. $v = v_0$ is the swept curve rotated by v_0 about the axis of revolution, thus its tangent is the tangent of the swept curve rotated by v_0. $u = u_0$ is a circle, with tangent $(-y, x, 0)$ in a coordinate system having the axis of revolution as its z axis where (x, y, z) are the coordinates of the given point.

For conics and lines, the tangent $\mathbf{t}$to the swept curve at an intersection point can be obtained without using the parameter value, as a by-product of the intersection calculation, using the relation

$$\mathbf{t} = \left(\frac{\mathrm{d}C}{\mathrm{d}x}, \frac{\mathrm{d}C}{\mathrm{d}y}, \frac{\mathrm{d}C}{\mathrm{d}z} \right)$$

where $C = 0$ is the implicit equation of the curve. The direction might be reversed, but this is not a serious difficulty — one can also remember where a special point of the curve was projected and reverse the direction based on its relative location.

5.2.4 Bounding box calculation

The main problem is that the surface of revolution might be infinite in the v parametric direction, if the swept curve is infinite. In this case, the curve must be trimmed using the curve boundary of the surface. This trimming might be useful for bounded curves as well.

Curves which form the surface boundary can be parametric curves or 3D curves. `pcurve`s can be handled easily. A box including all their 2D bounding boxes must be taken; its v range determines the part of the swept curve whose revolution includes all the `pcurve` 3D boundaries.

The 3D bounding curves are projected to a plane containing the axis of revolution. A common 2D bounding box is determined on that plane. The swept curve is projected to the same plane and is intersected with the four line segments forming the bounding box of the projected boundary. The minimum and maximum parameter values belonging to intersections specify a v range whose revolution includes all the 3D bounding curves. The swept curve can thus be safely trimmed by the union of the two v ranges.

Once the swept curve is finite, the task is simple. Let us project the swept curve to the xz and the yz planes (z is the axis of revolution). The 2D bounding boxes give the following bounds: $z_0 \leq z \leq z_1$, $|x| < x_0$ and $|y| < y_0$. The cylinder about the z axis with radius $\sqrt{x^2 + y^2}$ and $z_0 \leq z \leq z_1$ contains the surface of revolution. The bounding box of that cylinder is the result of the operation.

Chapter 6

Curve bounded faces

The key functionality of solids is the ray – solid intersection. B-Rep solids are bounded by faces. For them, this operation involves ray – face intersections. Recall (Section 2.4) that a face is part of an associated surface, bounded by one or more loops, i.e. closed curves lying on the surface. It is not enough to determine the ray – surface intersection — the intersection point must also lie within the boundary. Surface intersection algorithms almost never make use of boundary information intrinsically, therefore we have two separate tasks. The latter task, called henceforth *inside operation*, is the subject of this chapter. The chapter is subdivided as follows:

- The interior of a face is defined precisely, and the inside operation is considered from a purely topological point of view. The definition and usage of topological notions relies on intuition rather than rigorous mathematics, both in this thesis and [STEP42].
- The object-oriented design of the inside operation and associated objects is presented.
- Guiding principles of the implementation are given, followed by the implementation for most surface and curve types. Surface types are defined there in detail.

6.1 The topological point of view

Being a topological notion, the interior of a face is defined in topological terms. Each loop of the face, i.e. a connected and closed, but possibly self-intersecting curve divides the surface into two parts; the one part is inside the loop, the other outside. The interior of the face consists of the points inside all loops. Note that there is no difference in the semantics of "inner" and "outer" bounding loops. These notions only define a preferred way of embedding the face into the plane. Neither the STEP definition of interior nor the inside operation in GEANT4 make the distinction.

The precise definition of the interior of a loop involves a point $\mathbf{P}$ on it and a direction $\mathbf{d}$ which is a tangent to the surface, but not a tangent to the loop at $\mathbf{P}$. $\mathbf{d}$ points towards the interior of the face if and only if

$$(\mathbf{n}(\mathbf{P}) \times \mathbf{t}(\mathbf{P})) \cdot \mathbf{d} > 0 \tag{6.1}$$

where $\mathbf{n}(\mathbf{P})$ is the *normal* to the surface and $\mathbf{t}(\mathbf{P})$ is the *tangent* to the loop at point P. Figure Figure 6.1 demonstrates this definition.

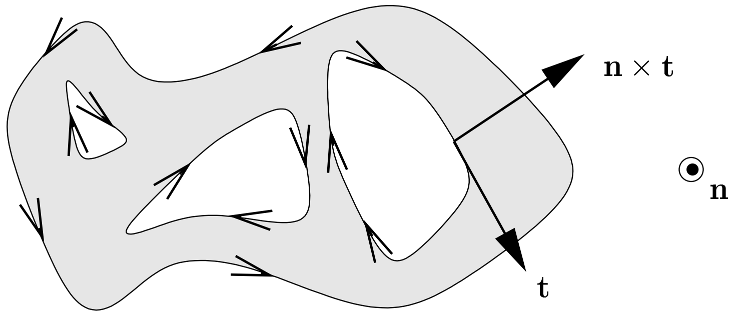


Figure 6.1: The interior of a face. As the surface normal points towards the reader, the outer loop runs counter-clockwise, and the inner loops clockwise.

The topological tangent and normal used here more or less agree with the corresponding geometrical notions. However, the tangent to a loop may not exist at singular points. In other words, a STEP curve may have knicks. Also, for some curves on the surface no normal can be defined. Such special cases are dealt with in Section 6.4.

This definition of the interior cannot be used directly; it offers no straightforward test to decide if a point on the surface lies in the interior or not. The solution adopted exploits the fact that the interior is connected and is inspired by the inside-outside algorithms on the plane[FvDFH90]. Let us consider a curve lying on the surface, starting from the point on the surface, and investigate which points it passes through. The curve will be referred to as the *test curve.* Three cases are worth distinguishing (see also Figure 6.2:

1. The test curve intersects at least one bounding loop. In this case, take the intersection closest to the starting point. Definition 6.1 allows testing if the test curve comes from the interior or the exterior, when plugged in the intersection point and the tangent of the test curve. We obtain a definitive answer in any case.

2. There is no intersection with the bounding loops but the test curve passes through a point known to be outside the face. Such points are on the boundary of a "naturally" bounded surface, bounded even when bounding loops are ignored. An example is an open NURBS surface. Points "at infinity", the *ideal points* in projective geometry[PP86] are also outside any face. An answer "outside" is obtained in this case.

3. There is no intersection with any of the bounding loops and no outside point is touched. The problem of being inside remains unsolved.

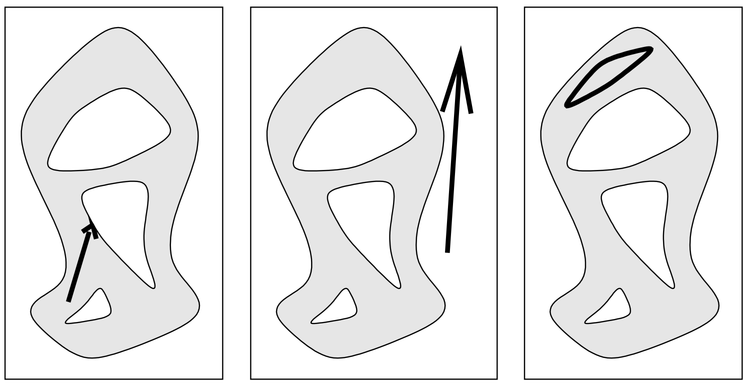


Figure 6.2: Testing if a point is on the interior of the face, by starting a curve from the point. Three cases are distinguished.

The test curve may be chosen freely among the curves lying on the surface. One can thus ensure that case 3 never occurs, by taking a curve going through either a boundary point or an outside point. This curve must then be tested with *every* bounding loop for intersection. This is the general schema for each of the inside operations.

The main problem is that the test curve, as well as curves forming boundaries, can be of any STEP type. The general curve – curve intersection problem is difficult to implement, and algorithms are unacceptably slow. The next section shows how the general intersection problem can be avoided for different types of surfaces.

6.2 The geometrical point of view

[NSK90] describes a general algorithm for the inside operation but it requires that the surface is present as a NURBS surface and all bounding curves are present as NURBS curves. This would require a lot of implementation work, and the resulting method is slow due to its generality. (See Section 6.2.4 for a more detailed description of the algorithm.) This section approaches the problem from the practical side: the efficiency of methods is valued higher than their generality. Therefore the general method will be avoided whenever possible by suitable choices for the test ray. Curve – curve intersections are always reduced to basic operations on curves: parallel projection onto a plane (Section 4.5), such that the one curve is mapped to a ray, then intersection of this ray with the other curve (Section 4.6). Projections are pre-computed whenever possible. Intersection tests should start with checking against the bounding box (Section 4.4).

The choice of a test curve is largely constrained by the surface it must lie on, therefore it depends on the type of the curve. The subsequent sections show and analyse these choices for each surface type. The test curve will be called $\mathcal{C}$ and its starting point, the ray – surface intersection, $\mathbf{P}$.

6.2.1 Pcurve boundaries and planes

If the surface is a plane, the boundary should be projected to it. The inside operation is reduced to projecting $\mathbf{P}$onto the plane and shooting a (2D) ray into any direction. Either an intersection occurs or the ray shoots off into infinity.

A similar problem occurs when all bounding loops are available as `pcurves`. $\mathbf{P}$must be represented in the 2D parameter space of the surface and a ray in that space is shot. However, some care must be taken, when choosing the direction, as the parameter space might be closed in one or both of the u and v directions and $\mathcal{C}$ might not go off to infinity. The simplest solution is to shoot $\mathcal{C}$ towards a point on one of the bounding loops.

“Mixed” boundaries, containing both `pcurves` and 3D curves, do not pose serious problems. Projections and intersections of projected images are not available for `pcurves`, but all the test curves used for different kinds of surfaces are either rays or connected ray segments in the parameter space of the surface.

6.2.2 Other elementary surfaces

Cylindrical surfaces

A `cylindrical_surface` consists of the set of points at a constant distance from a straight line, the *axis*. The choice for $\mathcal{C}$ is the ray starting from $\mathbf{P}$and parallel to the axis. This ray goes to infinity. Let us project the boundary to a plane $\mathcal{P}$ containing the axis and intersect them with the projected image of $\mathcal{C}$.

The problem is that projection is a many to one mapping (Section 4.5.4). Two different points of the boundary might be mapped to one on $\mathcal{P}$, coming from the two different sides of $\mathcal{P}$. The solution is to split the boundary with the plane $\mathcal{P}$ and to intersect only with the part of the boundary lying on the same side of $\mathcal{P}$ as the ray. Figure 6.3 visualises all of the above.

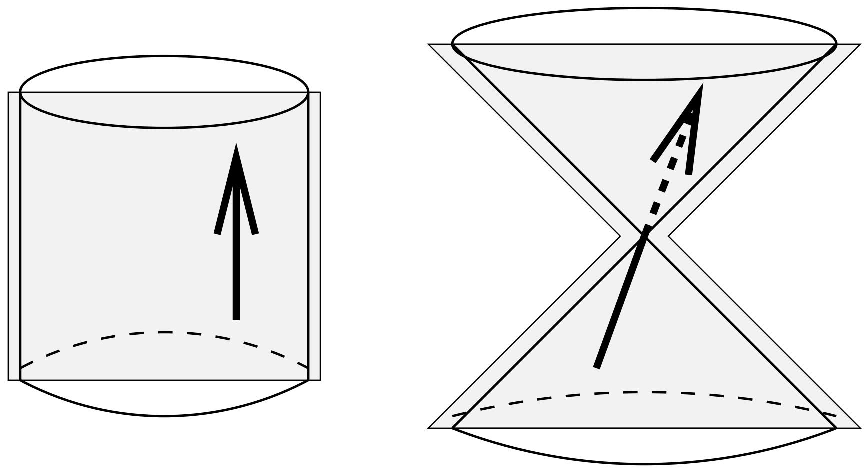


Figure 6.3: The test curve on a cylindrical and a conical surface. The boundary is split into two by the dotted plane.

Splitting boundaries is described in Section 6.3.

Conical surfaces

A `conical_surface` is the infinite surface produced by revolving a line about an intersecting line, the *axis*. The intersection point is called *apex*. Again, there is a choice for a $\mathcal{C}$ going off to infinity, the ray going through the apex. Projection is to a plane containing the axis, and the boundary is split with that plane; see Figure 6.3.

Spherical surfaces

The sphere is a closed surface; the only right choice for $\mathcal{C}$ is a curve going to a point on a boundary. The boundary is projected to a plane containing the centre of the sphere; they are split by this plane. Now we will show that it is possible to go from any point on the sphere to any other along arcs which are line segments on the projected image, which is a disk. The projection of $\mathcal{C}$ consists of line segments on diameters of the disk only:

- If both points are on the same side of the plane, move to the centre on that side. Otherwise, first move to the disk boundary, then move on the other side to the centre.
- Move from the centre to the destination.

See also Figure 6.4.

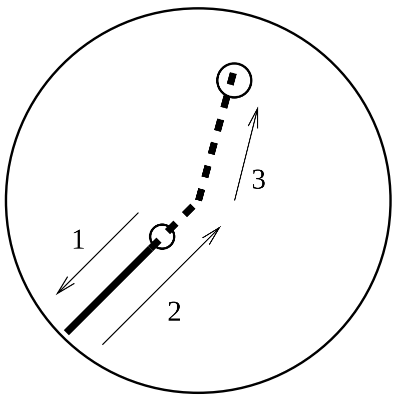


Figure 6.4: The test curve on a sphere. Projected view.

Toroidal surfaces

A `toroidal_surface` has a doughnut shape. It could be produced by revolving a circle about a non-intersecting line in its plane. This line is the symmetry axis of the torus.

Projection of the boundary is again to a plane containing the symmetry axis, and the boundary should be split with this plane. The parts should be split again into inner and outer parts — a point closer to the symmetry axis than the centre of the revolved circle is in the inner part — to make sure that no self-intersecting projected images occur.

Similarly as with the sphere, $\mathcal{C}$ must go to a point on a boundary, preferably along curves whose images are line segments:

- Move to the image of the axis along a normal to the axis. If the start and end point are on different sides of the image plane, move to the image boundary first and get to the axis on the other side.
- Move on the image of the axis to the point where the normal going through the destination point meets the axis. If necessary, get from the inner part to the outer part by moving to the boundary first.
- Move to the destination point along the normal.

Note that the second split, separating the inside and outside parts of the torus, is splitting with a cylinder. See also Figure 6.5.

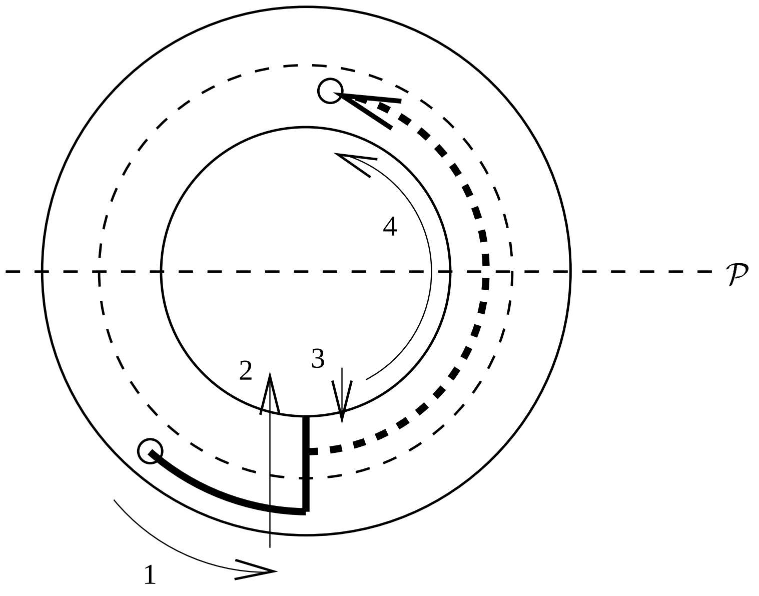


Figure 6.5: Test curve on a torus. The boundary is split into four parts (dashed lines). The projection onto the plane $\mathcal{P}$ maps the test ray (thick line) into line segments.

6.2.3 Swept surfaces

Surfaces of linear extrusion

This case is analogous to that of the cylinder (Section 6.2.2): a ray parallel to the extrusion axis is shot and intersection is tested on the projected image of the boundary. Projection is to a plane containing the axis, called $\mathcal{P}$. The difficulty lies in determining which planes are appropriate for splitting the boundary.

Complex algorithms are necessary to determine the splitting planes in advance, and implementation is likely to differ for all curve types. An adaptive, generic procedure is proposed instead. It exploits the fact that false or possibly false intersection points can be characterised in a simple manner, resulting in the following algorithm:

1. Find the closest intersection of the boundary with the ray in the projection. If the boundary was split already, take the part which might contain the ray. This is a simple task, as the boundary is only split with planes parallel to $\mathcal{P}$.

2. Determine the intersection point on the 3D ray as well as on the intersected boundary. The latter task is done by 2D point – parameter value – 3D point conversion.

3. Terminate if these two points are the same – a true intersection was found.

4. A possibly false intersection was found. Split the boundary with the plane equidistant from the two points and parallel to $\mathcal{P}$ (see also Figure 6.6). Now the two points are in different space partitions. Proceed with step 1.

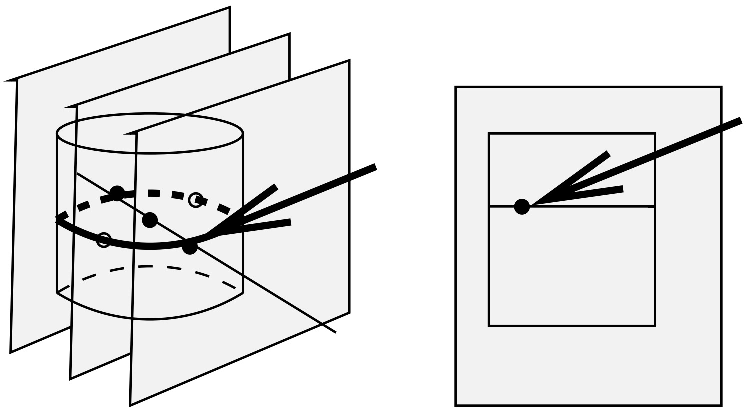


Figure 6.6: Adaptive subdivision of the boundary of a surface of linear extrusion. The plane in the middle is the new splitting plane.

It is expected that this method does not perform much more subdivisions than necessary. Note that even if the boundary is split into many parts, finding the appropriate part is fast: binary search with logarithmic execution time can be used.

Surfaces of revolution

This case is analogous to that of the torus (Section 6.2.2), but a general curve is revolved about the axis rather than a circle. Projection is performed onto a plane containing the revolution axis, and the boundary is split with this plane. However, the parts on the two sides of the plane cannot be split further in a trivial way to avoid false intersections.

It is possible to invent an adaptive method with the flavour of the method in Section 6.2.3. However, it would require a general algorithm of splitting the boundary with a cylinder, and the method presented in Section 6.3 cannot be generalised. We would have to resort to generic curve – circle intersections, and that would result in significant additional implementation effort. The problem of false intersections is resolved in a different way:

1. Another projection is performed, to a plane chosen at random. The resulting projections can be preserved between executions of this algorithm.
2. A ray is shot through the image of the candidate intersection, denoted by $\mathbf{P}'$. If the ray does not intersect the projected boundary at $\mathbf{P}'$, the candidate intersection is a false one. Note that an intersection at $\mathbf{P}'$ may only occur if the distance of $\mathbf{P}'$ from the ray starting point is at least $2 \cdot$ `kCarTolerance`.
3. The intersection is checked as in Section 6.2.3. If it proves to be a true intersection, we may terminate. Otherwise, let us proceed from step 1.

A disadvantage is that the algorithm does not "learn" in the same way as the one for the extruded surface. Whole segments of the boundary may overlap in the projection, and the resolution for false intersections may need to be performed frequently.

6.2.4 NURBS surfaces

On a NURBS surface, it is often impossible to find a test ray and a plane such that the image of the test ray is a 2D ray. Even if there is such a ray, it is difficult to find. We have to resort to test curves which are general

NURBS curves, e.g. the curve $u = u_0$ if u_0 is the coordinate of the point to be classified.

One choice is converting all bounding curves to NURBS curves and implement the general NURBS curve – NURBS curve intersection. There are solutions of different nature: subdivision algorithms and numerical algorithms. There is also a technique called *Bézier clipping*, a method based on subdivision, but converging faster than other subdivision methods. It can be termed as a geometrically based numerical method.

[NSK90] presents a method using Bézier clipping but cannot exploit the fact that one test – ray boundary intersection is enough in STEP (see Definition 6.1). Instead it determines if the number of intersections is even or odd, without actually performing the intersection. Performance comparisons are needed to decide if this method or the NURBS – NURBS intersection is more efficient.

6.2.5 Rectangular composite surfaces

Rectangular composite surfaces consist of a 2D array of surface patches which are rectangular in their own parameter spaces, such as untrimmed NURBS surfaces. They are glued together by mapping their parameter spaces onto the parameter space of the rectangular composite surface in the following way: the patch with indices (i, j) is re-parametrised linearly to become the parametric rectangle $(i-1, j-1, i, j)$. The resulting surface might be closed in one or both parametric directions.

The problem is that bounding loops do not "respect" the boundaries of patches. A bounding loop might intersect a patch boundary or might enclose the patch completely. The patch with the point to be classified might have to communicate with its neighbours to classify the point.

Methods in this chapter choose the test curve which is most appropriate to them. One might try to follow the test ray to a patch boundary, determine a test ray on the other patch, starting from the intersection, and follow that test ray; see Figure 6.7. This process is repeated until a bounding loop (case 1 in Section 6.1) or an outside point (case 2) is hit. The problem is, as the patches do not communicate, the test ray might be reflected to the patch where it comes from or form a loop in some other way (case 3).

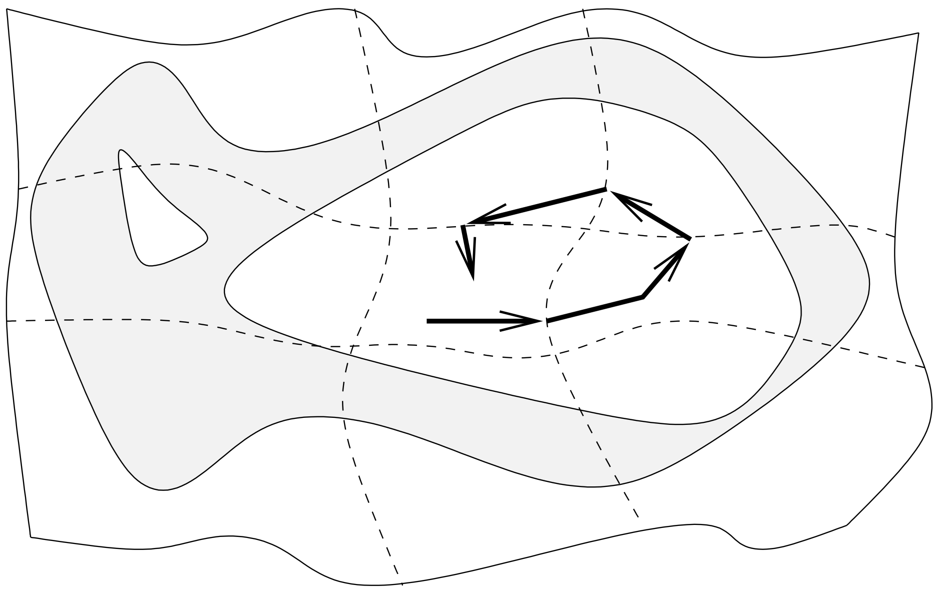


Figure 6.7: Problems with test rays on a rectangular composite surface.

6.3 Operations on the boundary

Let us first examine how a single bounding loop is best modelled. A bounding loop consists of several curves where the end point of a curve joins the start point of the next curve. It is closed. Curves do not intersect, except at common end points. This is almost the definition of the `composite_curve`, with the following differences:

- A `composite_curve` might be open.
- The bounding loop may self intersect at some points.
- A bounding loop may be a `vertex_loop`, a single point surrounded completely by the interior of the face.

The curve operations parallel projection, intersection in 2D as well as bounding box and tangent calculations can thus be reused; the boundary class `G4Boundary` can contain a `G4CompositeCurve`. Only the tangent operation (Section 4.7.3) has to be changed slightly: the tangent must be undefined at the self intersections.

Note that the above operations are implemented for every curve, therefore the more general `G4Curve` class can be used. This has the advantage that the `vertex_loop` can be implemented as a subtype of `G4Curve`. The implementation of all operations is rather straightforward: the tangent is undefined and no ray should intersect the vertex loop. The latter convention only causes problems when the test curve of the inside operation is chosen to go to the vertex loop, as the expected test curve – boundary intersection does not occur. The algorithm must answer "inside" in this case.

Now let us design the class for the whole boundary of the face. The preceding sections showed the need for the following operations:

- It contains a collection of `G4Curves`, as it must hold the bounding loops.
- Parallel projection onto a plane, i.e. projecting the loops one by one. The result is a collection of 2D curves. As 2D and 3D curves implemented by the same classes, so should boundaries; this operation returns a new boundary.
- Intersection in 2D is implemented the same way as the intersection for `composite_curves` in Section 4.6.3.
- Splitting the boundary with a plane. The one half of the result is the part in front of that plane and the other is the part behind the plane.
- Splitting the boundary with a special cylinder. This operation requires that the boundary is located on a torus and the cylinder must be the one containing the path of the centre of the revolved circle.

Splitting deserves a few more words. In the first case:

- The boundary is projected along a direction in the splitting plane. Thus the image of the splitting plane is a line.
- Each curve in the boundary is intersected with this line, i.e. with a ray starting far enough from the curves. Multiple intersections may occur; the ray must be moved forward, in front of the intersections already found and be intersected again.

  Once again, there is the problem of multiple points mapped to the same point, thus we might miss some intersections obscured by other intersections. The solution is to repeat the whole process with another projection and try again. It is likely that the missing intersections are found.
- The curves are split with the intersections and the pieces are classified to be lying on the one or the other side of the line, the image of the splitting plane. The tangent of the curve is used for this purpose; see Section 6.4 for dealing with special cases.

Non-intersecting curves lie entirely on one side of the splitting plane. Classifying an arbitrary point is enough to classify the entire curve.

The other case of splitting with a cylinder is similar. The intersection of the cylinder with the torus which contains the boundary is two parallel circles; these become line segments when projected to a surface containing the symmetry axis of the torus. Unfortunately, these line segments are tangents to the bounding curves, hence the intersection algorithms are allowed to ignore them. To avoid this problem, the line segments are moved towards each other by `kCarTolerance`.

The classification of the pieces coming from the intersection also differs. The 3D tangent is needed, obtainable by the conversions 2D point – parameter value – 3D point – 3D tangent. Classification is based on the fact whether the tangent points away from the symmetry axis or not. For the non-intersected curves, the criterion involves the distance of an arbitrary point from the symmetry axis.

6.4 Precision problems

Precision problems may arise when the test curve meets a bounding loop. Let us call the intersection point $\mathbf{P}$. Some examples from the previous sections:

1. The tangent $\mathbf{t}$ to the loop is undefined at $\mathbf{P}$.
2. The surface normal $\mathbf{n}$ is undefined at $\mathbf{P}$.
3. The left hand side of Definition 6.1 is near 0, i.e. the test curve is very close to being a tangent to the loop at $\mathbf{P}$.

All of these problems can be handled by moving the intersection point to the test curve by a small amount. As the test curve – bounding loop problem is always transformed to a ray – curve intersection in 2D, this amounts to moving the ray by lengths in the magnitude of `kCarTolerance`.

If the tangent is undefined, the ray might intersect the curve or be a tangent to the it, as shown in Figure 6.8. In the first case, it does not matter from which side the tangent is taken, moving the ray by `kCarTolerance` is safe. In the second case, the shift causes either that the tangent of the closer part of the curve is taken or that the ray misses the curve. This is correct behaviour.

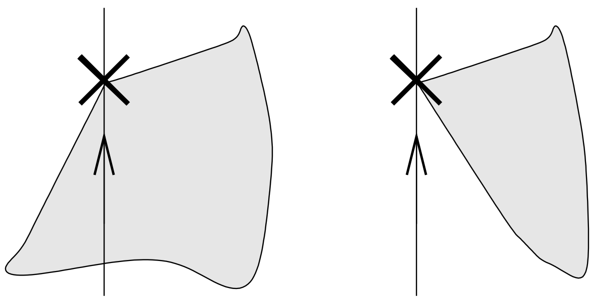


Figure 6.8: The two cases when the loop has no tangent defined at the ray – loop intersection point.

In case the tangent at the new intersection is undefined, the procedure should be repeated. Note that this is unlikely due to the choice of `kCarTolerance`; see Section 3.2.4.

Self intersections of the loop may also cause problems. Fortunately, this condition is easily detectable, as loops may only self intersect at the end points of their constituting curves. The ray is shifted when this occurs, in order to move it away from the end point.

After the tangent was obtained, we can check if the surface normal is defined. If not, the intersection point is moved back along the ray to ensure that it is defined. Entire curves on the surface might have the normal undefined, hence this might fail when the ray lies on such a “sharp edge”. In this case, also the ray should be moved. Figure 6.9 shows which points are tested and in what order. Note that the first candidate already has its normal defined, except for pathological cases, as the fact that the tangent is defined implies that the loop is part of a “sharp edge”.

Finally, the case when the ray is a tangent to the loop should be tested. More precisely, the angle between the ray direction $\mathbf{d}$ and the curve tangent $\mathbf{t}$ $\mathbf{d} \cdot \mathbf{t}$ provided $|\mathbf{d}| = |\mathbf{t}| = 1$, should be $0 \pm$ `kAngTolerance`. This case is analogous to the case of the undefined tangent: wither the ray is a tangent or it is intersecting. Compare Figures 6.8 and 6.10.

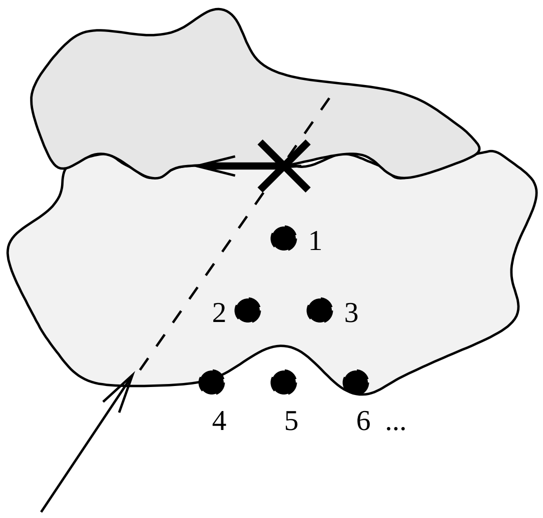


Figure 6.9: Candidate points and the order of testing when the surface normal is undefined at the ray – loop intersection.

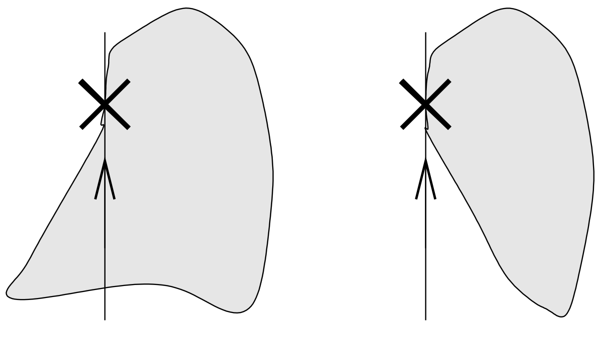


Figure 6.10: The two cases when the ray is locally a tangent to the loop.

Chapter 7

Conclusions

The Boundary Representation based geometric modeller of the particle detector simulation package GEANT4 has been extended with swept surfaces: surfaces of linear extrusion and revolution, as well as with a solution for the problem of classifying a point on a surface as being inside or outside the curves bounding the surface. The implementation of parametric curves has been extended and optimised; this involved a re-design of the curve related code. Thus a major step was taken towards full ISO 10303 STEP compliancy of the geometric modeller.

Throughout the thesis, the following key requirements for GEANT4 were respected when taking decisions:

Efficiency This requirement often resulted in specialising the algorithms for the individual types of STEP entities.

Accuracy This requirement was achieved by systematically considering special cases and the effects of numerical errors.

Development is continuing as the beta release of GEANT4 approaches. Implementation of the surface of revolution must be finalised and classifying points lying on curve bounded NURBS and rectangular composite surfaces must be handled.

Numerous improvement and optimisation ideas have appeared in this thesis. Some additional, more general ideas are listed below. It must be noted that STEP allows for a variety of descriptions for a given object, only a subset of which will be used in the design of detectors. Assessing the value of these ideas is difficult without knowing that particular subset.

Future optimisation efforts shall consider employing numerical methods for NURBS curve and surface intersection. These methods solve the equations describing the intersections directly, rather than try to exploit geometrical properties of the objects involved. They are efficient algorithms in general, but often fail to converge to a solution and are sensitive to issues of numerical stability; see e.g. [MD94]. These algorithms shall be implemented, their performance evaluated in comparison with other approaches, e.g. recursive subdivision and Bézier clipping. Even combining numerical methods with the others may be worthwhile, e.g. an intersection point obtained by subdivision could be polished with numerical methods. Finally, the method considered best should be integrated into GEANT4.

Bounding polygons and polyhedra could be employed instead of bounding rectangles and boxes. Intersecting the ray with the bounding object would become more expensive, but intersecting the ray with the object could be avoided more frequently.

STEP allows for multiple descriptions of the same object. Efficiency of tracking often depends on which representation is chosen; consider a line segment represented with a NURBS curve. The geometric modeller should make an effort to recognise when a conversion to a cheaper representation is possible.

Appendix A

Geant4 class categories

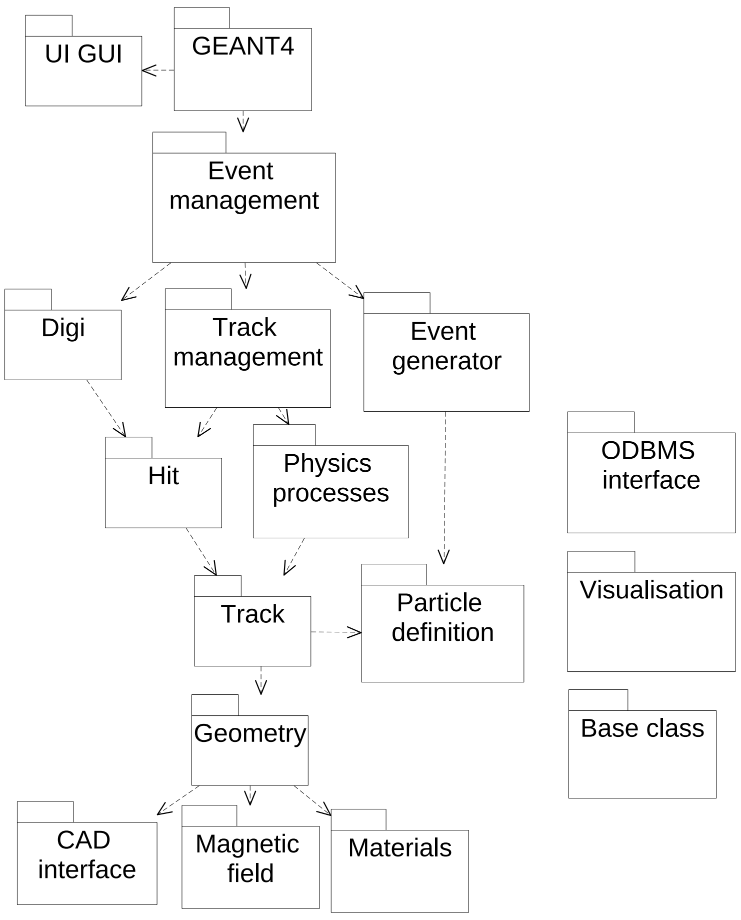


Figure A.1: Categories and their interactions in GEANT4 — coarse view.

Appendix B

Design diagrams of the B-Rep code

These diagrams provide an overview of the development presented in this thesis. All the appearing classes are within the Geometry category, B-Rep sub-category.

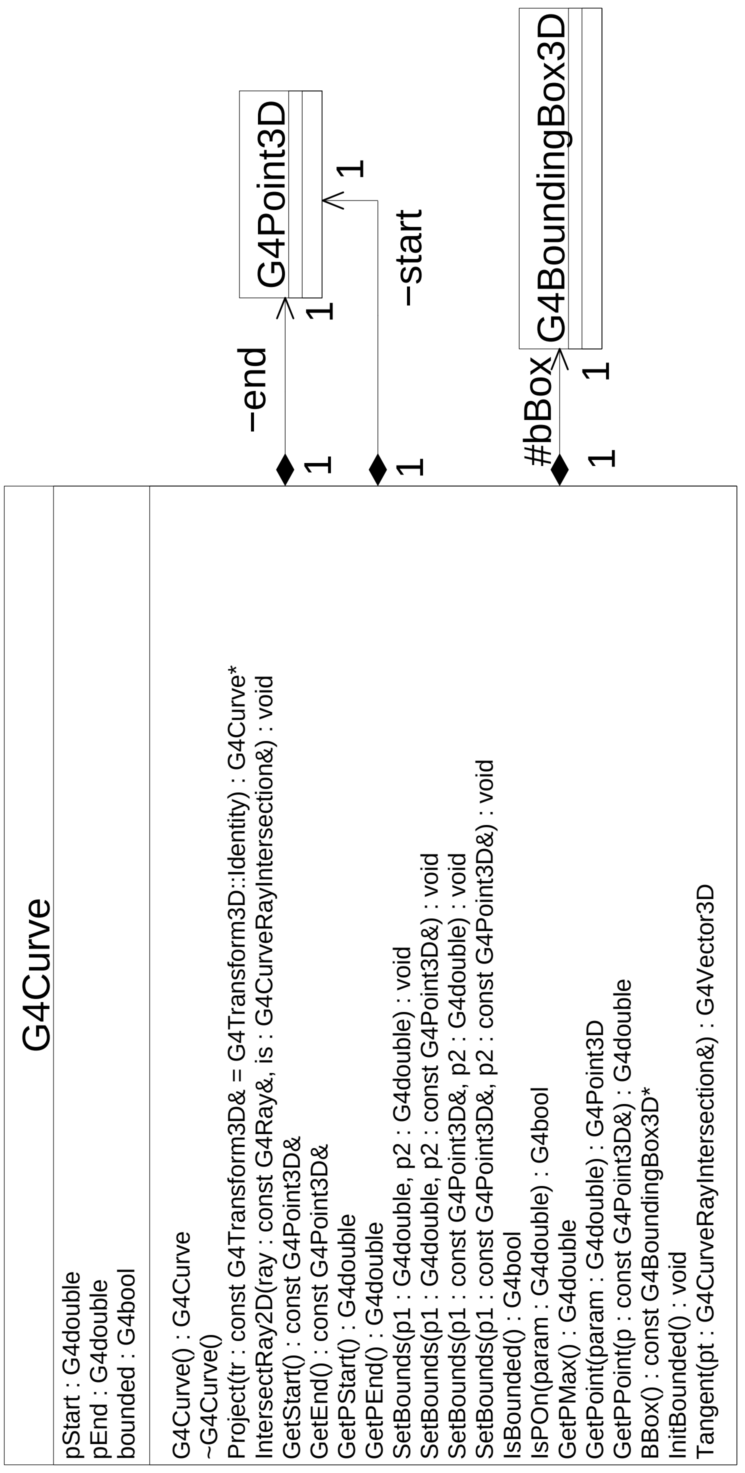
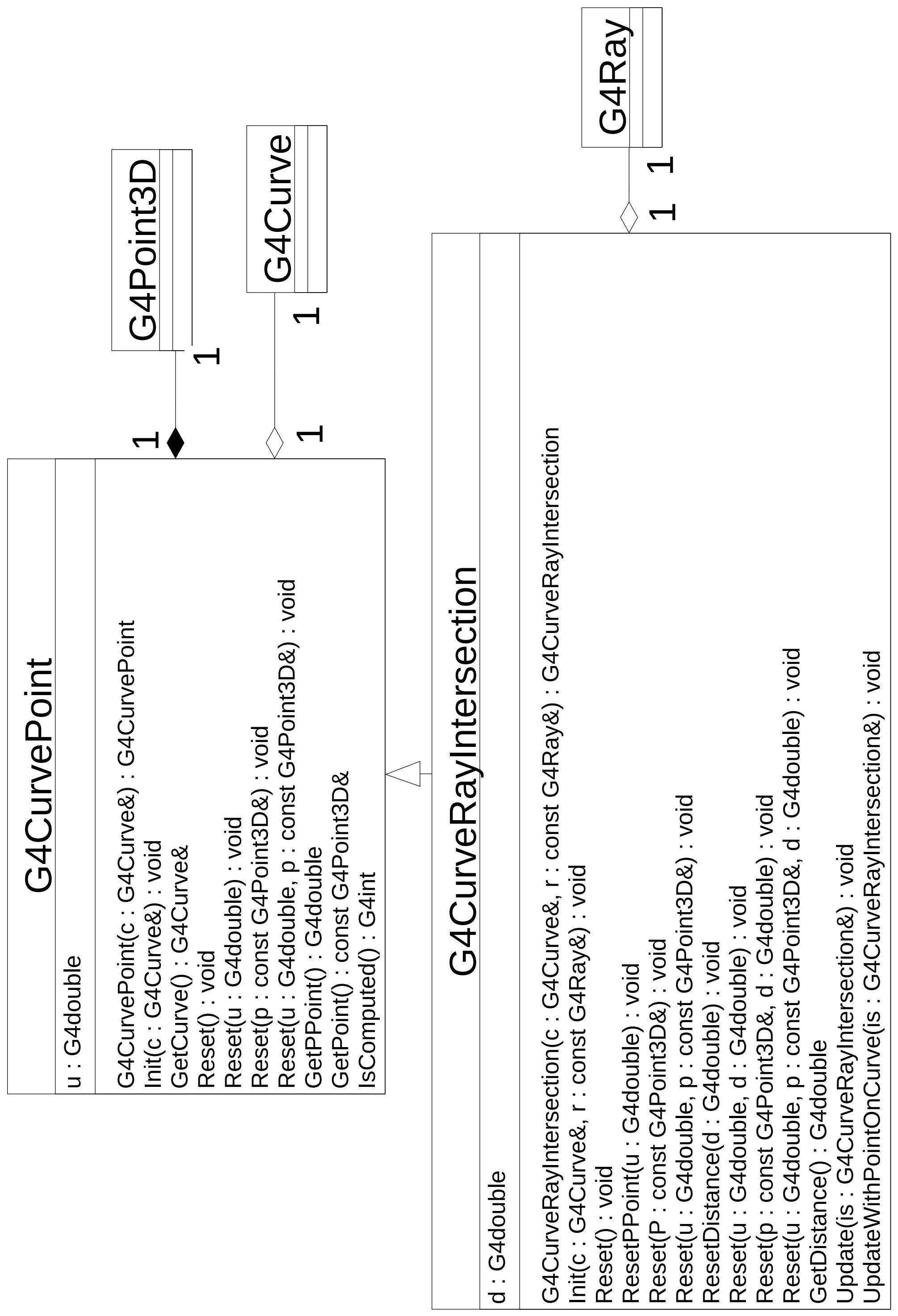

G4Curve
pStart : G4double
pEnd : G4double
bounded : G4bool
G4Curve() : G4Curve
~G4Curve()
Project(tr : const G4Transform3D& = G4Transform3D::Identity) : G4Curve*
IntersectRay2D(ray : const G4Ray&, is : G4CurveRayIntersection&) : void
GetStart() : const G4Point3D&
GetEnd() : const G4Point3D&
GetPStart() : G4double
GetPEnd() : G4double
SetBounds(p1 : G4double, p2 : G4double) : void
SetBounds(p1 : G4double, p2 : const G4Point3D&) : void
SetBounds(p1 : const G4Point3D&, p2 : G4double) : void
SetBounds(p1 : const G4Point3D&, p2 : const G4Point3D&) : void
IsBounded() : G4bool
IsPOn(param : G4double) : G4bool
GetPMax() : G4double
GetPoint(param : G4double) : G4Point3D
GetPPoint(p : const G4Point3D&) : G4double
BBox() : const G4BoundingBox3D*
InitBounded() : void
Tangent(pt : G4CurveRayIntersection&) : G4Vector3D
G4Point3D
−end
1
1
−start
1
1
G4BoundingBox3D
#bBox
1
1

Figure B.1: Interface of the base class for curves.



G4CurvePoint
u : G4double
G4CurvePoint(c : G4Curve&) : G4CurvePoint
Init(c : G4Curve&) : void
GetCurve() : G4Curve&
Reset() : void
Reset(u : G4double) : void
Reset(p : const G4Point3D&) : void
Reset(u : G4double, p : const G4Point3D&) : void
GetPPoint() : G4double
GetPoint() : const G4Point3D&
IsComputed() : G4int
G4Point3D
G4Curve
G4Ray
1
1
1
1
1
1
G4CurveRayIntersection
d : G4double
G4CurveRayIntersection(c : G4Curve&, r : const G4Ray&) : G4CurveRayIntersection
Init(c : G4Curve&, r : const G4Ray&) : void
Reset() : void
ResetPPoint(u : G4double) : void
Reset(P : const G4Point3D&) : void
Reset(u : G4double, p : const G4Point3D&) : void
ResetDistance(d : G4double) : void
Reset(u : G4double, d : G4double) : void
Reset(p : const G4Point3D&, d : G4double) : void
Reset(u : G4double, p : const G4Point3D&, d : G4double) : void
GetDistance() : G4double
Update(is : G4CurveRayIntersection&) : void
UpdateWithPointOnCurve(is : G4CurveRayIntersection&) : void

Figure B.2: Classes managing the two representations of a point on a curve and the three representations of a curve – ray intersection point.

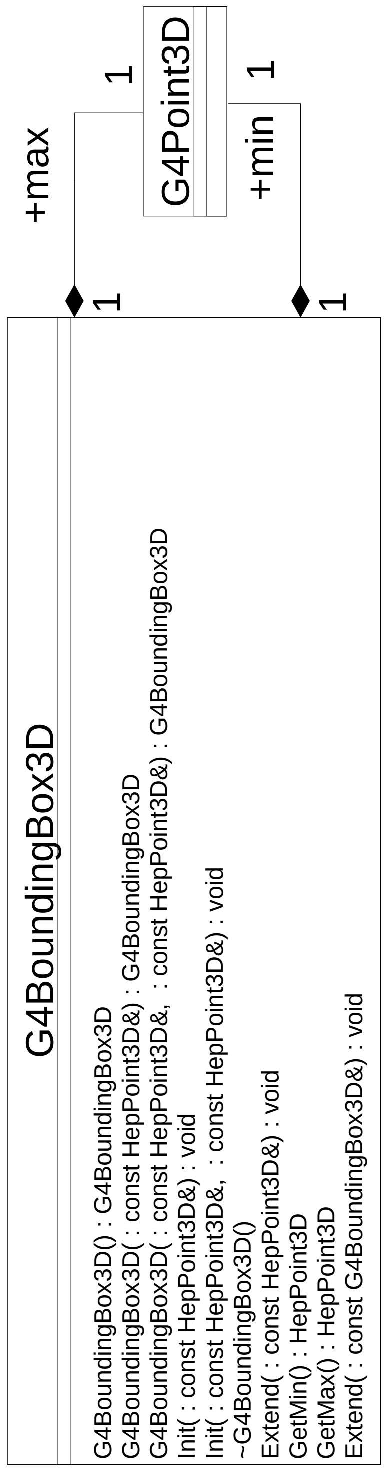

G4BoundingBox3D
G4BoundingBox3D() : G4BoundingBox3D
G4BoundingBox3D(: const HepPoint3D&) : G4BoundingBox3D
G4BoundingBox3D(: const HepPoint3D&, : const HepPoint3D&) : G4BoundingBox3D
Init(: const HepPoint3D&) : void
Init(: const HepPoint3D&, : const HepPoint3D&) : void
~G4BoundingBox3D()
Extend(: const HepPoint3D&) : void
GetMin() : HepPoint3D
GetMax() : HepPoint3D
Extend(: const G4BoundingBox3D&) : void
G4Point3D
+max
+min
1
1
1
1

Figure B.3: Class storing a bounding box.

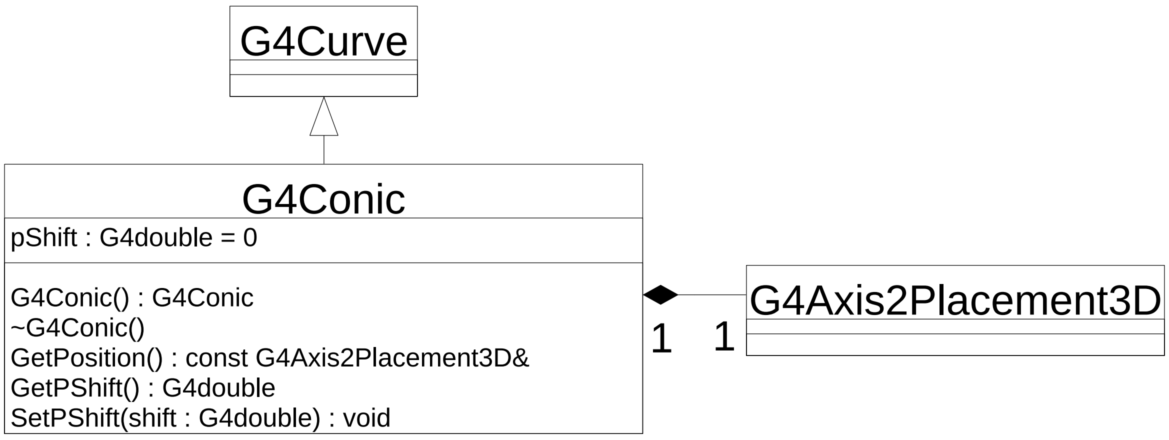


Figure B.4: Base class for conics.

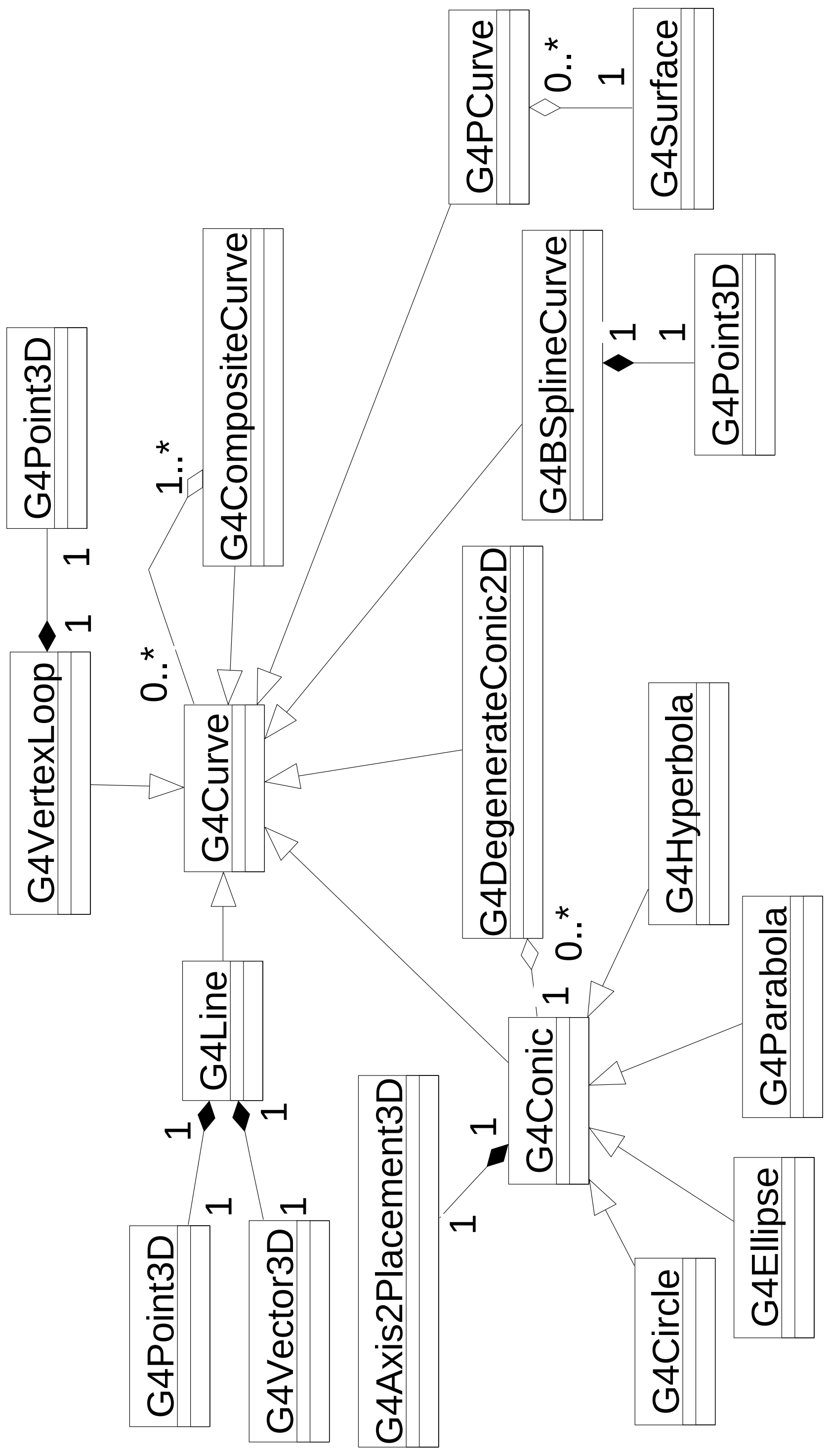


Figure B.5: The inheritance tree of curve classes.

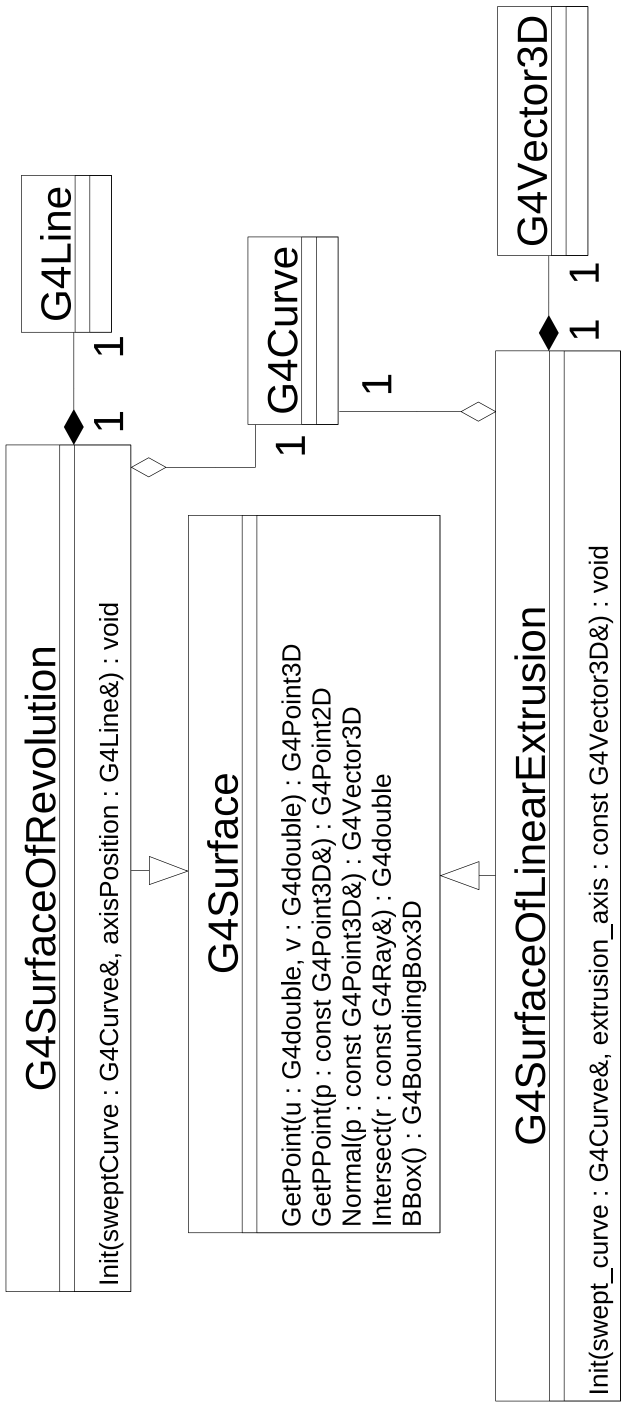

G4SurfaceOfRevolution
Init(sweptCurve : G4Curve&, axisPosition : G4Line&) : void
G4Line
1
1
G4Curve
1
1
G4Surface
GetPoint(u : G4double, v : G4double) : G4Point3D
GetPPoint(p : const G4Point3D&) : G4Point2D
Normal(p : const G4Point3D&) : G4Vector3D
Intersect(r : const G4Ray&) : G4double
BBox() : G4BoundingBox3D
G4SurfaceOfLinearExtrusion
Init(swept_curve : G4Curve&, extrusion_axis : const G4Vector3D&) : void
G4Vector3D
1
1

Figure B.6: Swept surfaces with the most important operations.

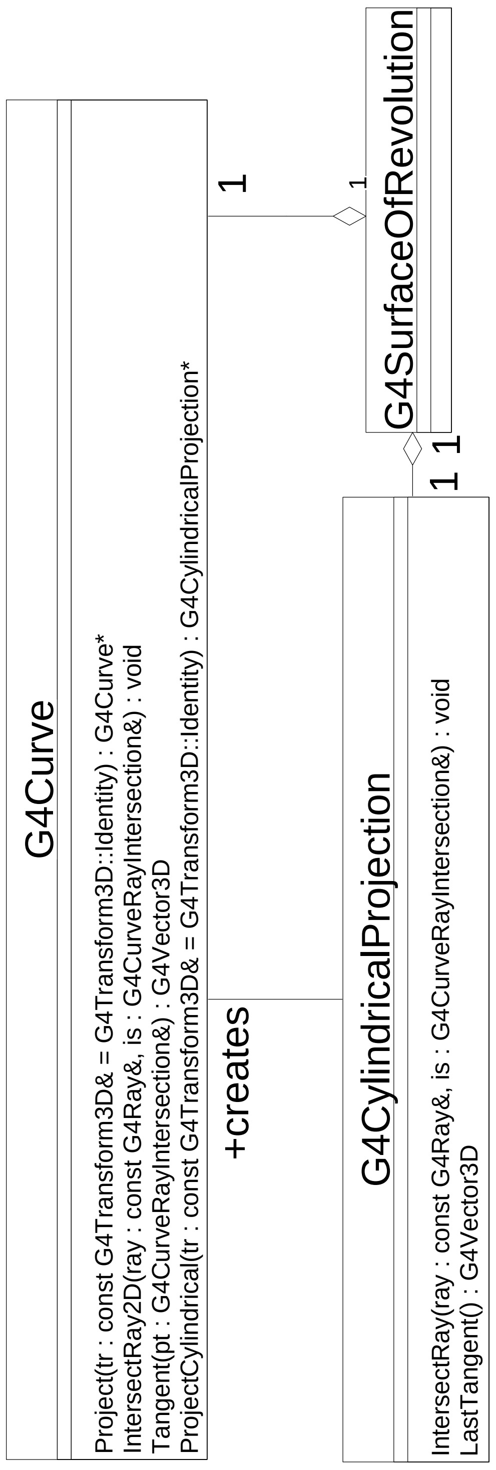

G4Curve
Project(tr : const G4Transform3D& = G4Transform3D::Identity) : G4Curve*
IntersectRay2D(ray : const G4Ray&, is : G4CurveRayIntersection&) : void
Tangent(pt : G4CurveRayIntersection&) : G4Vector3D
ProjectCylindrical(tr : const G4Transform3D& = G4Transform3D::Identity) : G4CylindricalProjection*
1
1
G4SurfaceOfRevolution
1 1
+creates
G4CylindricalProjection
IntersectRay(ray : const G4Ray&, is : G4CurveRayIntersection&) : void
LastTangent() : G4Vector3D

Figure B.7: Projection, 2D intersection and tangent calculation of a surface of revolution.

Appendix C

STEP class diagrams

The class diagrams present the important geometric and topological entities in [STEP203]. They were reverse engineered from the C++ translation of the original EXPRESS specification. Only generalisation relationships are shown. It is interesting to compare these diagrams with the ones in Chapter B.

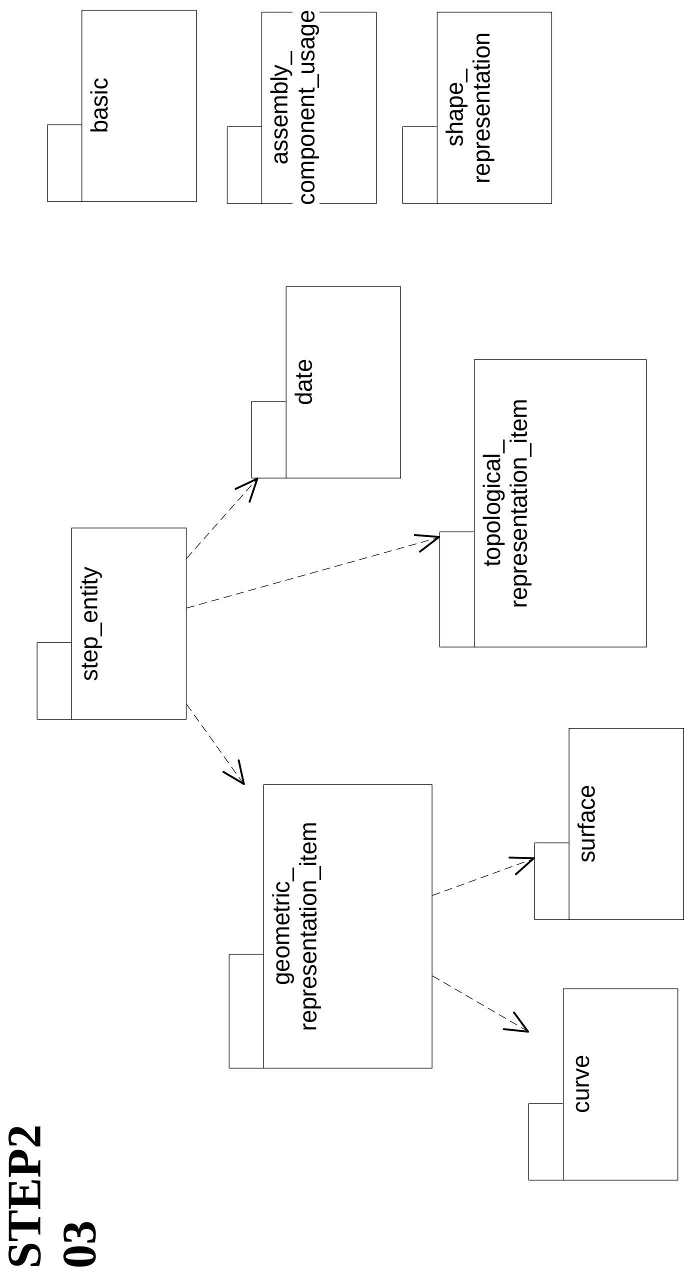

STEP203
basic
assembly_ component_usage
shape_ representation
step_entity
date
topological_ representation_item
geometric_ representation_item
surface
curve

Figure C.1: Structure of application protocol Configuration Controlled Design.

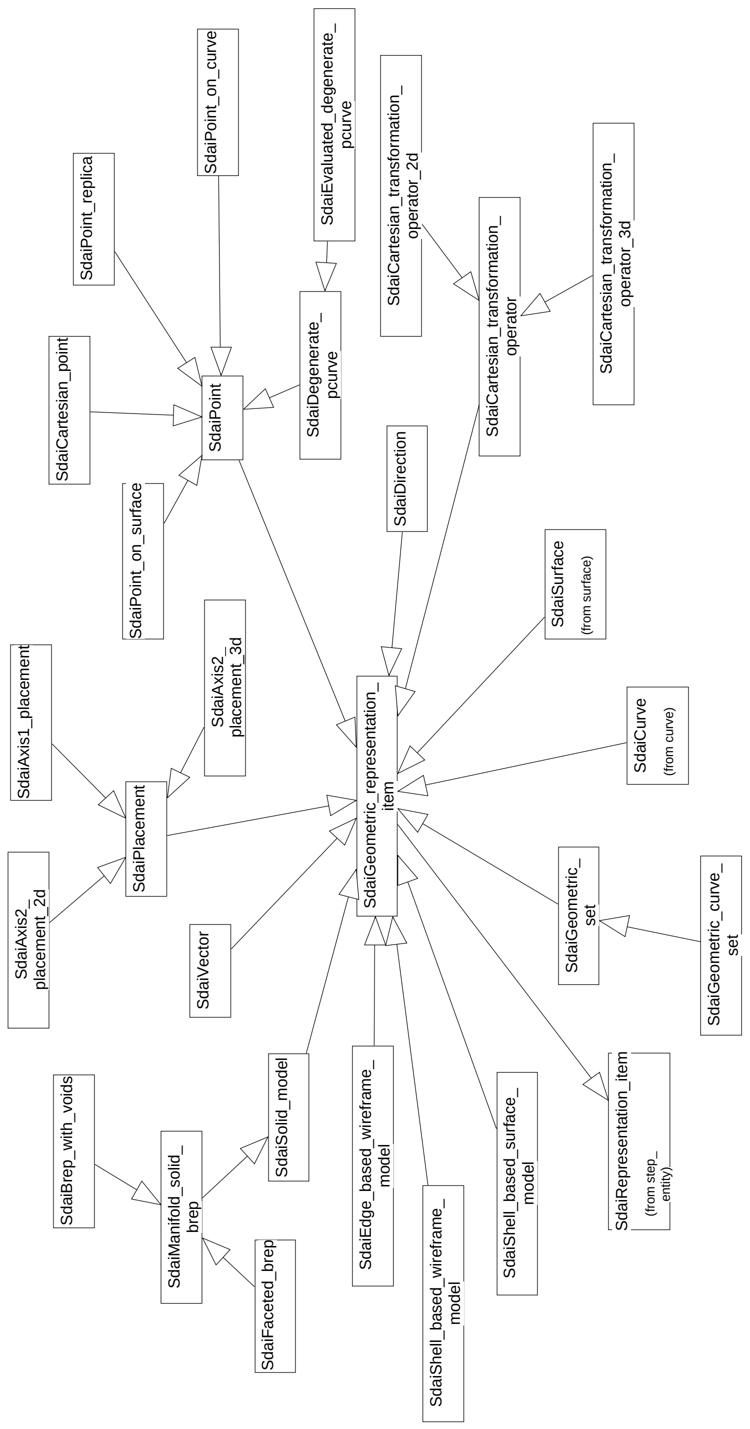


Figure C.2: Geometric entities.

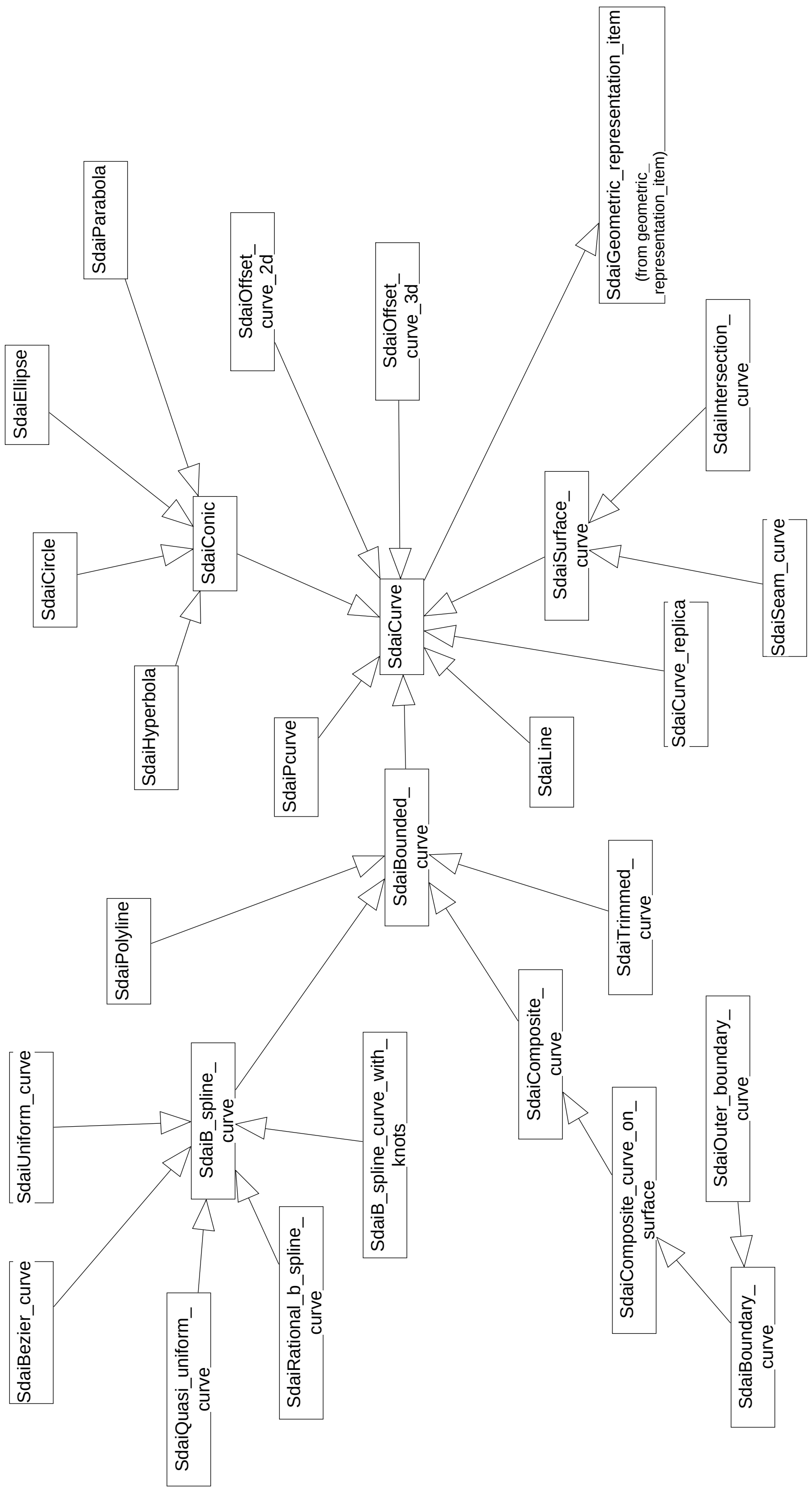


Figure C.3: Curves.

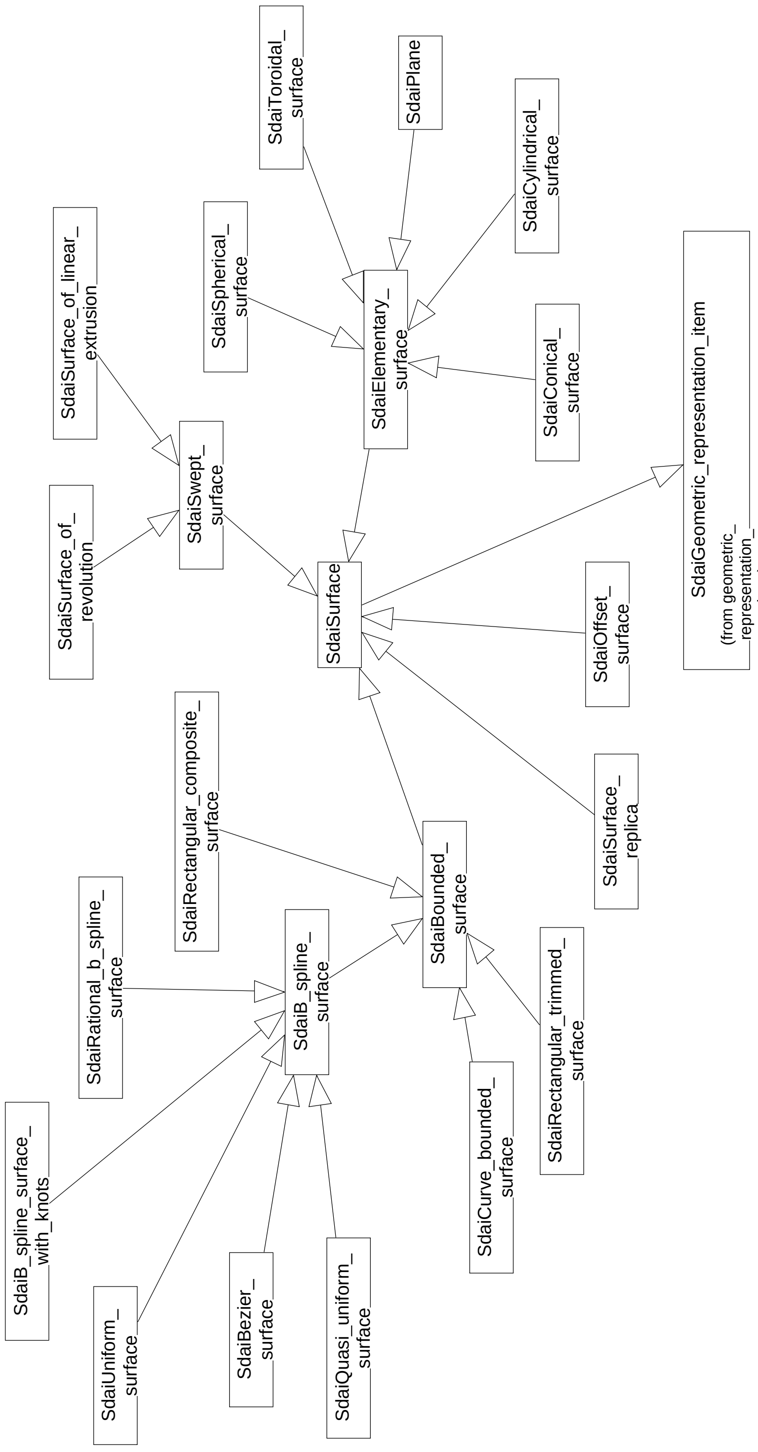

SdaiToroidal_ surface
SdaiPlane
SdaiCylindrical_ surface
SdaiSpherical_ surface
SdaiElementary_ surface
SdaiConical_ surface
SdaiSurface_of_linear_ extrusion
SdaiSwept_ surface
SdaiSurface_of_ revolution
SdaiSurface
SdaiGeometric_representation_item (from geometric_ representation_ item)
SdaiOffset_ surface
SdaiSurface_ replica
SdaiRectangular_composite_ surface
SdaiBounded_ surface
SdaiRational_b_spline_ surface
SdaiB_spline_ surface
SdaiRectangular_trimmed_ surface
SdaiCurve_bounded_ surface
SdaiB_spline_surface_ with_knots
SdaiUniform_ surface
SdaiBezier_ surface
SdaiQuasi_uniform_ surface

Figure C.4: Surfaces.

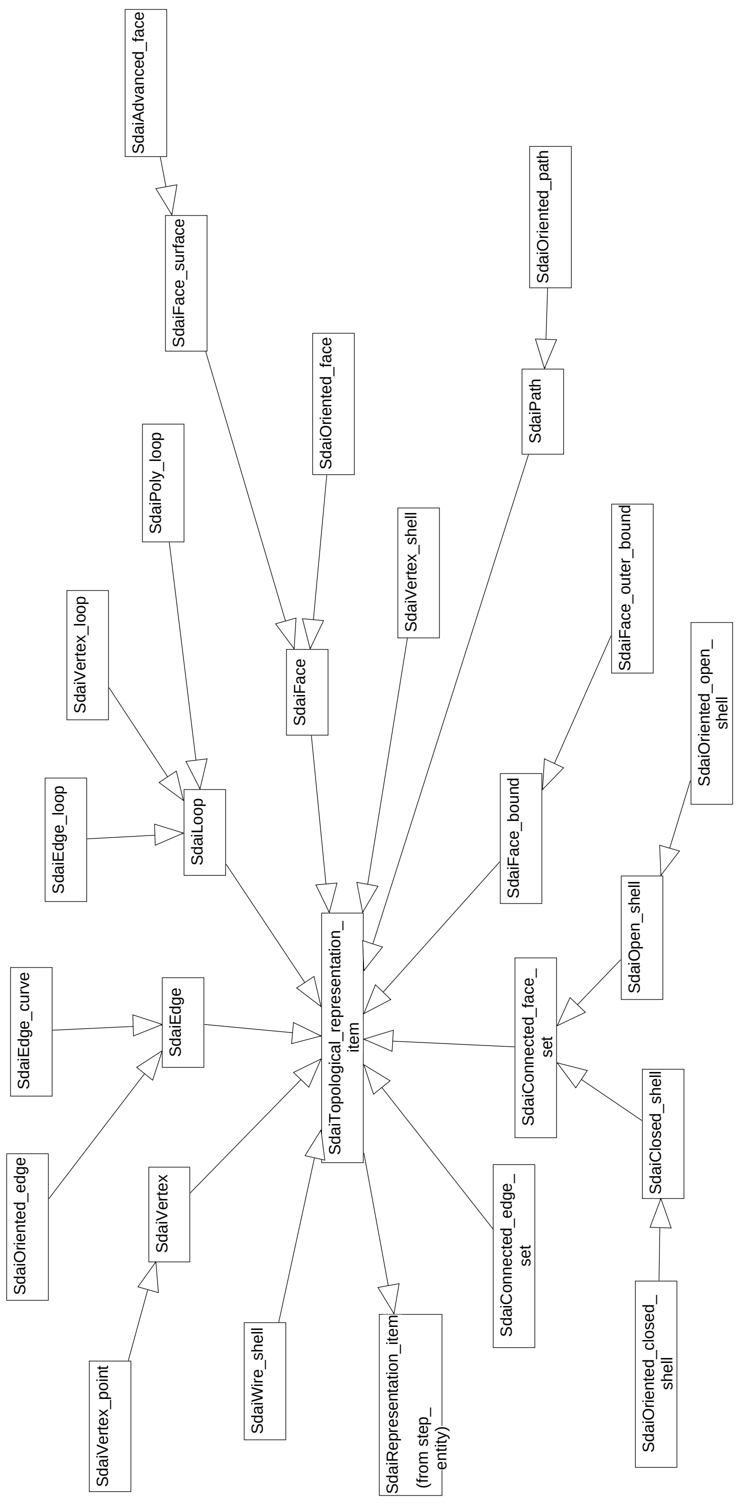


Figure C.5: Topological entities.

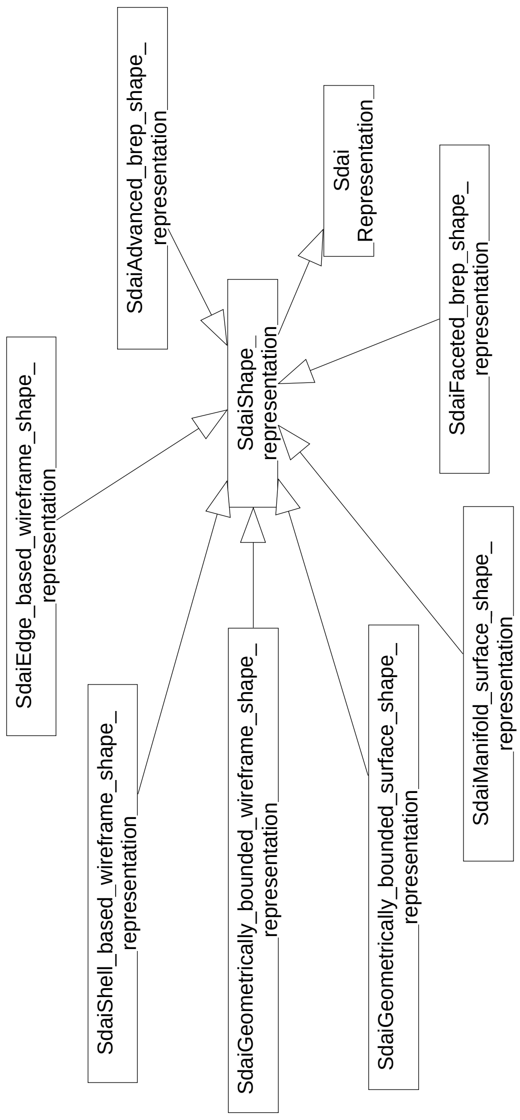


Figure C.6: Possible shape representations.

Bibliography

[AGGon] John Apostolakis, Simone Giani, and Vladimir Grichine. Particle propagation in a magnetic field in GEANT4. GEANT4 working note, in preparation.

[App93] Application Software Group, IT Division, CERN, Geneva, Switzerland. GEANT – *Detector Description and Simulation Tool, CERN Program Library Long Writeup W5013*, 1993. See the URL `http://wwwcn.cern.ch/asdoc/geant_html3/-geantall.html`.

[BBB87] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An Introduction to Splines for Computer Graphics and Geometric Modeling.* Morgan Kaufmann, Los Altos, California, 1987. Some basic techniques on ray tracing splines (see Sweeney).

[BK85] Willem F. Bronsvoort and Fopke Klok. Ray tracing generalized cylinders. *ACM Transactions on Graphics*, 4(4):291–303, October 1985. See corrigendum [BK87].

[BK87] Willem F. Bronsvoort and Fopke Klok. Corrigendum: "Ray Tracing Generalized Cylinders". *ACM Transactions on Graphics*, 6(3):238–239, July 1987. See [BK85].

[Boo93] G. Booch. *Object-Oriented Analysis and Design with Applications, Second Edition.* Benjamin/Cummings, 1993.

[Far93] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, Boston, 3. edition, 1993.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Co., Reading, MA, 2nd edition, 1990. Inside – outside algorithms on pp. 964–965.

[G$^+$95] Simone Giani et al. GEANT4: Simulation for the next generation of HEP experiments. In *Proceedings of the International Conference on Computing in High Energy Physics (CHEP)*, Rio de Janeiro, Brazil, 1995. See the URL `http://www.hep.net/-conferences/chep95/html/abstract/abs_8.htm`.

[G$^+$97a] Simone Giani et al. GEANT4: A world-wide collaboration to build object oriented hep simulation software. In *Proceedings of the International Conference on Computing in High Energy Physics (CHEP)*, Berlin, Germany, 1997. See the URL `http://www.ifh.de/CHEP97/pae_sess.htm#E3`.

[G$^+$97b] Simone Giani et al. GEANT4: An object-oriented toolkit for simulation in HEP. Status report, CERN, Geneva, Switzerland, 1997.

[Gla89] Andrew S. Glassner, editor. *An Introduction to Ray Tracing.* Academic Press, 1989.

[Kaj83] J. T. Kajiya. New techniques for ray tracing procedurally defined objects. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, volume 17/3, pages 91–102, July 1983.

[Ken95a] Paul Kent. Minimising precision problems in GEANT4 geometry. GEANT4 working note, 1995.

[Ken95b] Paul Kent. Pure tracking and geometry in GEANT4. GEANT4 working note, 1995.

[Lib93] Don Libes. The NIST EXPRESS toolkit — design and implementation. In *Proceedings of the Seventh Annual ASME Engineering Database Symposium*, San Diego, California, August 9-11 1993.

[MD94] Dinesh Manocha and James Demmel. Algorithms for intersecting parametric and algebraic curves I: Simple intersections. *ACM Transactions on Graphics*, 13(1):73–99, January 1994. ISSN 0730-0301.

[NSK90] Tomoyuki Nishita, Thomas W. Sederberg, and Masanori Kakimoto. Ray tracing trimmed rational surface patches. *Computer Graphics*, 24(4):337–345, August 1990.

[PP86] Michael A. Penna and Richard R. Patterson. *Projective Geometry and its Applications to Computer Graphics.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1986.

[PT89] Leslie Piegl and Wayne Tiller. A menagerie of rational B-spline circles. *IEEE Computer Graphics and Applications*, 9(5):48–56, September 1989.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.).* Cambridge University Press, Cambridge, 1992. ISBN 0-521-43108-5.

[RHD89] Alyn Rockwood, Kurt Heaton, and Tom Davis. Real-time rendering of trimmed surfaces. *Computer Graphics*, 23(3):107–116, July 1989.

[STEP1] ISO 10303-1. Overview and fundamental principles. In *Industrial automation systems and integration – Product data representation and exchange.* ISO TC 184/SC4, 1992.

[STEP11] ISO 10303-11. Description methods: The EXPRESS language reference manual. In *Industrial automation systems and integration – Product data representation and exchange.* ISO TC 184/SC4, 1992.

[STEP42] ISO 10303-42. Integrated generic resources: Geometric and topological representation. In *Industrial automation systems and integration – Product data representation and exchange.* ISO TC 184/SC4, 1992.

[STEP43] ISO 10303-43. Integrated generic resources: Representation structures. In *Industrial automation systems and integration – Product data representation and exchange.* ISO TC 184/SC4, 1992.

[STEP203] ISO 10303-203. Application protocol: Configuration controlled design. In *Industrial automation systems and integration – Product data representation and exchange.* ISO TC 184/SC4, 1992.

[STEP214] ISO 10303-214. Application protocol: Core data for automotive mechanical design processes. In *Industrial automation systems and integration – Product data representation and exchange.* ISO TC 184/SC4, 1997. Draft. See the URL `http://www.nist.gov/sc4/step/parts/part214/current/`.

[Sul95] Jari Sulkimo. Modelling physics detectors for simulation purposes using boundary representation. Master's thesis, Department of Computer Science, University of Tampere, Finland, 1995.

[UML] *The Unified Modeling Language WWW site.* `http://www.rational.com/uml`.

[Vuo96] Jouko Vuoskoski. *Exchange of Product Data between CAD Systems and a Physics Simulation Program.* PhD thesis, Tampere University of Technology, Finland, 1996.

[vW84] Jarke J. van Wijk. Ray tracing objects defined by sweeping planar cubic splines. *ACM Transactions on Graphics*, 3(3):223–237, July 1984.

[WNDO97] Mason Woo, Jackie Neider, Tom Davis, and OpenGL Architecture Review Board. *OpenGL programming guide: the official guide to learning OpenGL, version 1.1.* Addison-Wesley, Reading, MA, USA, 1997.