vertices. The sixth line states that the faces are described by 16 integers. The last four lines describe the four faces.

```
4
2.3 0 0
-1.2 1.6 0
-1.2 -1.6 0
0 0 2.7
16
0 1 2 -1
1 0 3 -1
3 0 2 -1
2 1 3 -1
```

Geo object	**description**
s_coord	*floating point number*
pl_angle	value()
pl_fixedvec	x() \s y()
pl_point	vec()
pl_edge	begin_vt() \s end_vt()
pl_ray	begin_vt() \s direction()
pl_line	ref_vt() \s direction()
pl_box	xmin() \s ymin() \s xmax() \s ymax()
pl_polygon	size() $\forall i$ {\n vertex(i)}
pl_disc	center() \s radius()
pl_arc	center() \s radius() \s begin_angle() \s enclosed_angle()
sp_fixedvec	x() \s y() \s z()
sp_point	vec()
sp_edge	begin_vt() \s end_vt()
sp_ray	begin_vt() \s direction()
sp_line	ref_vt() \s direction()
sp_box	xmin() \s ymin() \s zmin() \s xmax() \s ymax() \s zmax()
sp_polygon	size() $\forall i$ {\n vertex(i)}
sp_polyhedron	vertex_count() \n $\forall i$ {\n vertex(i)} n (the number of following integers) $\forall i$ { $\forall j$ {\s index(i, j)} \s -1 }
sp_sphere	center() \s radius()
sp_disc	center() \s supporting_plane().normal() \s radius()
sp_arc	supporting_plane() \s planar_projection()

Appendix A

IO Format

In this appendix the file format for PlaGeo and SpaGeo objects is described. The files are written in ASCII format. The only primitives that are written to file are integers, floats, spaces and newlines. The last two are denoted by `\s` and `\n` in this appendix.

In the table below, class functions are used to describe the format. The result of those functions are either a number, in which case this number is written, or another geometrical object, in which case this object is written.

For example, take a line with reference vertex (1.7, 5.1) and direction (-0.6, 0.8). According to the table, a line is written as its reference vertex (a `pl_vec`) followed by a space followed by its direction (a `pl_unitvec`). The vector and unit vector are both derived from `pl_fixedvec`, so are written as the x coordinate followed by a space and the y coordinate. The resulting output will be:

```
1.7 5.1 -0.6 0.8
```

Note that nothing is printed after the final digit. The user should supply spaces, tabs and newlines to separate the consecutive primitives. This is in line with normal output conventions, where no space is added after `int`s or `float`s.

Remarks

The derivatives of `pl_fixedvec` –`pl_vec` and `pl_unitvec`– have the same format as `pl_fixedvec` itself. The same holds for `sp_fixedvec` and its derivatives `sp_vec` and `sp_unitvec`.

When reading a unit vector, it is checked whether the vector really has length 1 (allowing for a small error margin). If this is not the case, a reading error results. This is important for lines and rays, where a unit vector is needed for the direction.

A box can be empty. In that case the the maximum values should be smaller than the minimum values, preferably 1 1 –1 –1 for a planar box and 1 1 1 –1 –1 –1 for a spatial box.

Examples

Here is an example file that contains a tetrahedron (a polyhedron with four vertices). The first line states that there are four vertices. The next four lines give the coordinates of the

Part IV

Appendices

```
int intersect(sp_vec &witness, sp_polygon pgn, sp_ray ray)

int intersect(sp_vec &witness, sp_polygon pgn, sp_edge edge)

int intersect(sp_edgereray &witnesses, sp_polygon pgn, sp_plane plane)

int intersect(sp_edgereray &witnesses, sp_polygon pgn1, sp_polygon pgn2)
```

Finally there is a routine for the case of two spheres.

```
int intersect(sp_disc &witness, sp_sphere sphere1, sp_sphere sphere2)
```

14.2 Distances

An important concept is the distance or clearance between two objects. The clearance is defined as the length of the shortest line segment that joins a point of one object to a point of the other. There are routines to compute the clearance for all pairs of objects.[1]

There are two ways to ask for the clearance, having the following form:

```
s_coord clearance sp_line line, sp_ray ray)
s_coord clearance sp_line line, sp_ray ray, s_coord bound)
```

They both compute the clearance between, in this case, a line and a ray. The second routine takes an extra parameter, the bound. If the clearance is bigger than this bound, the bound will be returned instead of the clearance. This gives the possibility of optimizations: it is often easier to decide that an object lies far away than to compute the exact distance.

There are also some routines especially for optimization purposes. They will normally not be used by the user of SpaGeo, but may be handy if some new object types are created and efficient routines need to be written for them. The following routines make a quick and dirty test if two objects lie far apart:

```
int certain_clearance(sp_box box1, sp_box box2, s_coord bound)
int certain_clearance(sp_fixedvec vt, sp_box b, s_coord bound)
```

If those routines return true, the two bounding boxes or the bounding box and the vertex lie further than `bound` apart. If they return false, not much can be said: the objects might be closer together than bound or further apart than bound, though not very much further. What, in fact, is done, is that for each coordinate the distance in that direction is compared with `bound`. This is cheaper than comparing Pythagorean distances and catches a lot of cases when things are far away.

[1] Currently, the clearance between an arc and another object is not implemented

Although for completeness sake every pair of objects can be passed as parameter, some combinations do not make sense, because of the inexact computations of SpaGeo.

For example, the two objects can be 'too big', so the answer is always 'yes'. This is the case with two planes or with a plane and a line. Those objects extend indefinitely; the only possibility for them not to intersect is to be exactly parallel. No attempt is made to check this.

On the other hand, the objects can be 'too small'. The sum of the dimensions of the two objects should be at least 3, otherwise the answer will always be 'no'. For `do_cross` only the boundaries of three dimensional objects are considered, their dimension should be taken as two in this case. So, according to SpaGeo, two edges do never overlap (not even an edge with itself), because the sum of their dimensions is 2. A zero dimensional object will never cross another object.

Reporting Routines

There are some routines that report the intersection area between two objects. They are always of the form:

```
int intersect(results ..., first_object, second_object)
```

All routines that are mentioned here may also be called with the last two arguments swapped.

When an intersection routine returns 0, no intersections were found. The result does not contain anything useful. Otherwise, the result describes the intersection area (this is called the witness). When two results are returned (as is the case with an arc and a plane), the return value 1 indicates that only the first witness is valid, the value 2 indicates that only the second witness is valid and the value 3 indicates that both are valid.

Here is a list of all intersection routines where on of the objects is a plane.

```
int intersect(sp_vec &witness, sp_plane plane  sp_line line)

int intersect(sp_vec &witness, sp_plane plane, sp_ray ray)

int intersect(sp_vec &witness, sp_plane plane, sp_edge edge)

int intersect(sp_line &witness, sp_plane plane1, sp_plane plane2)

int intersect(sp_edgereray &witnesses, sp_plane plane, sp_polygon pgn)

int intersect(sp_disc &witness, sp_plane plane, sp_sphere sphere)

int intersect(sp_edge &witness, sp_plane plane, sp_disc disc)

int intersect(sp_vec &wn1, sp_vec &wn2, sp_plane plane, sp_arc arc)
```

Here is a list of all intersection routines where on of the objects is a polygon.

```
int intersect(sp_vec &witness, sp_polygon pgn, sp_line line)
```

Chapter 14

High Level Routines

14.1 Intersections

SpaGeo has a number of routines to detect if two geometric objects intersect. There is one group of routines that answer the decision question only (is there an intersection or not) and another group that, in addition, gives information where this intersection takes place. The first group is complete, in the sense that for every pair of objects such a question can be asked. The second group is not complete. Sometimes the answer would be too trivial (for example, the intersection area of a point and a polygon is either the point or empty), sometimes the answer would be too complicated (for example the intersection area for a polygon and a disc).

Decision Routines

In order to describe the various routines that answer if two geometrical objects intersect, it is convenient to divide the objects in three groups:

zerodim	onedim	twodim	threedim
sp_point	sp_edge	sp_plane	sp_box
sp_fixedvec	sp_ray	sp_polygon	sp_polyhedron
	sp_line	sp_disc	sp_sphere
	sp_arc		

There are two different kinds of intersection query routines:

```
int do_overlap(sp_object obj1, sp_object obj2)
int do_cross(sp_object obj1, sp_object obj2)
```

Both routines can not only take `sp_objects`, but also objects of type `sp_fixedvec` (giving the same result as `sp_point`) and `sp_box`.

The first routine, `do_overlap`, checks if the two objects have a point in common. Three dimensional objects are considered to be filled, so if an object is totally contained in another object, `do_overlap` will return true (1). This is the difference with `do_cross`, which only takes into account the boundary of three dimensional objects. When neither of the objects is three dimensional, the result of `do_overlap` and `do_cross` is always the same.

`c.index(i,j)` returns the vertex index of the `j`th index of the `i`th polygon.

`c.is_ok()` checks the requirements for being well defined. Returns 1 if `c` fulfills the requirements, otherwise 0.

`c.is_topologically_ok()` checks only the topological requirements.

`c.make_right_oriented()` A right oriented polyhedron is a polyhedron of which all the faces are right oriented, that is, when looked upon from the outside the vertices are enumerated in counterclockwise order. This routine shuffles the indices of the faces such that `c` will become right oriented.

`c.an_edge_on_vertex(i)` returns the index of an edge having `c.vertex(i)` as one of its endpoints.

`c.an_edge_on_face(int i)` returns the index of an edge that is part of the boundary of `c.polygon(i)`.

`c.next_edge_on_vertex(vt, edge)` Assuming that `c.vertex(vt)` is an endpoint of `c.edge(edge)`, this routine returns the index of the next edge that is incident on this vertex. If `c` is right oriented, the edges will be enumerated in counterclockwise order (seen from the outside).

`c.next_edge_on_face(face, edge)` Assuming that `c.edge(edge)` is on the boundary of `c.polygon(face)`, this routine returns the index of the next edge that is incident on this face. If `c` is right oriented, the edges will be enumerated in counterclockwise order (seen from the outside).

`c.vertex1(edge)` returns the index of the vertex that is one of the endpoints of `c.edge(edge)`.

`c.vertex2(edge)` returns the index of the other endpoint of `c.edge(edge)`.

`c.opposite_vertex(edge, vt)` Assuming that `c.vertex(vt)` is an endpoint of `c.edge(edge)`, this routine returns the index of the other endpoint.

`c.face1(edge)` returns the index of one of the polygons incident on `c.edge(edge)`.

`c.face2(edge)` returns the index of the other polygon incident on `c.edge(edge)`.

`c.opposite_face(edge,face)` Assuming that `c.polygon(face)` is incident on `c.edge(edge)`, this function returns the index of the other face incident on this edge.

Examples

Description

The type `sp_polyhedron` or `sp_phdn` for short is used to describe polyhedra. Polyhedra are objects bounded by polygonal faces. The faces must form a closed surface that divides the space in two parts. The finite part is called the inside, the infinite part is called the outside.

Constructors

The first constructor makes an empty polyhedron. The second constructor takes two parameters. The first parameter is a reray of all `pl_vecs` –the vertices of the polyhedron. The second parameter is a reray of integers –indices in the vertex reray– describing the connectivity. Every polygonal face is described by enumerating its vertices. Every face is concluded by an end marker, the value −1. For example, the unit cube can be described as follows:

```
vertices:
(<0,0,0>, <1,0,0>, <1,1,0>, <0,1,0>, <0,0,1>, <1,0,1>, <1,1,1>, <0,1,1>)
indices:
(0, 1, 2, 3, -1, 4, 5, 6, 7, -1, 0, 1, 5, 4, -1,
1, 2, 6, 5, -1,  2, 3, 7, 6, -1, 3, 0, 4, 7, -1)
```

There are some requirements for a polyhedron to be well defined. All the indices must be inside the bounds of the reray of vertices. Every edge (two consecutive vertices in the index list) must occur exactly twice, so it bounds two faces. The vertices and edges of the polyhedron must form a connected graph. The vertices of a face must all lie in a plane. Faces may not intersect other faces (except on the edges).

The first three requirements can be checked without knowing the position of the vertices. They are called the topological requirements. The two last requirements are geometric in nature.

Class Functions

`c.face_count()` returns the number of faces of `c`. The faces are numbered from 0 until `face_count()-1`.

`c.polygon(i)` returns face `i` as a polygon.

`c.vertex_count()` returns the number of vertices of `c`.

`c.facevertex_count()` returns the number of vertices of `c`, where every vertex is counted once for every face that it appears in.

`c.vertex(i)` returns the position of the `i`th vertex.

`c.all_vertices()` returns a reray of all vertices.

`c.edge_count()` returns the number of edges of `c`.

`c.edge(i)` returns the `i`th edge.

`c.size_of_face(int i)` returns the number of vertices of face `i`.

`c.inner_approximation_d(res, eps)` makes `res` a reray of vectors. The first is the begin point of `c` the last is the end point of `c`. When the consecutive point are connected, an approximation for `c` is obtained which lies nowhere further than `eps` away from `c`. The approximation lies to the center side of the arc.

`c.outer_approximation_d(res, eps)` makes `res` a reray of vectors. The first is the begin point of `c` the last is the end point of `c`. When the consecutive point are connected, an approximation for `c` is obtained which lies nowhere further than `eps` away from `c`. Seen from the center, the approximation lies to the other side of the arc.

`c.inner_approximation_n(res, n)` makes an approximation of `c` of at least `n` vertices, on the inside of the arc.

`c.outer_approximation_n(res, n)` makes an approximation of `c` of at least `n` vertices, on the outside of the arc.

13.10 SP_POLYHEDRON or SP_PHDN

Synopsis

```
class sp_polyhedron: public sp_object{
                    sp_polyhedron()
                    sp_polyhedron(sp_vecreray vtcs, intreray indices)
    int             face_count() const
    sp_polygon      polygon(int i) const
    int             vertex_count() const
    int             facevertex_count() const
    sp_vec          vertex(int i) const
    sp_vecreray     all_vertices() const
    int             edge_count() const
    sp_edge         edge(int i) const
    int             size_of_face(int i) const
    int             index(int i, int j) const
    int             is_ok() const
    int             is_topologically_ok() const
    void            make_right_oriented()
    int             an_edge_on_vertex(int i) const
    int             an_edge_on_face(int i) const
    int             next_edge_on_vertex(int vt, int edge) const
    int             next_edge_on_face(int face, int edge) const
    int             vertex1(int edge) const
    int             vertex2(int edge) const
    int             opposite_vertex(int edge, int vt) const
    int             face1(int edge) const
    int             face2(int edge) const
    int             opposite_face(int edge, int face) const
};
```

```
                sp_arc(sp_plane plane, pl_arc arc)
    sp_vec      center() const
    sp_disc     supporting_disc() const
    sp_plane    supporting_plane() const
    pl_arc      planar_projection() const
    s_coord     radius() const
    s_coord     length() const
    sp_vec      begin_vt() const
    sp_vec      end_vt() const
    void        set_angles(pl_angle a, pl_angle d)
    void        inner_approximation_d(sp_vecreray &res, s_coord eps) const
    void        outer_approximation_d(sp_vecreray &res, s_coord eps) const
    void        inner_approximation_n(sp_vecreray &res, int n) const
    void        outer_approximation_n(sp_vecreray &res, int n) const
};
```

Description

An arc is a part of a circle. Arcs can be represented by the type `sp_arc`.

Constructors

The first constructor leaves `c` undefined. The second constructor takes two parameters. The first parameter is a plane. The second parameter is a planar arc. This planar arc describes `c` relative to the plane that was passed as the first parameter. Here, `plane.e1()` is taken as the x-axis of the plane and `plane.e2()` is taken as the y-axis. Preferably, the reference vertex of the plane should be the center of the arc.

Class Functions

`c.center()` returns the center of the arc. The center of an arc is center of the circle of which the arc is a part.

`c.supporting_disc()` returns the disc of which `c` is part of the boundary.

`c.supporting_plane()` returns the plane in which `c` lies. The reference vertex of this plane is the center of the arc. This is equivalent with taking the supporting plane of the supporting disc.

`c.planar_projection()` returns the planar arc that describes `c` in the plane `c.supporting_plane`.

`c.radius()` returns the radius of the supporting circle.

`c.length()` returns the length of `c`.

`c.begin_vt()` returns the begin point of the arc.

`c.end_vt` returns the endpoint of `c`.

Class Functions

`c.center` returns the center of the sphere.

`c.radius()` returns the radius of the sphere.

13.8 SP_DISC

Synopsis

```
class sp_disc: public sp_object {
                sp_disc()
                sp_disc(sp_plane plane, s_coord radius)
    sp_vec      center() const
    sp_plane    supporting_plane() const
    pl_disc     planar_projection() const
    s_coord     radius() const
    sp_vec      a_vertex() const
};
```

Description

A `sp_disc` is the three dimensional counterpart of a `pl_disc`.

Constructors

The first constructor leaves `c` undefined. The second constructor takes two parameters, a plane and a length. The resulting disc is situated in the plane, with its center at the reference vertex of the plane. The second parameter gives the radius of the disc.

Class Functions

`c.center()` returns the center of `c`.

`c.supporting_plane()` returns the plane in which `c` lies. The reference vertex of this plane coincides with the center of `c`.

`c.planar_projection()` return a two dimensional disc centered in the origin with the same radius. This is `c` projected to its own supporting plane.

`c.radius()` returns the radius of the disc.

`c.a_vertex()` return a vertex on the boundary of `c`.

13.9 SP_ARC

Synopsis

```
class sp_arc: public sp_object {
                sp_arc()
```

Description

Polygons in three dimensions can be represented by the type sp_polygon. A shorthand for the name is sp_pgn.

Constructors

The first constructor makes an empty polygon. The second constructor takes a reray of vectors, the consecutive vertices of the polygon. All vertices should lie in a plane. If not, they will first be projected onto a plane through some of the vertices.

Class Functions

c.size() returns the number of vertices.

c.supporting_plane() returns the plane in which the vertices lie.

c.vertex(i) returns the ith vertex of the polygon.

c.edge(i) returns the ith edge of the polygon, i.e. the edge from vertex i to vertex i+1.

c.all_vertices() returns a reray with all vertices.

c.planar_projection() returns the two dimensional polygon that is the projection of c on the supporting plane. Because of the way the supporting plane is chosen, this polygon is always counterclockwise.

13.7 SP_SPHERE

Synopsis

```
class sp_sphere: public sp_object {
public:
                sp_sphere()
                sp_sphere(sp_vec center, s_coord radius)
    sp_vec      center() const
    s_coord     radius() const
};
```

Description

A sphere can be represented by the type sp_sphere. A sphere is characterized by its center and its radius.

Constructors

The first constructor leaves the sphere undefined. The second constructor takes the center and the radius as parameters. The radius must be nonnegative.

Class Functions

`c.ref_vt()` returns a vertex in the plane.

`c.normal()` returns the normal vector of the plane.

`c.e1()` returns the first axis of the plane. If only the normal was specified for the plane, a random, but fixed, unit vector perpendicular to the normal is chosen.

`c.e2()` returns the second axis of the plane. The normal and first and second axis are perpendicular to each other. The equation `c.normal() == cross(c.e1(), c.e2())` holds.

`c.placeproject(vec)` projects `vec`, taken as a place vector, onto `c`. The result is a two dimensional vector `result`. The three dimensional projection point can be found by taking `c.placevec(result)`.

`c.dirproject(vec)` takes the part of the direction vector `vec` that lies inside plane `c`. The result is a planar vector `result`. The spatial vector can be found by taking `c.dirvec(result)`.

`c.placevec(vec2d)` transforms a two dimensional vector to a three dimensional vector according to the formula `c.ref_vt() + vec2d.x() * c.e1() + vec2d.y() * c.e2()`.

`c.dirvec(vec2d)` transforms a two dimensional vector to a three dimensional vector according to the formula `vec2d.x() * c.e1() + vec2d.y() * c.e2()`.

`c.intersect(line2d, plane)` checks if two planes intersect. The two dimensional line is the intersection line described terms of the coordinates induced by `c`. The three dimensional line has reference point `c.placevec(line2d.ref_vt())` and direction `c.dirvec(line2d.direction())`.

13.6 SP_POLYGON or SP_PGN

Synopsis

```
class sp_polygon: public sp_object {
                sp_polygon()
                sp_polygon(sp_vecreray vl)
    int         size() const
    sp_plane    supporting_plane() const
    sp_vec      vertex(int i) const
    sp_vecreray all_vertices() const
    sp_edge     edge(int i) const
    pl_polygon  planar_projection() const
};
```

Constructors

The first constructor leaves the edge undefined. The second constructor takes two vectors as parameters, the begin point and end point respectively.

Class Functions

`c.begin_vt()` returns the begin point of the edge.

`c.end_vt` returns the end point of the edge.

`c.direction()` returns the direction from begin point to end point.

`c.supporting_line()` returns the line on which `c` lies. The direction of the line and the edge coincide.

`c.length()` returns the Euclidean length of `c`.

13.5 SP_PLANE

Synopsis

```
class sp_plane: public sp_object {
                sp_plane()
                sp_plane(sp_fixedvec vec, sp_unitvec normal)
                sp_plane(sp_fixedvec vec, sp_unitvec uvec1, sp_unitvec uvec2)
    sp_vec      ref_vt() const
    sp_unitvec  normal() const
    sp_unitvec  e1() const
    sp_unitvec  e2() const
    pl_vec      placeproject(sp_fixedvec vec) const
    pl_vec      dirproject(sp_fixedvec vec) const
    sp_vec      placevec(pl_vec vec2d) const
    sp_vec      dirvec(pl_vec vec2d) const
    int         intersect(pl_line &line2d, sp_plane plane) const
};
```

Description

The type `sp_plane` is used to represent planes.

Constructors

The first constructor leaves the plane undefined. The second constructor takes a reference vertex (a vertex in the plane) and a normal as parameters. The third constructor takes a reference vertex and two unit vectors as parameters. The unit vectors should be perpendicular and are taken as the two main axes of the plane.

13.3 SP_RAY

Synopsis

```
class sp_ray: public sp_object {
                sp_ray()
                sp_ray(sp_vec vt, sp_unitvec uvec)
    sp_vec      begin_vt() const
    sp_unitvec  direction() const
};
```

Description

A half line or ray can be described with the type `sp_ray`. The direction of a ray points away from its begin vertex.

Constructors

The first constructor leaves the ray undefined. The second constructor takes the begin point of the ray and its direction as parameters.

Class Functions

`c.begin_vt()` returns the begin vertex of `c`.

`c.direction()` returns the direction of `c`.

`c.supporting_line()` returns the line on which `c` lies. The direction of the supporting line is the same as the direction of `c`.

13.4 SP_EDGE

Synopsis

```
class sp_edge: public sp_object {
                sp_edge()
                sp_edge(const sp_vec v1, const sp_vec v2):
    sp_vec      begin_vt() const
    sp_vec      end_vt() const
    sp_unitvec  direction() const
    sp_line     supporting_line() const
    s_coord     length() const
};
```

Description

A `sp_edge` is the straight line segment between two points in space. An edge is directed from the begin point to the end point.

```
                sp_line(const sp_vec vt, const sp_unitvec uvec)
    sp_vec      ref_vt() const
    sp_unitvec  direction() const
    sp_vec      placevec(s_coord co) const
    sp_vec      dirvec(s_coord co) const
    s_coord     placeproject(sp_fixedvec vec)
    s_coord     dirproject(sp_fixedvec vec)
};
```

Description

Lines in space can be described with the type `sp_line`. Lines are directed.

Constructors

The first constructor leaves `c` undefined. The second constructor takes two vectors as parameters, which are two points on the line. `c` will be directed from the first point to the second one. The third constructor initializes `c` to a line through `vt` in the direction of `uvec`.

Class Functions

`c.ref_vt()` returns a vertex on `c`. The vertex that is returned is random, but guaranteed to be the same for subsequent calls.

`c.direction()` returns the direction of `c`.

`c.placevec(co)` transforms a coordinate to a 3D vector according to `result = c.ref_vt() + co * c.direction()`.

`c.dirvec(co)` transforms a coordinate to a 3D vector which is `co * c.direction()`.

`c.placeproject(vec)` projects a direction vector onto the line. The result is a coordinate on the line. The three dimensional vector can be found by `c.placevec(result)`.

`c.dirproject(vec)` returns the component of `vec` along `c`, i.e., the dot product of `vec` and `c.direction()`. The three dimensional vector can be found by taking `c.dirvec(result)`.

Other Routines

`sp_vec placeproject(sp_fixedvec vec, sp_line line)`

Projects `vec` onto `line`. This does the same as `line.placevec(line.placeproject(vec))`.

`sp_vec dirproject(sp_fixedvec vec, sp_line line)`

This routine returns the component of `vec` along `line`. It does the same as `line.dirvec(line.dirproject(vec))`.

Chapter 13

Spatial Objects

This chapter describes the geometric objects of SpaGeo. All the planar objects –points, lines, rays, edges, polygons, discs and arcs– have a three dimensional counterpart. Besides there are planes, polyhedra and spheres.

13.1 SP_POINT

Synopsis

```
class sp_point: public sp_vec {
            sp_point()
            sp_point(sp_fixedvec vec)
            sp_point(s_coord x, s_coord y, s_coord z)
    sp_vec  vec();
};
```

Description

Points in space are represented by the type `sp_point`. This type inherits from `sp_object`.

Constructors

The first constructor leaves the point undefined. The second constructor initializes `c` with a vector, the third one with the corresponding x-, y- and z-values.

Class Functions

`c.vec()` returns the vector corresponding with `c`.

13.2 SP_LINE

Synopsis

```
class sp_line: public sp_object {
                sp_line()
                sp_line(const sp_vec v1, const sp_vec v2)
```

```
    sp_vec        translation_part() const
    sp_ortho_tfm  operator*(sp_ortho_tfm tfm) const
    sp_ortho_tfm  inverse() const
};
```

Description

The most general angle preserving transformation is represented by the type `sp_ortho_tfm`. This transformation consists of a rotation, a uniform scale and a translation, in that order.

Constructors

The first constructor takes three parameters. The first two describe the rotation, the third gives the scaling factor and the last one is the translation vector.

The other constructors leave some or all parts out. In this case a default value is chosen for those parts; i.e. a zero rotation, a scaling with a factor 1 and a zero translation.

Class Functions

`c.rotation_part()` returns the `sp_rotation` describing the rotation part.

`c.scaling_part()` returns the scaling factor.

`c.translation_part()` returns the translation vector.

`c * tfm` returns the composition of `c` and `tfm`.

`c.inverse()` returns the inverse transformation of `c`.

Constructors

There are quite a few constructors. This is due to the fact that a rotation part may be given either as a axis and angle pair or as a `sp_rotation`, and because the rotation part and translation part may be present or not. If a part is not present, it will be initialized to zero (angle or translation).

Only the first constructor is described here. It takes three parameters. The first two parameters are an axis and an angle, like the parameters of a `sp_rotation`. The third parameter is a translation vector. When this transformation is applied to an object, the object is first rotated around the `axis` over `angle` radians and the translated over `vec`.

Class Functions

`c.translation_part()` returns the translation part of `c`.

`c.rotation_part()` returns the rotation part of `c`. The axis and angle can be obtained from this `sp_rotation`.

`c * tfm` returns the composition of `c` and `tfm`. This means that `(c*tfm)*x` is equal to `c*(tfm*x)` for `x` a length or vector.

`c.inverse()` returns the inverse transformation of `c`.

Examples

Here it is shown how a rotation around a line that does not go through the origin can be computed. This rotation can easily be performed in three steps. First translate so that the reference vertex of the line goes to the origin. Then rotate around the line. Finally translate the reference vertex back to its original position.

```
sp_rotra line_rotation(sp_line line, pl_angle angle)
{
  return sp_rotra(line.ref_vt()) *
      sp_rotra(line.direction(), angle) * sp_rotra(-line.ref_vt());
}
```

12.5 SP_ORTHO_TFM

Synopsis

```
class sp_ortho_tfm: public sp_transform {
                  sp_ortho_tfm(sp_unitvec axis, pl_angle angle,
                                    s_coord co, sp_fixedvec vec)
                  sp_ortho_tfm(sp_unitvec axis, pl_angle ang)
                  sp_ortho_tfm(s_coord co)
                  sp_ortho_tfm(sp_fixedvec vec)
                  sp_ortho_tfm()
    sp_rotation   rotation_part() const
    s_coord       scale_part() const
```

Class Functions

`c.axis_and_angle(vec)` returns the rotation angle as the result and makes `vec` equal to the rotation axis. The angle and axis may be different from the originally specified angle and axis.

`c * tfm` returns the rotation that results from composing `c` and `tfm`. The result is such that `result*x` is equal to `c*( tfm*x)`, for any vector, direction or length `x`.

`c *= tfm` is a shorthand for `c = c * tfm`.

`c.inverse()` returns the inverse rotation of `c`.

Examples

A well known way to describe rotations is by giving three parameters that are rotation angles around coordinate axes. Here is a routine that computes the resulting rotation when rotating first around the x-axis, then around the y-axis and then around the z-axis.

```
sp_rotation rot_xyz(float x_angle, float y_angle, float z_angle)
{
   return sp_rotation(sp_zdir, z_angle) * sp_rotation(sp_ydir, y_angle) *
          sp_rotation(sp_xdir, x_angle);
}
```

12.4 SP_ROTRA

Synopsis

```
class sp_rotra: public sp_unit_tfm {
                sp_rotra(sp_unitvec axis, pl_angle angle, sp_fixedvec vec)
                sp_rotra(sp_rotation r, sp_fixedvec vec)
                sp_rotra(sp_unitvec axis, pl_angle angle)
                sp_rotra(sp_rotation r)
                sp_rotra(sp_fixedvec v)
                sp_rotra()
    sp_vec      translation_part() const
    sp_rotation rotation_part() const
    sp_rotra    operator*(sp_rotra tfm) const
    sp_rotra    inverse() const
};
```

Description

A transformation that leaves distances invariant and does not mirror objects, can always be described by a rotation around the origin followed by a translation. The type `sp_rotra` is used to represent those kind of transformations. Of course, the rotation part or the translation part can be zero, in which case a `sp_rotra` has the same effect as the corresponding `sp_translation` or `sp_rotation` respectively.

```
                    sp_translation(sp_fixedvec vec)
                    sp_translation()
    sp_vec          vec() const
    sp_translation  operator*(sp_translation tfm) const
    sp_translation  inverse() const
};
```

Description

Translations over a vector can be represented by the type `pl_translation`. When applied to an object, this vector is added to all place vectors of the object. A translation does not alter lengths and direction vectors.

Constructors

The first constructor takes the translation vector as parameter. The second constructor initializes the translation vector to `(0,0,0)`.

Class Functions

`c.vec()` returns the translation vector.

12.3 SP_ROTATION

Synopsis

```
class sp_rotation: public sp_unit_tfm {
                sp_rotation(sp_unitvec axis, pl_angle angle)
                sp_rotation()
    pl_angle    axis_and_angle(sp_unitvec &vec) const
    sp_rotation operator*(sp_rotation tfm) const
    void        operator*=(sp_rotation tfm)
    sp_rotation inverse() const
};
```

Description

Rotations can be represented by the type `sp_rotation`. Rotations in space do not only need an angle, but also a rotation axis. This axis always goes through the origin.

Constructors

The first constructor takes the direction of the rotation axis and the rotation angle as parameters. A rotation around the positive z-axis over $\pi/2$ will transform the x-axis to the y-axis and the y-axis to the negative x-axis. The second constructor initializes the angle to 0.

Chapter 12

Transformations

This chapter describes all the different spatial transformations. All the transformations in this chapter are derived from the class sp_transform. The common interface is not repeated here.

12.1 SP_UNISCALE

Synopsis

```
class sp_uniscale: public sp_transform {
                    sp_uniscale(s_coord f)
                    sp_uniscale()
    s_coord         factor() const
};
```

Description

The type sp_uniscale describes uniform scaling transformations. Every point is multiplied by a factor. Both positive and negative scaling factors are possible. A negative factor mirrors an object in the origin. A zero factor is not recommended, as it will reduce all objects to a point.

Constructors

The first constructor takes the scaling factor is given as a parameter. The second constructor sets the scaling factor to 1.

Class Functions

c.factor() returns the scaling factor of c.

12.2 SP_TRANSLATION

Synopsis

```
class sp_translation: public sp_unit_tfm {
```

Class Functions

`c.bbox()` returns the bounding box of `c`, that is, the smallest box containing `c`. If `c` extends infinitely in a direction, the largest possible value in that direction is chosen.

`c.transform(tfm)` transforms `c` according to `tfm`.

`c.rotate(axis, angle)` transforms `c` by rotating it `angle` radians around `axis`.

`c.translate(vec)` transforms `c` by translating it over a vector `vec`.

`c.scale(co)` transforms `c` by scaling it by a factor `co`.

`c.type()` returns a value of type `sp_object_type`, which indicates the type of object.

Description

Three dimensional objects can be uniformly scaled, translated and rotated by transformations. An object of the same type will result.

All transformations are derived from the the type `sp_transform`, either directly or through its descendant `sp_unit_tfm`, the base class for all transformations that preserve distances. No objects of type `sp_transform` or `sp_unit_tfm` exist. Those types only serve as base classes for other transformations.

A transformation can be characterized by describing how it transforms place vectors, direction vectors and lengths. Actually, the first part would be enough. A direction vector is the difference of two place vectors, so the transformation of direction vectors follows from this. In the same way, the transformation of lengths follows from the transformation of vectors.

Class Functions

`c * vec` returns the transformed place vector.

`c * uvec` returns the transformed direction vector. Written in terms of place vector transformations, this is equal to `normalized(c*pl_vec(uvec) - c*pl_vec(0,0,0))`.

`c * co` returns the transformed length. Written in terms of place vector transformations, this is equal to `(c*pl_vec(co,0,0) - c*pl_vec(0,0,0)).length()`.

11.5 Spatial Objects

Synopsis

```
class sp_object {
    sp_box          bbox() const
    void            transform(sp_transform tfm)
    void            rotate(sp_unitvec axis, pl_angle angle)
    void            translate(sp_fixedvec vec)
    void            scale(s_coord co)
    sp_object_type  type()
};

enum sp_object_type {SP_POINT, SP_LINE, SP_RAY, SP_EDGE, SP_PLANE,
                    SP_POLYGON, SP_POLYHEDRON, SP_SPHERE, SP_CIRCLE, SP_ARC};
```

Description

All three dimensional objects in the sequel share a common interface. They all have a bounding box, they can all be transformed and they can all answer what type of object they are. This shared functionality is defined in the base class `sp_object`.

`c.ymax() c.ymin()` return the maximum and minimum y value.

`c.zmax() c.zmin()` return the maximum and minimum z value.

`c.make_empty()` makes `c` an empty box.

`c.add_point(vec)` enlarges `c`, if necessary, so that `c` contains point `vec`.

`c.add_first_point(vec)` makes `c` a box containing only point `vec`.

`c.add_non_first_point(vec)` is only defined if `c` is defined and not empty. Then it is equivalent with `c.add_point(vec)`.

`c.translate(vec)` translates `c` over vector `vec`.

`c.scale(fac)` scales `c` with fac, i.e., multiplies all coordinates with `fac`.

`c += bb` makes `c` the union box of `c` and `bb`. The union box of two boxes is defined here as the box just big enough to contain both boxes.

Other Routines

```
sp_box   intersect(sp_box bb1, sp_box bb2)
```

computes the intersection of `bb1` and `bb2`.

```
sp_box   operator+(sp_box bb1, sp_box bb2)
```

computes the union box of `bb1` and `bb2`. The union box is defined above.

```
int lies_inside(sp_fixedvec vec, sp_box bb);
```

returns whether `vec` lies inside `bb`.

```
int lies_inside(sp_box bb1, sp_box bb2);
```

returns whether `bb1` lies totally inside `bb2`.

11.4 Transformations

Synopsis

```
class sp_transform{
    sp_vec      operator*(sp_vec vec) const
    sp_unitvec  operator*(sp_unitvec uvec) const
    s_coord     operator*(s_coord co) const
};

class sp_unit_tfm : public sp_transform{ };
```

Examples

Here is a small example that shows how an orthogonal base can be constructed when one unit vector (uvec1) is given.

```
uvec2 = uvec1.perpendicular();
uvec3 = normalized(cross(uvec1, uvec2));
```

The first line makes uvec2 a unit vector perpendicular to uvec1. In the next line the cross product of uvec1 and uvec2 is taken. The cross product of two vectors is always perpendicular to those vectors. Because the cross product returns a sp_vec, not a sp_unitvec, the function normalized is needed to turn the result into a unit vector.

11.3 SP_BOX

Synopsis

```
class sp_box {
                sp_box()
                sp_box(sp_fixedvec vec)
    int         is_empty() const
    s_coord     xmax() const
    s_coord     xmin() const
    s_coord     ymax() const
    s_coord     ymin() const
    s_coord     zmax() const
    s_coord     zmin() const
    void        make_empty()
    void        add_point(sp_fixedvec vec)
    void        add_first_point(sp_fixedvec vec)
    void        add_non_first_point(sp_fixedvec vec)
    void        translate(sp_fixedvec vec)
    void        scale(s_coord fac)
    void        operator+=(sp_box bb)
};
```

Description

The type sp_box is the three dimensional counterpart of pl_box. The edges of the box are parallel to one of the three coordinate axes.

Constructors

The first constructor leaves c undefined. The second constructor initializes c to a degenerated box, containing only the point vec.

Class Functions

c.xmax() c.xmin() return the maximum and minimum x value.

`c.normalize()` multiplies `c` with a factor so that `c` will be of unit length afterwards.

`c.perpendicular()` returns a unit vector that is perpendicular to `c`.

`-c` returns the inverse of `c`.

Other Routines

The following functions define a lexicographic ordering on spatial vectors. First the x value is taken into account, then the y value and finally –if both x and y are the same– the z value.

```
int operator< (sp_fixedvec, sp_fixedvec)
int operator> (sp_fixedvec, sp_fixedvec)
int operator<= (sp_fixedvec, sp_fixedvec)
int operator>= (sp_fixedvec, sp_fixedvec)
```

```
sp_vec operator-(sp_fixedvec vec)
```

`-vec` returns the inverse of `vec`.

```
sp_vec operator*(s_coord l, sp_fixedvec vec)
```

`l*vec` returns `vec` multiplied by `l`.

```
sp_vec operator+(sp_fixedvec vec1, sp_fixedvec vec2)
```

`vec1 + vec2` returns the sum vector of the two vectors.

```
sp_vec operator-(sp_fixedvec vec1, sp_fixedvec vec2)
```

`vec1 - vec2` returns the difference of the two vectors `vec1` and `vec2`.

```
s_coord dot(sp_fixedvec vec1, sp_fixedvec vec2)
```

returns the dot product of `vec1` and `vec2`.

```
sp_vec cross(sp_fixedvec vec1, sp_fixedvec vec2)
```

returns the cross product of `vec1` and `vec2`.

There are three predefined unit vectors:

```
const sp_unitvec sp_xdir, sp_ydir, sp_zdir
```

that point along the main coordinate axes.

```
s_coord factorize(sp_unitvec &result, sp_fixedvec vec)
```

factorizes `vec` in its length (the return value) and a unit vector. If `vec` had zero length, the unit vector is randomly chosen, otherwise `result` will point in the same direction as `vec`.

```
sp_unitvec normalized(sp_fixedvec vec)
```

returns a unit vector pointing in the same direction as `vec`. Again, a random unit vector is chosen if `vec` is the zero vector.

```
sp_vec perpendicular_component(sp_fixedvec vec, sp_unitvec uvec)
```

gives the component of `vec` which is perpendicular to `uvec`.

```
    void        operator*=(s_coord);
    void        operator+=(sp_fixedvec vec);
    void        operator-=(sp_fixedvec vec);
    void        operator=(sp_fixedvec vec)
};

class sp_unitvec: public sp_fixedvec {
                    sp_unitvec()
    sp_unitvec      perpendicular() const
    sp_unitvec      operator-() const
    void            operator=(const sp_unitvec &uvec)
};

const sp_unitvec sp_xdir, sp_ydir, sp_zdir;
```

Description

There are three types for defining three dimensional vectors. The base vector class is `sp_fixedvec`. This type represents vectors with x-, y- and z-coordinates. The contents of a `sp_fixedvec` can not be altered. There is even no assignment operator defined for `sp_fixedvec`s.

Derived from `sp_fixedvec` are `sp_vec` and `sp_unitvec`. The type `sp_vec` supplies routines to alter its coordinates. It is the type normally used for vectors. Unit vectors can be represented by the type `sp_unitvec`.

Routines normally take `sp_fixedvec`s as parameters (so all types of vectors can be passed as a parameter), but return `sp_vec`s.

Constructors

The constructors that take three `s_coord`s initialize the x-, y- and z-value accordingly. The constructors without parameters leave the vector in an unspecified, but valid, state. Especially, the length of the vector will be 1 for a `sp_unitvec`.

Class Functions

`c.x() c.y() c.z()` return the x-, y- and z-value respectively.

`c.length()` returns the Euclidean length of the vector.

`c.sqlength()` returns the square of the length.

`c.setx(x) c.sety(y) c.setz(z)` set a value.

`c *= l` multiplies `c` with a factor `l`. If `l` is negative, `c` will point in the opposite direction afterwards.

`c += vec` adds `vec` to `c`.

`c -= vec` subtracts `vec` from `c`.

Chapter 11

The Basic Types of SpaGeo

11.1 Introduction

SpaGeo is the three dimensional sister of PLaGeo. Many of the objects that exist in PlaGeo, like vectors, edges, arcs and polygons, exist in SpaGeo as well. The difference is that all coordinates are three dimensional. There are also new objects: planes, polyhedra and spheres.

All the names of SpaGeo objects have the prefix 'sp_'. So a `sp_edge` is an edge in space, whereas a `pl_edge` is an edge in the plane.

The SpaGeo documentation assumes that the reader is familiar with PlaGeo. Because a lot of things carry over immediately from PlaGeo, the description of the objects is rather concise, with little examples. This part therefore has more the flavor of a reference manual.

11.2 Vectors

Synopsis

```
class sp_fixedvec {
                sp_fixedvec()
                sp_fixedvec(s_coord x, s_coord y, s_coord z)
    s_coord     length() const
    s_coord     sqlength() const
    s_coord     x() const
    s_coord     y() const
    s_coord     z() const
};

class sp_vec: public sp_fixedvec {
                sp_vec()
                sp_vec(sp_fixedvec vec)
                sp_vec(s_coord x, s_coord y, s_coord z)
    void        set_x(s_coord x)
    void        set_y(s_coord y)
    void        set_z(s_coord z)
    void        normalize()
```

Part III

SpaGeo

bounded clearance

no. of vertices	3	10	20	50	100
separable, first	0.014	0.039	0.064	0.140	0.370
separable, next	0.006	0.006	0.006	0.007	0.008
non-separable, first	0.37	1.29	3.57	11.630	34.870
non-separable, next	0.21	0.76	2.35	9.090	28.970

clearance

no. of vertices	3	10	20	50	100
separable, first	0.24	0.54	1.88	7.83	29.26
separable, next	0.23	0.52	1.86	7.67	28.95
non-separable, first	0.53	1.53	4.14	11.87	39.53
non-separable, next	0.35	1.00	3.05	9.42	34.78

Figure 10.2: Timing results of clearance in msec.

do-overlap

no. of vertices	3	10	20	50	100
separable, first	0.01	0.03	0.05	0.11	0.27
separable, next	0.003	0.003	0.003	0.003	0.005
non-separable, first	0.22	0.69	1.52	3.78	8.01
non-separable, next	0.06	0.24	0.61	1.54	3.57

do-cross

no. of vertices	3	10	20	50	100
separable, first	0.017	0.03	0.05	0.13	0.24
separable, next	0.003	0.003	0.004	0.004	0.005
non-separable, first	0.28	0.75	1.57	3.95	8.00
non-separable, next	0.06	0.26	0.66	1.69	3.78

Figure 10.1: Timing results of intersections in msec.

10.2 Clearance between Polygons

The routines to compute the clearance are not very clever: they basically compute the clearance between every pair of edges, giving an algorithm with a running time proportional to the product of the number of sides of the polygons.

The clearance was tested both with the normal clearance function as with the one taking an extra parameter, the bounded clearance. The bound was taken so small that the polygons were further apart than this bound.

As can be seen from figure 10.2, the use of a bound can be very advantageous when the bounding boxes of the polygons are far apart. When the bounding boxes are not, though the polygons themselves are, the effect is much less. The reason for the difference between the first and next computations is again caching.

Chapter 10

Performance

In this chapter we give some indications about the performance of some of the plageo-routines. The tests were done on an Silicon Graphics Indigo with an R3000 processor running at 33 MHZ. This machine is rated on the SpecMarks benchmark with 24.2 SPECfp92 and 22.4 SPECint92.

10.1 Intersections of Polygons

Because of the way intersections between two polygons are dealt with, there are several cases that should be considered separately.

First, it is important whether the two polygons are separable by a vertical or horizontal line. If this is the case, a bounding box test is sufficient and the intersection test is very fast, regardless of the size of the polygons.

Second, it is important if this is the first intersection test done with a polygon. This is because of caching. The first time the bounding box of a polygon must be computed and, if the polygons are non-separable, the vertices are sorted. The results are stored with the polygon, so that subsequent intersections need not do this again.

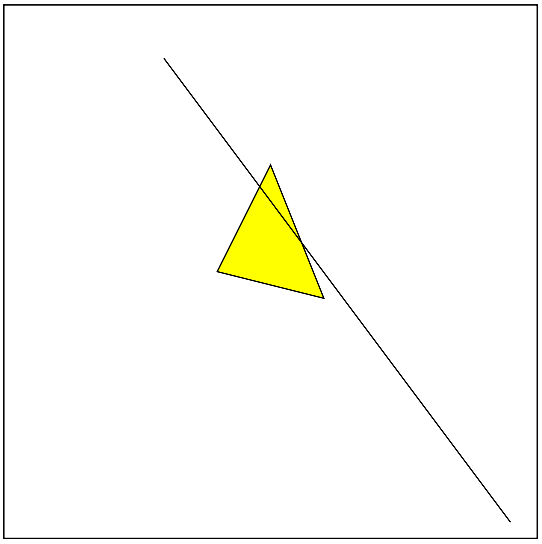
Third, the number of vertices is important. The more vertices, the more time an intersection normally takes.

In figure 10.1 the timing results of do_overlap and do_cross are shown. The times that are reported are the number of milliseconds needed for one intersection check of a pair of polygons, each with n vertices. Of the two procedures, do_overlap is about ten percent faster on the average.

The tests were done with two non-intersecting polygons. Intersection testing of intersecting polygons takes time comparable to non-separable polygons. In fact, it will be a bit faster, because the testing can stop as soon as an intersection is found.

Note that especially for big polygons it makes a big difference if a bounding box test suffices. The answer to the question 'How many pairs of polygons of 100 vertices each can be checked for intersections' can be 125 or 200000, depending on which case is appropriate. To make an estimate how much time intersection checking will take in a given case, this must be taken into consideration.

```
        }
        win.pageready();
        for (i=0; i<edges.size(); i++)
            win.draw(edges[i]);
    }
}
```



Interactive rendering

In the example below the user is asked to draw a number of edges with the mouse. An edge is started with a press of the left mouse button and ends when the button is released. If the endpoint of the edge is placed outside the window, the edge is not taken into account. The program ends when the begin point of an edge is placed outside the window.

```
#include <plageo.h>

main()
{
    pl_glwindow win(0, 0, 400, 400);
    win.set_viewingarea(-500, -500, 1000);
    win.set_directshow(1);
    pl_box visible_area;
    visible_area.add_first_point(pl_vertex(-500, -500));
    visible_area.add_point(pl_vertex(500, 500));
    pl_vertex vt1, vt2;
    int i;
    pl_edgereray edges;

    while (1) {
        while (!win.get_mousestatus(1)) ;   // wait for leftmouse press
        vt1 = win.get_mouseposition();
        if (!lies_inside(vt1, visible_area))
            break;
        win.draw(vt1);
        while (win.get_mousestatus(1)) ;   // wait for leftmouse release
        vt2 = win.get_mouseposition();
        if (lies_inside(vt2, visible_area)) {
            edges.append(pl_edge(vt1, vt2));
```

```
#include <unistd.h>
#include <plageo.h>

float vertexcoords[] = {400, 500, 600, 450, 500, 700};

void draw_it(pl_drawarea &area)
{
    int i, size;
    pl_vertex vt;
    size = sizeof(vertexcoords)/sizeof(vertexcoords[0]);
    pl_vertexreray vrr;
    for (i=0; i<size-1; i+=2) {
      vt.setx(vertexcoords[i]);
      vt.sety(vertexcoords[i+1]);
      vrr.append(vt);
    }
    pl_pgn pgn(vrr);
    pl_edge edge(pl_vertex(300, 900), pl_vertex(950, 30));
    area.draw(pgn);
    area.draw(edge);
    area.pageready();
}

main()
{
    pl_psfile ps_area(cout);
    draw_it(ps_area);
    pl_glwindow gl_area;
    draw_it(gl_area);
    sleep(10);
}
```

The produced file was included with the epsf style in this LaTeXdocument, leading to the following picture:

```
};

extern void pl_use_colormap(int col);

extern void pl_use_rgbcolor(int col);

extern void pl_use_autocolor(int col);
```

Description

Like `pl_glwindow`, `pl_glsubwin` can only be used on machines having the Graphics Library of Silicon Graphics. The type `pl_glsubwin` is a faster and more versatile renderarea than the type `pl_glwindow`, but some knowledge of the Graphics Library is necessary. It can be used to draw PlaGeo pictures in a (part of) an existing GL window.

The `pl_glsubwin` renderer is not tied to a specific window. It is the responsibility of the user that the right window is set. The GL window may be in colormap mode or RGB mode, it may be single or double buffered. The `pl_glsubwin` only does some drawing, it does not swap buffers or respond to events (like redraw events). This is the responsibility of the user.

Constructors

The first constructor gets four parameters. Those indicate in which part of the window PlaGeo will render. They give the bottom left pixel (in window coordinates) and the size of the area. This part of the window is called the render area.

The second constructor leaves the render area undefined. It should be set later on with the routine `set_renderarea`.

Class Functions

`c.set_renderarea(x, y, dx, dy)` sets the render area. The parameters are the same as that of the first constructor.

`c.set_colorfunc` The window in which the `pl_glsubwin` renders can be either in colormap or in RGB mode. Normally, this is checked during every render action and the appropriate routine is used to set a color. In order to tell `c` that the window is in colormap mode, call `c.set_colorfunc(pl_use_colormap)`. The other two routines (`pl_use_rgbcolor` and `pl_use_autocolor`) tell `c` that the window is in RGB mode or to use the default behavior respectively.

9.5 Rendering Examples

Rendering in two different renderers

Below an example is given where a simple scene, consisting of a triangle and an edge, is rendered both to a postscript file and to a window. Those drawing areas are created in the main function. The drawing itself is done in another routine. As this routine uses only the functions of the common interface, it can be used for both renderers.

The second constructor takes only the size of the window. The user must place the window somewhere on the screen. In fact, the size parameters here only give the ratio between the x- and y-size.

The last constructor makes a square window. It acts the same as the previous constructor with both parameters set to the same value.

Class Functions

Most class functions are inherited from the base class `pl_drawarea`. Here only the ones that are particular for a GL window are described.

`c.set_directshow(b)` This routine sets a switch to decide whether objects should be rendered immediately to the window (b equal to 1) or that this should be delayed until a call of `pageready` (b equal to 0).[1] The second type is the default.

`c.hide()` hides the window. Drawing an object makes the window reappear. The window is hidden automatically when the renderer is destructed.

`c.get_mouseposition()` gives a vertex that describes the point where the mouse is currently pointing. The mouse can be outside the window, in which case the returned vertex will lie outside the viewing area.

`c.get_mousestatus(button)` returns the status (pressed is 1, not pressed is 0) of the requested mousebutton. The parameter `button` should be either 1 (left button), 2 (middle button) or 3 (right button).

`c.identifier()` gives the GL window identifier of the window that is opened, or −1 if no window is opened. Normally, this identifier is not needed, but it can be handy for doing special GL stuff that this simple interface doesn't support.

Remarks

The GL window uses the color parameters as an index in the GL colormap. Users familiar with the GL can use this fact to create their own colors by changing an entry of the colormap. In that case, the colors do not correspond with the colors used by other renderers any more.

When compiling a program that uses a GL window, the gl library should be linked. Therefore put the switch `-fm_s -lgl_s` just after `-lplageo` on the compilation line.

9.4 PL_GLSUBWIN

Synopsis

```
class pl_glsubwin: public pl_drawarea {
            pl_glsubwin(int x, int y, int dx, int dy)
            pl_glsubwin() ;
    void    set_renderarea(int x, int y, int dx, int dy)
    void    set_colorfunc(void (*cf)(int) )
```

[1] For people familiar with the GL: drawn in single buffer mode or double buffer mode.

Description

One of the renderers that is available renders objects to a file containing PostScript commands. The class `pl_psfile` that describes this renderer, inherits from the class `pl_drawarea`, so it doesn't need to define its own interface for drawing objects.

Constructors

The first constructor takes five arguments. The first is the output stream to which the output is written. This can be `cout` or a stream opened by the user for output (an ofstream). See chapter 4 for more information on streams.

The other parameters describe the shape of the image that is drawn. The first two give the bottom left point of the image, the last two give the size of the image. Do not confuse these parameters with the viewing area: the viewing area is the part of the world of geometrical objects that is visible, those parameters describe the part of the page that is used. Normally, the xsize and ysize of the viewingarea and the page should have the same ratio, otherwise the image is distorted.

The second constructor takes only the stream as parameter. The image takes a default, square, size.

Remarks

Although the result is a color PostScript file, do not expect to see those colors on a black and white printer. The colors will be turned into different shades of grey on such a printer.

9.3 GL renderer

Synopsis

```
class pl_glwindow: public pl_drawarea {
                pl_glwindow(int x,int y,int dx,int dy)
                pl_glwindow(int dx,int dy)
                pl_glwindow()
    int         set_directshow(int b)
    void        hide();
    pl_vertex   get_mouseposition()
    int         get_mousestatus(int button)
    long        identifier() const
}
```

Description

On machines that support the Graphics Library of Silicon Graphics, there is the renderer `pl_glwindow`. This renderer draws objects in a GL window on the screen.

Constructors

The first constructor creates a window starting at pixel `x`, `y`. The left bottom of the screen is taken as origin. The size of the window is `dx` by `dy` pixels.

`c.pageready()` is a procedure that should be called when a page is ready. It starts a new, clean page.

`c.set_textsize(size)` sets the size of text characters. The size is specified as the permillage of the size of the viewing area, just like it is done with the vertex size.

Other Routines

All colors can be described by three components: red, green and blue. The function:

```
void pl_color_components(int &red, int &green, int &blue, int col)
```

returns the values of the components of color `col`. Those values range from 0 to 255.

The user of a renderer will usually not need this function. It is needed for implementing a renderer.

Remarks

Although there do not exist objects of type `pl_drawarea`, pointers or references to this type may exist and may indeed be quite useful. An example of this is given at the end of this chapter, where there is a routine with a reference to a drawarea as parameter, so any derived type can be given as a parameter.

The different parameters of a drawarea have default values as described below:

- the viewing area extends from 0 to 1000 in both x- and y-direction.
- the background color is 7 (white).
- the line color is 0 (black).
- the area color is 3 (yellow).
- both outline and inside of polygons are drawn.
- vertices are rendered as a cross.
- the vertex size is 10.
- the text size is 20.

Every renderer tries to map the color numbers to the same colors. The first 8 colors, from index 0 to 7, are black, red, green, yellow, blue, magenta, cyan and white.

9.2 Postscript Renderer

Synopsis

```
class pl_psfile: public pl_drawarea {
    pl_psfile(ostream &os, int xoffset, int yoffset, int xsize, int ysize)
    pl_psfile(ostream &os)
};
```

Description

The type `pl_drawarea` serves as a base type for all actual drawing classes. It describes the interface that they all have, but is not itself a type of renderer. Therefore, no objects of this type are allowed to exist (as there is no constructor, this would be impossible anyway). An actual renderer must supply a rectangular space (a page) where objects can be rendered.

There are a number of colors available. The precise number is available in the constant `PL_COLORCOUNT`. Colors are referenced by their number, which can range between 0 and `PL_COLORCOUNT`—1.

Class Functions

`c.clear()` clears the page, that is, paints it in the background color.

`c.draw(...)` draws an object on the page.

`c.draw_text(place,string)` draws the text of `string` on the page. It starts writing at the specified `place`.

`c.set_viewingarea(x,y,xsize,ysize)` describes the part of the world that is rendered on the page. The `x` and `y` give the bottom left coordinate that can be seen. The `xsize` and `ysize` give the size of the rectangle that is viewed.

`c.set_viewingarea(x,y,size)` does the same as the previous routine, except that the size in both directions is the same (a square part is viewed).

`c.get_viewingarea(x,y,xsize,ysize)` returns the parameters that describe the viewing area.

`c.set_backcol(col)` sets the color that is used for painting the background.

`c.set_textcol(col)` sets the color that is used for drawing text.

`c.set_linecol(col)` sets the color that is used for one and zero dimensional parts. This includes vertices, edges and the outline of polygons and discs.

`c.set_areacol(col)` sets the color that is used for areas, like the interior of polygons or discs.

`c.set_areafill(how)` sets a switch to decide how areas (polygons and discs) are rendered. If the value is 1, only the inside is drawn. If the value is 2, only an outline is drawn. If the value is 3, both inside and outside are drawn. For other values, nothing is drawn. The return value is the previous value of the switch.

`c.set_vertexshape(int shape)` sets a switch to decide how vertices are drawn. Vertices can be drawn as a dot(0), a cross(1), an open square(2) or a filled box(3).

`c.set_vertexsize(size)` sets the radius of the shape that is used to render a vertex. The size is specified as the permillage of the size of the viewing area. So, if the viewing size is 600 and the vertex size is 10, a vertex with coordinates (100,200) is rendered within a box with extremal points (94,194) and (106,206). If the viewing size is not the same in both directions, the maximum is taken.

Chapter 9

Rendering of Objects

Often it is required to draw geometric objects, for example in a window on the screen or on a laser printer. This chapter describes the support that PlaGeo gives in this respect. First the general framework for rendering geometrical objects is described. Then two special cases are described: rendering to a postscript file and rendering to a GL-window.

9.1 pl_drawarea

Synopsis

```
class pl_drawarea {
    void    clear()
    void    draw(const pl_fixedvec &vec)
    void    draw(const pl_box &box)
    void    draw(const pl_object &obj);
    void    draw_text(const pl_vertex &place, const char *string)
    void    set_viewingarea(float x, float y, float xsize, float ysize)
    void    set_viewingarea(float x, float y, float size);
    void    get_viewingarea(float &x, float &y, float &xsize, float &ysize);
    int     set_backcol(int col)
    int     set_textcol(int col)
    int     set_linecol(int col)
    int     set_areacol(int col)
    int     set_areafill(int how)
    enum areafills {NOFILL=0, FILL=1, OUTLINE=2, FILLOUT=3};
    int     set_vertexshape(int shape)
    enum vertexshapes {DOT=0, CROSS=1, SQUARE=2, BOX=3};
    float   set_vertexsize(float size)
    float   set_textsize(float size)
    void    pageready()
}

const int PL_COLORCOUNT
```

```
s_coord clearance(const pl_line &line, const pl_ray &ray)
s_coord clearance(const pl_line &line, const pl_ray &ray, s_coord bound)
```

They both compute the clearance between, in this case, a line and a ray. The second routine takes an extra parameter, the bound. If the clearance is bigger than this bound, the bound will be returned instead of the clearance. This gives the possibility of optimizations: it is often easier to decide that an object lies far away than to compute the exact distance.

There are also some routines especially for optimization purposes. They will normally not be used by the user of PlaGeo, but may be handy if some new object types are created and efficient routines need to be written for them. The following routines make a quick and dirty test if two objects lie far apart:

```
int certain_clearance(const pl_box &bb1, const pl_box &bb2, s_coord bound)
int certain_clearance(const pl_fixedvec &vt, const pl_box &b, s_coord bound)
```

If those routines return true, the two bounding boxes or the bounding box and the vertex lie further than `bound` apart. If they return false, not much can be said: the objects might be closer together than bound or further apart than bound, though not very much further. What, in fact, is done, is that for each coordinate the distance in that direction is compared with `bound`. This is cheaper than comparing Pythagorean distances and catches a lot of cases when things are far away.

is made to check if the lines are exactly parallel. On the other hand, a zero dimensional object will never cross another object, because that would mean that a point lies exactly on a line or another point.

Reporting Coincidences

Sometimes it is not enough to know that two objects intersect; the intersection area is needed as well. PlaGeo provides an answer for those kind of questions when both objects are one dimensional. The objects that are supported are:

- pl_edge
- pl_ray
- pl_line
- pl_arc
- pl_disc (the outline of the disc, so the circle)

When both objects are straight, there can only be one intersection point (witness). Those routines have the form:

```
int intersect(pl_vec &wn, const pl_ray &ray, const pl_edge &edge)
```

The result is true if the two objects intersect and, if they do, the witness is returned in the first argument.

When one or both objects are curved (a disc or an arc), there may be two witnesses. Those routines look like:

```
int intersect(pl_vec &wn1, pl_vec &wn2,
              const pl_edge &edge, const pl_disc &disc)
```

In this case the result can be 0 (no intersections), 1 (only the first witness is valid), 2 (the second witness is valid) or 3 (both witnesses are valid).

Remarks

Parallel lines or edges are a problem. According to PlaGeo two parallel lines do never intersect, not even a line with itself! The reason for this is that with inexact computations it cannot be decided if two lines coincide exactly.

The outline of boxes and polygons are not supported for those routines yet. In a later version of PlaGeo this will be provided.

8.2 Distances

An important concept is the distance or clearance between two objects. The clearance is defined as the length of the shortest line segment that joins a point of one object to a point of the other. There are routines to compute the clearance for all pairs of objects. The only exception is the absence of a clearance routine for lines.

There are two ways to ask for the clearance, having the following form:

Chapter 8

High Level Routines

8.1 Intersections

PlaGeo has a number of routines to detect if two geometric objects intersect. There is one group of routines that answer the decision question only (is there an intersection or not) and another group that, in addition, gives information where this intersection takes place. The first group is complete, in the sense that for every pair of objects such a question can be asked. The second group is not complete. Sometimes the answer would be too trivial (for example, the intersection area of a point and a polygon is either the point or empty), sometimes the answer would be too complicated (for example the intersection area for a polygon and a disc).

Decision Routines

In order to describe the various routines that answer if two geometrical objects intersect, it is convenient to divide the objects in three groups:

zerodim	onedim	twodim
pl_point	pl_edge	pl_box
pl_fixedvec	pl_ray	pl_disc
	pl_line	pl_polygon
	pl_arc	

There are two different kinds of intersection query routines:

```
int do_overlap(const pl_object &obj1, const pl_object &obj2)
int do_cross(const pl_object &obj1, const pl_object &obj2)
```

Both routines can not only take `pl_objects`, but also objects of type `pl_fixedvec` (giving the same result as `pl_point`) and `pl_box`.

The first routine, `do_overlap`, checks if the two objects have a point in common. Two dimensional objects are considered to be filled, so if an object is totally contained in another object, `do_overlap` will return true (1). This is the difference with `do_cross`, which only takes into account the boundary of two dimensional objects.

Although for completeness sake every pair of objects can be passed as parameter, some combinations do not make sense, because of the inexact computations of PlaGeo. For example, when asked if two lines cross, the answer will always be 'yes'. No attempt

`c.radius()` returns the radius of `c`.

`c.set_radius(r)` makes `r` the new value of the radius of `c`.

`c.set_angles(startangle, diffangle)` sets the begin angle and enclosed angle of `c`.

`c.begin_vt()` returns the starting vertex of the arc.

`c.end_vt()` returns the end vertex of the arc.

`c.begin_angle()` returns the angle of the begin vertex.

`c.end_angle` returns the angle of the end vertex.

`c.enclosed_angle` returns the difference between the end and begin angle (i.e. the last parameter of the constructor).

`c.small_vt()` when the arc is oriented counterclockwise (positive enclosed angle), this is the same as `c.begin_vt()`, otherwise `c.end_vt()`.

`c.big_vt()` returns the endpoint opposite to `c.small_vt()`.

`c.small_angle` when the arc is oriented counterclockwise (positive enclosed angle), this is the same as `c.begin_angle()`, otherwise `c.end_angle()`.

`c.big_angle()` when the arc is oriented counterclockwise (positive enclosed angle), this is the same as `c.end_angle()`, otherwise `c.begin_angle()`.

`c.length()` returns the length of the arc.

`c.supporting_disc()` returns the disc on which the arc lies.

`c.outer_approximation_d(result, epsilon)` approximates `c` by a number of vertices. Seen from the center, when the vertices are connected by edges, the edges lie outside `c`, but never further away than `epsilon`.

`c.outer_approximation_n(result, n)` approximates `c` by at least `n` vertices. Seen from the center, when the vertices are connected by edges, the edges lie outside `c`.

`c.inner_approximation_d(result, epsilon)` approximates `c` by a number of vertices. Seen from the center, when the vertices are connected by edges, the edges lie inside `c`, but never further away than `epsilon` from `c`.

`c.inner_approximation_n(result, n)` approximates `c` by at least `n` vertices. Seen from the center, when the vertices are connected by edges, the edges lie inside `c`.

```
    pl_vec      small_vt() const
    pl_vec      big_vt() const
    pl_angle    small_angle() const
    pl_angle    big_angle() const
    s_coord     length() const
    pl_disc     supporting_disc() const
    void        outer_approximation_d(pl_vecreray &r, s_coord eps) const
    void        outer_approximation_n(pl_vecreray &r, int n) const
    void        inner_approximation_d(pl_vecreray &r, s_coord eps) const
    void        inner_approximation_n(pl_vecreray &r, int n) const
    void        set_center(pl_fixedvec vt)
    void        set_radius(s_coord radius)
    void        set_angles(pl_angle angle, pl_angle diffangle)
};
```

Description

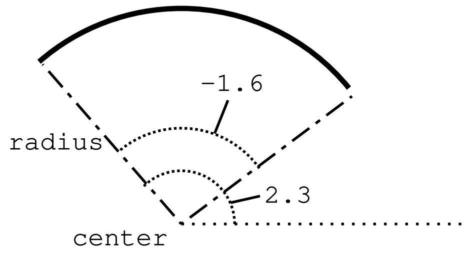
Circle arcs can be represented by the the type `pl_arc`. Arcs can be directed clockwise or counterclockwise.

Constructors

The first constructor leaves the arc undefined.

The second constructor takes four arguments. The first two arguments, center and radius, describe the circle of which the arc is a part. The third argument gives the angle of the start point of the arc. This angle is measured counterclockwise from a ray parallel to the x-axis through the center. The last argument is the number of radians the arc spans. This number should lie between -2π and 2π. A positive value means the arc is directed counterclockwise.

The third constructor is like the second, only the supporting circle is described by a disc instead of a center and a radius.



`arc(center, radius, 2.3, -1.6)`

Class Functions

`c.center()` returns the center of `c`.

`c.set_center(vt)` gives `c` a new center.

Class Functions

`c.center()` returns the center of `c`.

`c.radius()` returns the radius of `c`.

`c.area()` returns the area of `c`.

`c.set_center(vt)` gives `c` a new center.

`c.set_radius(r)` makes `r` the new value of the radius of `c`.

`c.outer_approximation_d(result, epsilon)` approximates `c` by a number of vertices. When the vertices are connected to form a polygon, `c` is contained in the polygon. The edge of the polygon is never more than `epsilon` away from `c`.

`c.outer_approximation_n(result, n)` approximates `c` by a number of vertices. When the vertices are connected to form a polygon, `c` is contained in the polygon. At least `n` vertices are used.

`c.inner_approximation_d(result, epsilon)` approximates `c` by a number of vertices. When the vertices are connected to form a polygon, the polygon is contained in `c`. The edge of the polygon is never more than `epsilon` away from `c`.

`c.inner_approximation_n(result, n)` approximates `c` by a number of vertices. When the vertices are connected to form a polygon, the polygon is contained in `c`. At least `n` vertices are used.

Remarks

In the previous version of PlaGeo there were routines `outer_approximation` and `inner_approximation` that had the same functionality as the current approximation routines with the `_d` suffix.

7.7 PL_ARC

Synopsis

```
class pl_arc {
                pl_arc()
                pl_arc(pl_fixedvec vt, s_coord radius,
                       pl_angle startangle, pl_angle diffangle)
                pl_arc(const pl_disc &c, pl_angle ang, pl_angle diffang)
    pl_vertex   center() const
    s_coord     radius() const
    pl_vertex   begin_vt() const
    pl_vertex   end_vt() const
    pl_angle    begin_angle() const
    pl_angle    end_angle() const
    pl_angle    enclosed_angle() const
```

`c.edge_mod(i)` gives the i-th edge of `c`, where i is taken modulo the size of the polygon (i may be negative).

`c.all_edges(result)` makes result a reray containing all edges of the polygon, so that `result[i]` is equal to `c.edge(i)`.

`c.is_ok()` checks whether the polygon is well-defined; returns 1 if this is the case, 0 otherwise.

`c.area()` returns the area of `c`.

`c.is_convex()` returns whether `c` is convex.

`c.is_clockwise()` returns whether the the vertices are specified in clockwise order.

`c.invert_orientation()` reorders the vertices such that a clockwise polygon turns into a counterclockwise one and vice versa.

Remarks

A lot of caching is done on polygons. For example, the first time the bounding box of a polygon is requested it is computed and stored internally. Another request will just deliver the stored bounding box. This caching information is copied during assignation of a polygon to another one.

7.6 PL_DISC

Synopsis

```
class pl_disc {
                pl_disc()
                pl_disc(pl_fixedvec vt, s_coord radius)
    pl_vec      center() const
    s_coord     radius() const
    s_coord     area() const
    void        set_center(pl_fixedvec vt)
    void        set_radius(s_coord radius)
    void        outer_approximation_d(pl_vecreray &r, s_coord eps) const
    void        outer_approximation_n(pl_vecreray &r, int n) const
    void        inner_approximation_d(pl_vecreray &r, s_coord eps) const
    void        inner_approximation_n(pl_vecreray &r, int n) const
};
```

Description

Filled circles are represented by the type `pl_disc`.

Constructors

The first constructor leaves the disc undefined. The second constructor takes the center of the disc and the radius as parameters.

7.5 PL_POLYGON or PL_PGN

Synopsis

```
class pl_polygon {
                pl_polygon()
                pl_polygon(pl_vecreray vl)
    int         size() const
    pl_vec      vertex(int i) const
    pl_vec      vertex_mod(int i) const
    pl_vecreray all_vertices() const
    pl_edge     edge(int i) const
    pl_edge     edge_mod(int i) const
    void        all_edges(pl_edgereray &result) const
    int         is_ok() const
    s_coord     area() const
    int         is_convex() const
    int         is_clockwise() const
    void        invert_orientation()
};
```

Description

The type `pl_polygon` is used to represent polygons, i.e. triangles, quadrilaterals and so on. The routines that deal with polygons expect them to be simple: edges of the polygons are not allowed to intersect. Polygons cannot have holes and are always finite. A shorthand that can be used instead of the whole name is `pl_pgn`.

Constructors

The first constructor initializes `c` to the empty polygon (no vertices. The second constructor takes a reray of vectors as argument. Those are the vertices, in order, of the polygon. The last vertex is automatically connected to the first, so the first vertex should not be duplicated.

Class Functions

`c.size()` gives the number of vertices (or edges) of `c`.

`c.vertex(i)` gives the i-th vertex of the polygon. Vertices are numbered from 0 upwards.

`c.vertex_mod(i)` gives the i-th vertex of the polygon, where i is taken modulo the size of the polygon (i may be negative).

`c.all_vertices()` returns a reray containing all vertices of the polygon, so that `result[i]` is equal to `c.vertex(i)`.

`c.edge(i)` gives the i-th edge of `c`, i.e. the edge from vertex i to vertex $i+1$ (vertex 0 for the last edge).

Class Functions

`c.begin_vt` returns the endpoint of the ray.

`c.direction()` delivers the direction of the ray.

`c.supporting_line()` returns the line on which the ray lies. The direction of the ray and line is the same.

7.4 PL_EDGE

Synopsis

```
class pl_edge {
                pl_edge()
                pl_edge(pl_fixedvec vt1, pl_fixedvec vt2)
    pl_vec      begin_vt() const
    pl_vec      end_vt() const
    pl_line     supporting_line() const
    s_coord     length() const
    pl_unitvec  direction() const
};
```

Description

Edges or line segments are represented by the type `pl_edge`. Like rays and lines, edges are directed. The direction points from the begin point to the end point.

Constructors

The first constructor leaves the edge undefined. The second constructor initializes the edge with a begin vertex and an end vertex.

Class Functions

`c.begin_vt()` returns the begin point of `c`.

`c.end_vt` returns the endpoint of `c`.

`c.supporting_line` returns the line on which `c` lies. The direction of the edge and the line are the same.

`c.length` returns the length of `c`.

`c.direction()` returns the direction of `c`.

```
int almost_parallel(const pl_line &line1, const pl_line &line2,
                    s_coord dir_tol)
{
  return fabs_pp(cross(line1.direction(),line2.direction())) <= dir_tol;
}

int almost_equal(const pl_line &line1, const pl_line &line2,
                 s_coord dir_tolerance, s_coord dist_tolerance)
{
    if (!almost_parallel(line1, line2, dir_tolerance))
        return 0;
    vt = line2.ref_vertex();
    return (vt-project(vt, line1)).length() <= dist_tolerance;
}
```

In the first routine the directions are compared. If the directions are almost the same, the cross product of the unit direction vectors must be small (the cross product is the sine of the enclosed angle). Notice the use of `fabs_pp`, which takes the absolute value of either a float or a double.

If the lines are parallel, the second routine computes the distance from a vertex on the second line to the first line. This might have been done easier with the clearance routine that is described later on. Here the vector from the vertex to its projection on the line is constructed and the length of this vector is taken.

7.3 PL_RAY

Synopsis

```
class pl_ray {
                pl_ray()
                pl_ray(pl_fixedvec vt, pl_unitvec uvec)
    pl_vec      begin_vt() const
    pl_unitvec  direction() const
    pl_line     supporting_line() const
};
```

Description

The type `pl_ray` represents half lines or rays. Rays are directed: the direction of a ray always points away from its endpoint.

Constructors

The first constructor leaves the ray undefined. The second constructor takes the endpoint of the ray and the direction.

`c.normal()` is a shorthand for `c.direction().normal()`.

`c.placevec(co)` returns the point on `c` that lies `co` away from the reference vertex (taking a signed distance. So the result is `c.ref_vt() + co*c.direction()`.

`c.dirvec(co)` returns a vector of length |`co`| along `c`. To be precise, the result is `co*c.direction()`.

`c.placeproject(vec)` returns the projection point of `vec` on `c`. The result is given as a coordinate on `c`. The corresponding vector can be found as `c.placevec(result)`.

`c.dirproject(vec)` returns the component of `vec` along `c`. The corresponding vector cab be found as `c.dirproject(result)`.

Other Routines

```
pl_vec placeproject(pl_fixedvec vec, pl_line l)
```

returns the perpendicular projection of vector `vec` on the line `l`. The vector is taken as a place vector.

```
pl_vec dirproject(pl_fixedvec vec, pl_line l)
```

returns the projection of vector `vec` on the line `l`, where the vector is taken as a direction vector. That is, the component of the vector along `l` is returned.

The routines

```
int is_to_the_left(pl_fixedvec vt, pl_line l)
int is_to_the_right(pl_fixedvec vt, pl_line l)
```

decide whether a vertex lies to the left (right) of a line. Here the direction of the line is taken into account.

The two routines

```
int is_above(pl_fixedvec vt, pl_line l)
int is_below(pl_fixedvec vt, pl_line l)
```

can be used to find out if a vertex lies above or below a line. When the line is vertical, a point left of the line is said to be above.

Examples

As was said in the introduction, there is no routine to decide if two lines coincide. Here we give an example how such a routine can be written. The routine `almost_equal` takes the two lines as parameter and furthermore a direction tolerance and a distance tolerance, two numbers that indicate how much the lines may differ.

The routine uses the help function `almost_parallel` to decide if the two lines have the same direction.

Examples

The next routine computes the distance between a point and its image under a transformation. First the point is copied. This copy is transformed and the distance between the two points is returned. The clearance function is described in the sequel.

```
s_coord image_dist(pl_point vt, pl_rotra tfm)
{
    pl_point vt2(vt);
    vt2.transform(tfm);
  // return the length of the vector joining the two points
    return clearance(vt2, vt1);
}
```

7.2 PL_LINE

Synopsis

```
class pl_line {
                pl_line()
                pl_line(pl_fixedvec vt1, pl_fixedvec vt2)
                pl_line(pl_fixedvec vt, pl_unitvec uvec)
    pl_vec      ref_vt() const
    pl_unitvec  direction() const
    pl_unitvec  normal() const
    pl_vec placevec(s_coord co) const
    pl_vec dirvec(s_coord co) const
    s_coord     placeproject(pl_fixedvec vec) const
    s_coord     dirproject(pl_fixedvec vec) const
};
```

Description

Lines in the plane can be described with the type `pl_line`. Lines are directed, though this fact can be ignored if one wishes.

Constructors

The first constructor leaves `c` undefined. The second constructor delivers a line through `vt1` and `vt2`, directed from `vt1` to `vt2`. The third constructor initializes `c` to a line through `vt` in the direction of `uvec`.

Class Functions

`c.ref_vt()` returns a vertex on the line. The vertex that is returned is random, but guaranteed to be the same for subsequent calls.

`c.direction()` returns the direction of the line.

Chapter 7

Planar Objects

This chapter describes the geometric objects of PlaGeo. All those objects are derived from the type `pl_object`, described earlier. The routines that they inherit from this ancestor are not described here.

7.1 PL_POINT

Synopsis

```
class pl_point: public pl_object {
public:
            pl_point()
            pl_point(pl_fixedvec vec)
            pl_point(s_coord a, s_coord b)
    pl_vec  vec() const
};
```

Description

`pl_point` is the type for points in the plane. A point is described by a single vector, its place relative to the origin. Therefore, a point can either be represented by a `pl_point` or a `pl_vec`. The basic difference between a point and a vector is that the type `pl_point` is derived from `pl_object`. This means that it can be used in more abstract ways. For example, a pointer to a point can be a member of an array of pointers to objects. If this is not used, the vector representation is probably more appropriate in most cases.

Constructors

The first constructor leaves the point undefined. The second constructor initializes `c` with a vector, the third one with the corresponding x- and y-value.

Class Functions

`c.vec()` returns the vector corresponding with `c`.

```
                  pl_ortho_tfm(pl_angle angle, s_coord fac, pl_fixedvec vec)
    pl_ortho_tfm  operator*(pl_ortho_tfm tfm) const
    void          get_params(pl_angle &angle, s_coord &fac, pl_vec &vec)
};
```

Description

The most general transformation in PlaGeo is the type `pl_ortho_tfm`. This type describes general orthogonal (angle preserving) transformations, consisting of a rotation, a uniform scaling and a translation. The effect of this transformation on an object is the same as the effect of those three consecutive transformations, so first the rotation, then the scaling and finally the translation.

Constructors

The constructor takes the rotation angle, the scaling factor and the translation factor as parameters. If no parameter is supplied, the `pl_ortho_tfm` is initialized as the identity transformation.

Class Functions

`c.get_params(angle, fac, vec)` assigns to the parameters the values that describe this `pl_ortho_tfm`.

`c * tfm` returns a new `pl_ortho_tfm` that is the composition of `c` and `tfm`.

Class Functions

`c.angle()` returns the rotation angle. This angle may differ by a multiple of 2π from the value that was supplied.

6.4 PL_ROTRA

Synopsis

```
class pl_rotra: public pl_unit_tfm {
public:
                pl_rotra(pl_angle theta, pl_fixedvec vec)
                pl_rotra(pl_angle theta)
                pl_rotra(pl_fixedvec vec)
                pl_rotra()
    pl_rotra    inverse() const;
    pl_rotra    operator*(pl_rotra tfm) const;
    void        get_params(pl_angle &theta, pl_vec &vec) const;
};
```

Description

The type `pl_rotra` is used to describe coordinate transformations consisting of a rotation followed by translation. A counterclockwise rotation is positive, a clockwise rotation is negative.

Constructors

The first constructor is the most complete: both the angle and the translation are specified. In the other constructors one or two of them are left out. In that case a zero rotation or translation is assumed, so the last constructor initializes `c` to the identity transformation.

Class Functions

`c * tfm` returns a new transformation which is product of `c` and `tfm`. This means that for every coord `co` `(c*tfm)*co` is equal to `c*(tfm*co)`.

`c.inverse()` returns the inverse transformation of `c`.

`c.get_params(theta,vec)` returns the angle and the vector by which `c` is described. This can be seen as the inverse of the constructor. The angle, however, is normalized in the range $[-\pi, \pi]$.

6.5 PL_ORTHO_TFM

Synopsis

```
class pl_ortho: public pl_transform {
                  pl_ortho_tfm()
```

Class Functions

`c.factor()` returns the scaling factor of `c`.

6.2 PL_TRANSLATION

Synopsis

```
class pl_translation: public pl_unit_tfm {
                pl_translation()
                pl_translation(pl_fixedvec vec)
    pl_vec      vec() const
};
```

Description

Translations over a vector can be represented by the type `pl_translation`. When applied to an object, this vector is added to all coordinates. Translations do not alter lengths, angles and directions.

Constructors

The translation vector is given as a parameter of the constructor. When no parameter is given, the translation vector will be initialized to the null vector.

Class Functions

`c.vec()` returns the translation vector.

6.3 PL_ROTATION

Synopsis

```
class pl_rotation: public pl_unit_tfm {
                pl_rotation()
                pl_rotation(pl_angle theta)
    pl_angle    angle() const
};
```

Description

Rotations around the origin are represented by the type `pl_rotation`.

Constructors

The rotation constructor takes the angle as parameter. When no parameter is supplied, the value 0 is assumed.

Chapter 6

Planar Transformations

This chapter describes the different coordinate transformations of PlaGeo. Geometric objects can be scaled (uniformly), translated and rotated. The classes that represent those transformations are described here. All classes are derived from the base class `pl_transform`, that was described in section 5.4.

6.1 PL_UNISCALE

Synopsis

```
class pl_uniscale: public pl_transform {
                pl_uniscale()
                pl_uniscale(s_coord f)
    s_coord     factor() const
};
```

Description

The type `pl_uniscale` describes uniform scaling transformations. When this transformation is applied to an object, all coordinates are multiplied by a factor. This factor may be negative.

When the factor is positive, angles and unit vectors (directions) are not altered by the transformation. Lengths (`s_coord`) and coordinates (`pl_vec`) are multiplied by the factor.

When the factor is negative, the transformation returns the inverse of unit vectors, π is added to angles and lengths are multiplied with the absolute value of the scaling factor.

Constructors

The scaling factor is given as a parameter for the `pl_uniscale`. When no parameter is given, the scale factor will be 1, so it will result in an identity transformation.

The scaling factor may be negative, or even zero. Scaling objects with a zero factor can result in bad results, though.

Class Functions

`c.bbox()` returns the bounding box of the object, that is the smallest axis parallel box that contains all the point of `c`. For objects that extend infinitely, like lines, a bounding box of maximal size is returned.

`c.transform(tfm)` transforms `c` according to `tfm`.

`c.rotate(angle)` rotates `c` over `angle`.

`c.translate(vec)` translates `c` over `vec`.

`c.scale(co)` scale `c` by a factor `co`.

`c.type()` returns a value of type `pl_object_type`, which indicates the type of object.

Examples

The type inquiry facility can be used to make heterogeneous lists. The types `pl_point` and `pl_edge` are both derived from `pl_object`.

```
typedef pl_object *objp;

typedef reray<objp> objpreray;

int count_points(objpreray orr)
{
  int i, count=0;
  for (i=0; i<orr.size(); i++)
    if (orr[i]->type() == PL_POINT) count++;
  return count;
}

main()
{
  objpreray orr(3);
  orr[0] = new pl_point;
  orr[1] = new pl_edge;
  orr[2] = new pl_point;
  cout << count_points(orr) << '\n';    // this will output 2
}
```

Description

There are a number of operations –translating, rotating and uniform scaling– that can be applied geometric objects. When such a transformation is applied to an object, in general a valid object of the same type will result. This is not surprising, as there is a freedom to choose the origin, the x-axis and the length unit.[1]

To describe a transformation, it suffices to describe the effect of the transformation on the basic geometric types: place vectors, direction vectors, lengths and angles. In fact, one could express the effect on the last three in terms of the effect on place vectors, but that is not done here.

The base class from which all transformations are derived is called `pl_transform`. Derived from this class is the class `pl_unit_tfm`, which is the base class for all unit transformations, i.e. transformations that do not change distances.

Class Functions

`c * vec` returns the transformed place vector.

`c * uvec` returns the transformed direction vector.

`c * co` returns the transformed length.

`c * angle` returns the transformed angle. This acts on absolute angles, that is angles with the positive x-axis. Relative angles (angles between two lines or two direction vectors) are not influenced by transformations.

5.5 Planar Objects

Synopsis

```
enum pl_object_type {PL_POINT, PL_LINE, PL_RAY, PL_EDGE,
                    PL_POLYGON, PL_DISC, PL_ARC};

class pl_object {
    pl_box          bbox()
    void            transform(pl_transform tfm)
    void            rotate(pl_angle angle)
    void            translate(pl_fixedvec vec)
    void            scale(s_coord co)
    pl_object_type  type()
};
```

Description

All geometric types in the sequel are derived from the type `pl_object`. All objects can be transformed by orthogonal transformations. They all have a bounding box and can return information about what type they are.

[1] An exception are axis parallel boxes, which are tied to a particular coordinate frame (so do not remain axis-parallel when rotated).

`c += bb` makes `c` the union of `c` and `bb`, i.e. a box just big enough to contain both boxes.

Other Routines

`pl_box intersect(pl_box bb1, pl_box bb2)`

computes the intersection of boxes `bb1` and `bb2`.

`int do_intersect(pl_box bb1, pl_box bb2)`

computes if the `bb1` and `bb2` intersect.

`pl_box operator+(pl_box bb1, pl_box bb2)`

computes a box just big enough to contain both `bb1` and `bb2`.

`int lies_inside(pl_fixedvec vec, pl_box bb)`

computes if the point `vec` lies inside `bb`.

`int lies_inside(pl_box bb1, pl_box bb2)`

returns whether the `bb1` lies totally inside `bb2`.

Remarks

The different `add_point` routines exist out of efficiency considerations. It is a bit cheaper if there is no need to check for an empty box.

5.4 Transformations

Synopsis

```
class pl_transform {
    pl_vec      operator*(pl_vec vec)
    pl_unitvec  operator*(pl_unitvec uvec)
    s_coord     operator*(s_coord co)
    pl_angle    operator*(pl_angle angle)
};

class pl_unit_tfm : public pl_transform {};
```

```
    s_coord     ymax() const
    s_coord     ymin() const
    s_coord     area() const
    void        make_empty()
    void        add_point(pl_fixedvec vec)
    void        add_first_point(pl_fixedvec vec)
    void        add_non_first_point(pl_fixedvec vec)
    void        translate(pl_fixedvec vec)
    void        scale(s_coord fac)
    void        operator+=(pl_box bb)
};
```

Description

A box is an axis-parallel box. A box is different from other geometric objects in that it is tied to a particular coordinate system (the one it is parallel to). In particular, boxes can not be rotated, as they would not remain axis parallel in that case. This reflects the fact that boxes normally do not correspond to object in the world that is described, but rather to things that are used during computations.

Boxes are used internally for the optimization of algorithms. For example, to examine if two objects intersect, first a bounding box can be constructed around them to look if the boxes intersect. Only if the bounding boxes intersect –a simple test– a precise test must be performed.

Constructors

The first constructor leaves `c` undefined. The second initializes `c` to a degenerated box, containing only the point `vec`.

Class Functions

`c.is_empty()` is a boolean function that indicates whether `c` is empty.

`c.xmax()` gives the biggest x-value of the box.

`c.xmin() c.ymax() c.ymin()` analogous.

`c.area()` returns the area (width times height) of `c`.

`c.make_empty` makes `c` empty.

`c.add_point(vec)` enlarges `c` if necessary, so that it contains point `vec`.

`c.add_first_point(vec)` equivalent with `{ c.make_empty(); c.add_point(vec); }`.

`c.add_non_first_point(vec)` defined only if `c` is not empty. In that case equivalent with `c.add_point(vec)`.

`c.translate(vec)` translates `c` over a vector `vec`.

`c.scale(fac)` scales `c`, i.e. multiplies all coordinates with `fac`.

returns the cross product of `vec1` and `vec2`, i.e.
`vec1.x() * vec2.y() - vec1.y() * vec2.x()`

`const pl_unitvec pl_xdir, pl_ydir`

Those are two predefined unit vectors in the direction of the positive x- and y-axis.

`s_coord factorize(pl_unitvec &uvec, pl_fixedvec vec)`

Factorizes a vector `vec` in a unitvec `uvec` and a length (the return value of the function). `vec` may be the null vector, in which case `uvec` will become a random unit vector.

`pl_unitvec normalized(pl_fixedvec vec)`

Returns a unit vector in the same direction as `vec`.

Examples

In the following examples some of the previous routines are used. After the double slash (a C++ style comment) the value of some variables is shown.

```
        pl_vec c1;                     // value of c1 undefined
        pl_vec c2(3.1, 1.4);           // value of c2: (3.1, 1.4)
        c1.set_x(1);
        c1.set_y(c2.x());                 // value of c1: (1 , 3.1)
        if (c1 <= c2)                   // true, because  1 < 3.1
            c2 = c1;                    // value of c2: (1 , 1.4)

foo()
{
    s_coord x;
    pl_vec vec1, vec2;
    pl_vec vec3(12, -1);
    vec2 = pl_vec(3, 4);
    vec1 = vec3 - vec2;     // vec1 == (9, -5)
    vec1 *= vec2.length();  // vec1 == (45, -25)
    x = dot(vec2, vec3);    // x == 32
}
```

5.3 PL_BOX

Synopsis

```
class pl_box {
                pl_box()
                pl_box(pl_fixedvec vec)
    int         is_empty() const
    s_coord     xmax() const
    s_coord     xmin() const
```

Other Routines

The following functions implement a lexicographic ordering on `pl_coords`, first on x, then on y. So: $a < b \equiv a.x() < b.x() \bigvee (a.x() = b.x() \bigwedge a.y() < b.y())$

```
int operator< (pl_fixedvec a, pl_fixedvec b)
int operator> (pl_fixedvec a, pl_fixedvec b)
int operator<= (pl_fixedvec a, pl_fixedvec b)
int operator>= (pl_fixedvec a, pl_fixedvec b)
```

```
pl_angle angle_of(pl_fixedvec vec)
```

`angle_of(vec)` returns the angle that `vec` makes with the positive x-axis.

```
pl_angle angle_between(pl_unitvec uvec1, pl_unitvec uvec2)
```

returns the angle between `uvec1` and `uvec2`.

```
pl_vec to_polar(pl_fixedvec vec)
```

converts a set of Cartesian coordinates to polar coordinates. The radius is returned as the x-coordinate, the angle (in radians) is returned as the y-coordinate.

```
pl_vec from_polar(pl_fixedvec vec)
```

converts from polar coordinates to Cartesian coordinates. The x-part of `vec` (the radius) should be positive.

```
pl_vec operator-(pl_fixedvec vec)
```

`-vec` returns the inverse vector (the vector pointing in the opposite direction).

```
pl_vec operator*(s_coord l, pl_fixedvec vec)
```

`l*vec` returns `vec` multiplied with `l`.

```
pl_vec operator+(pl_fixedvec vec1, pl_fixedvec vec2)
```

`vec1 + vec2` returns the sum of the two vectors `vec1` and `vec2`.

```
pl_vec operator-(pl_fixedvec vec1, pl_fixedvec vec2)
```

`vec1 - vec2` returns the difference of the two vectors `vec1` and `vec2`.

```
s_coord dot(pl_fixedvec vec1, pl_fixedvec vec2)
```

returns the dot product of `vec1` and `vec2`, i.e.
`vec1.x() * vec2.x() + vec1.y() * vec2.y()`

```
s_coord cross(pl_fixedvec vec1, pl_fixedvec vec2)
```

Derived from `pl_fixedvec` are `pl_vec` and `pl_unitvec`. The type `pl_vec` supplies routines to alter its coordinates. It is the type normally used for vectors. Vectors of length one can be represented by the type `pl_unitvec`. Unit vectors are normally used to describe directions.

Routines normally take `pl_fixedvec`s as parameters , but return `pl_vec`s. Because all vector classes have `pl_fixedvec` as base class, they can all be passed as an argument when a `pl_fixedvec` is expected.

Constructors

Constructors without parameters leave the vector in an undefined state. For unit vectors the length of the vector is guaranteed to be one. The unit vector constructor with the angle as argument makes a unit vector pointing in the direction of `angle`. That is, if the angle is zero the vector points along the positive x-axis, if the angle is $\pi/2$ the vector points along the positive y-axis. The other constructors are straightforward.

Class Functions

First the class functions of `pl_fixedvec` are described. Because the other two vector types are derived from this type, those functions can be called for them as well.

`c.x()` returns the x-coordinate of `c`.

`c.y()` returns the y-coordinate of `c`.

`c.length()` returns the length of the vector.

`c.sqlength()` returns the square of the length.

Here are the functions that are only valid for `pl_vec`s.

`c.set_x(x)` sets the x-coordinate of `c` to `x`.

`c.set_y(y)` sets the y-coordinate of `c` to `y`.

`c *= co` scales `c` with a factor `co`.

`c += vec` adds `vec` to `c`.

`c -= vec` subtracts `vec` from `c`.

Here are the class functions for unit vectors.

`c.normal()` returns a unit vector perpendicular to `c`. The direction is the direction of `c`, rotated counterclockwise over an angle of 90 degrees.

`-c` returns the unit vector in the opposite direction of `c`.

`c.value()` returns the current value in radians.

`c.normalized_value(lowlimit)` returns the value in radians, but normalized in the indicated range. The value of `c` is not changed.

`c.lies_between(small, big)` answers if `c` lies between `small` and `big`. All the arithmetic is done modulo 2π. `c` lies in between `small` and `big` if, starting from `small`, walking counterclockwise over the unit circle `c` is reached before `big`.

5.2 Vectors

Synopsis

```
class pl_fixedvec {
                pl_fixedvec()
                pl_fixedvec(s_coord x, s_coord y)
    s_coord     x() const
    s_coord     y() const
    s_coord     length() const
    s_coord     sqlength() const
};

class pl_vec: public pl_fixedvec {
                pl_vec()
                pl_vec(pl_fixedvec vec)
                pl_vec(const s_coord x, const s_coord y)
    void        set_x(s_coord x)
    void        set_y(s_coord y)
    void        operator*=(s_coord co)
    void        operator+=(pl_fixedvec vec)
    void        operator-=(pl_fixedvec vec)
    void        operator=(pl_fixedvec vec)
};


class pl_unitvec: public pl_fixedvec {
                    pl_unitvec()
                    pl_unitvec(pl_angle angle)
    pl_unitvec      operator-() const
    pl_unitvec      normal() const
    void            operator=(pl_unitvec uvec)
};
```

Description

There are three types for defining two dimensional vectors. The base vector class is `pl_fixedvec`. This type represents vectors with x- and y-coordinates. The contents of a `pl_fixedvec` can not be altered. There is even no assignment operator defined for `pl_fixedvecs`.

Chapter 5

The Basic Types of PlaGeo

This chapter describes the basic geometric types of PlaGeo. That is, the types that deal with coordinates –different kinds of vectors and also boxes that have sides parallel to the coordinate axes– and angles, and types that serve as common basis for other types (for geometric objects and for geometric transformations).

5.1 PL_ANGLE

Synopsis

```
class pl_angle {
                pl_angle()
                pl_angle(s_coord v)
    void        normalize(s_coord lowlimit = -M_PI)
    s_coord     value() const
    s_coord     normalized_value(s_coord lowlimit = -M_PI) const
    int         lies_between(pl_angle small, pl_angle large) const
};
```

Description

The type `pl_angle` is used to represent angles. Angles are expressed in radians. A peculiarity of angles is that two values that differ by a multiple of 2π may indicate the same angle.

Constructors

The first constructor leaves the angle undefined, the second constructor expects the angle in radians.

Class Functions

`c.normalize(lowlimit)` normalizes the value of the angle to the interval [`lowlimit`, `lowlimit` $+ 2\pi$) by subtracting or adding the appropriate multiple of 2π. When no argument is give the interval $[-\pi, \pi)$ is taken.

Part II

PlaGeo

4.2 I/O of geometric types

The main reason for doing IO in the way C++ does it, is that it is easily extendible to user defined types. The Geo libraries enable the user to write and read any type of geometrical object in the same way as the built in types.

The output only contains numerical values and whitespace. There is no leading or trailing whitespace written, so, in order to separate consecutive objects, the user has to do this.

The file format is described in appendix A. Normally the output corresponds quite closely to one of the constructors of the objects.

The following fragment gives an example. The types that are used here are types of PlaGeo. They are described in later sections.

```
#include <fstream.h>
void write_it()
{
    pl_edge edge(pl_vec(0, 1), pl_vec(10, 0));
    pl_vec vt(5, 6);
    ofstream os("data");
    os << vt << ' ' << edge << '\n';
}

void read_it()
{
    pl_edge edge;
    pl_vec vt;
    ifstream is("data");
    is >> vt >> edge;
}
```

In order to read from a stream, the shiftright operator is used, to write to a stream the shiftleft operator is used; the arrows of the operator are pointing in the direction of the data flow. Those shift operators are overloaded, so C++ automatically uses the right procedure to read or write a given type. In the last line of the program we see that it is possible to write several objects, of different types, in one statement. In this case first a double is written, followed by a string, an integer and a char (a linefeed in this case). In the same way it would be possible to read several objects with one statement, where the objects should be separated by shiftright operators.

It is a good practice to check whether everything has gone right during input or output. To check this, the streams have a member function which can be called as follows:

```
if (!cin.good()) error("Something went wrong during reading");
```

Once an error is detected, further operations on a stream will fail, i.e.,no data will be read or written.

Until now, input and output was done on the predefined streams `cin` and `cout`. If the program wants to read or write a particular file it can do so by declaring an ifstream or ofstream respectively. For example, data can be read from a file named 'data' and then something is written to the file 'results'.

```
#include <fstream.h>

int i;
ifstream source("data");
if (!source)
  cerr << "unable to open 'data' for input.\n";
ofstream target("results");
if (!target)
  cerr << "unable to open 'results' for output.\n";
source >> i;
if (i == 42)
  target << "Input file starts with non random number.\n";
```

It is also possible to declare a file stream first and to open it later. For example:

```
ifstream file;
char *name;
...
file.open(name);
...
file.close();
```

Of course, much more can be said about iostreams. For more information on the subject see some book on C++ or the corresponding manual pages.

Chapter 4

IO of Objects

PlaGeo and SpaGeo have some support to write geometric objects to files in text format and to read them in again. This support is based on the standard way of doing IO in C++, the iostream library. First a brief description is given of this library, which can be skipped by those familiar with the C++ IO mechanism. Then the extensions are described.

4.1 Iostream Basics

Although it is still possible to read and write text files in the C-way, using `scanf` and `printf`, there is another mechanism in C++ that is preferred: the iostream library. As an introduction, consider the following program.

```
#include <iostream.h>
main()
{
    double f;
    cin >> f;
    cout << f << " truncated is " << (int)f << '\n';
}
```

This is equivalent to the following C-program.

```
#include <stdio.h>
main()
{
    double f;
    scanf("%lf", &f);
    printf("%lf truncated is %d\n", f, (int)f);
}
```

In the iostream library, cin is used for standard input, cout for standard output. (There is a third predefined stream, cerr, that can be used for standard error output.) In this example, a floating point number is read from standard input. In the next line this number and its truncated value are written to standard output. For example, when the input is `13.801` the output will be: `13.801 truncated is 13`.

```
}

foo(float *data, int datalength)
{
    int i;
    intreray elem_of_order(datalength);
    for (i=0; i<datalength; i++)
        elem_of_order[i] = i;
    global = data;
    elem_of_order.sort(compare);

    intreray order_of_elem(datalength);
    for (i=0; i<datalength; i++)
        order_of_elem[elem_of_order[i]] = i;
}
```

Caveats

When a reray grows, it is sometimes necessary to reserve a new chunk of memory for the reray and copy the elements from the old to the new position. Normally this is not visible to the user, but there are two cases that sometimes lead to unexpected behavior.

As was mentioned before, the assignment operator should behave as expected for the element type of the reray. Especially when the element type contains a pointer that is used as a dynamic array –the result of malloc or new is stored there– care should be taken.

Be very careful with references in the reray. They become invalid whenever the reray is resized. The following is an example of dangerous code.

```
/* irr is a reray of integers */
int *ip = &irr[2];      // ip is a pointer to a reray element
int &ir = irr.last();   // ir is a reference to a reray element
irr.append(107);        // this may invalidate ip and ir
*ip = 12;               // unsafe
ir = 13;                // unsafe
```

```
    if (x.a > y.a) return -1;
    if (x.a < y.a) return 1;
    if (x.b > y.b) return -1;
    if (x.b < y.b) return 1;
    return 0;
}

main()
{
    mytype_rr mrr;
    mytype mt;
    mt.a = 2; mt.b = 12;
    mrr.newsize(2);
    mrr[0] = mt;
    mt.b = 7;
    mrr[1] = mt;
    mrr.sort(lexicomp);
    mt = mrr[0];
}
```

Tips

Use static rerays in procedures. They will retain their storage between calls. Although this is less space efficient, there will be a lot less of dynamic memory allocation.

```
foo()
{
    // initialize once
    // keep memory allocated
    static reray<T> trr;
    trr.newsize(0);
    ...
}
```

```
foo()
{
    // initialize and allocate
    // every time foo() is called
    reray<T> trr;
    ...
}
```

More about sorting

Sometimes one wants to have a sorted index in some data structure. Here we give an example how rerays can be used to do this. We want to create an index for an array of floats. (we do not sort the array itself; maybe we want to keep the array in the same order or maybe we just are too lazy and want to use the reray sort).

```
float *global;

int compare(int index1,int index2)
{
    if (global[index1] > global[index2]) return 1;
    if (global[index1] < global[index2]) return -1;
    return 0;
```

efficiency is important, it is worth looking if an assignment to a reray can be turned into a swap of contents.

Remarks

Normally, no out of bounds checking is done on rerays. This behavior can be altered by defining a precompiler variable `RERAYCHECKS`, either by having a line

```
#define RERAYCHECKS 1
```

in the source files where checks should be on, or by giving the option `-DRERAYCHECKS` on the compile line, so something like

```
CC -I/usr/staff/include -DRERAYCHECKS -c file.C
```

Not checking is of course faster, but can lead to strange errors, so during the development of a program it is advisable to turn checking on. When an out of bounds error is detected, the program will be aborted and a message is written to standard output. A core file is dumped, which can be inspected with a debugger to see where the bug happened.

When it is known beforehand how many elements a reray will contain, it is more efficient to declare the reray with the appropriate length (or to do newsize) and then assign to the consecutive elements than to start with an empty reray and do appends.

However, the second method is not as bad as it might seem at first glance. For each reray two sizes are kept: the size of the reray as known to the outside (with `size()`) and an internal size, that gives the space that is really available. Only when the internal size is too small, resize and newsize should allocate new space. In this case some extra space is reserved for future use. In this way a series of n appends will cost $O(\log n)$ allocations.

The procedure `compactify` makes the internal size equal to the external size and then frees the memory. Normally this involves the copying of all the elements of the reray, which makes it a costly procedure when the reray is big.

Rerays are implemented in a separate module, not as part of PlaGeo, so they can be used without using PlaGeo. In that case the header file `reray.h` must be included in the program.

Examples

The following program gives an example of the use of rerays. A type is declared (`mytype`) and a reray of this type (`mytype_rr`). An ordering function on this type is defined. A reray variable is declared and some elements are placed in the reray. After this the reray is sorted and the first element is assigned to a variable.

```
struct mytype {
    int a, b;
};

typedef reray<mytype> mytype_rr;

int lexicomp(const mytype &x, const mytype &y)
{
```

The brackets in this declaration indicate that templates, a C++ concept for parameterized classes, are used.

Constructors

The first constructor initializes `c` to a reray of length zero (no elements). The second constructor initializes `c` to a reray of length s.

Class Functions

`c.size()` gives the number of elements of `c`.

`c[i]` gives element `i`. This can not only be used to get the value of an element, but also to assign a value to an element: `c[i] = ..` is legal. Normally, no check is made to make sure that `i` is within the bounds of `c`, but see how this behavior can be changed in the remarks below.

`c.last()` returns the last element of the reray, so is equivalent with `c[c.size()-1]`.

`c.newsize(n)` makes the size of `c` equal to `n`. After this operation the contents of `c` is not defined.

`c.resize(n)` makes the size of `c` equal to `n`. The elements that remain valid will retain their value. If `n` is bigger than the old size, the new values are not defined. Because elements are copied, `resize` takes more time than `newsize`.

`c.grow(i)` grows the reray with `i` elements (or shrinks it if `i` is negative. This is equivalent with `c.resize(c.size()+i)`.

`c.append(s)` appends an element to the back of `c`. This is equivalent with:
`{c.grow(1); c.last() = s; }`

`c.compactify()` is semantically a no-op. It frees memory that is used by `c` but is not actually needed. Do not use it, unless memory usage is a real trouble.

`c.sort(compare)` sorts the elements of `c` in ascending order according to `compare`. `compare` must be a function that takes two Ts (or, preferably, two const references to T) and returns -1, 0 or 1 with the meaning that the first element is smaller, equal respectively bigger than the second. The `compare` function must be a valid ordering (a transitive relation), otherwise the result of sorting is undefined.

`c.swap_contents(trr)` is a routine to swap the contents of two rerays. Semantically, this function acts like the following code fragment:

```
reray<T> tempreray = c;
c = rr2;
rr2 = tempreray;
```

The performance is a lot better, though. The swapping of the contents is done in constant time (irrespective of the length of the reray). An assignment has a cost proportional to the length of a reray, because all the elements must be copied. So, if

```
sqrt_pp(x);
```

will call the first sqrt-function if `x` is of type double, the second one if `x` is of type float. Instead of the standard mathematical functions these ones were used consequently.

Those functions can of course also be used by the user, not only in programs using this library, but in any program. Just make sure to include the header file `math++.h`.

3.2 Rerays

Synopsis

```
class reray {
                reray()
                reray(unsigned int s)
    int         size() const
    T &         operator[](unsigned int i) const
    T &         last() const
    void        newsize(unsigned int)
    void        resize(unsigned int)
    void        grow(int)
    void        append(const T &s)
    void        compactify() const;
    void        sort(int (* compare)(const T &, const T&));
    void        sort(int (* compare)(T, T))
    void        swap_contents(reray<T> &trr)
};
```

Description

This section describes a family of classes, resizable arrays or rerays for short. Just like arrays, rerays can have any type or class as element type. This element type is given the name T in the synopsis. Individual elements can be accessed by means of a subscript operator. The first element has subscript 0.

The main difference with arrays is that rerays can grow: elements can be appended and the reray can be resized. The memory management that is needed for this is done transparently to the user. Another difference is that a reray can be assigned to another reray. In this case all elements are copied.

Reray variables are not just of type reray, but of type reray of some type, like an array variable can be of type array of int or array of float, not just of type array.

Because some of the operations on rerays copy elements, the assignment operator (`=`) should be valid for the element type. That is, if a reray of type `Y` is used, and `t1` and `t2` are variables of type `T`, then `t1 = t2;` should have the expected effect. Normally this is no problem, but if take care when you use `malloc` or `new`.

To make it easy to work with a reray of a certain type, say a reray of doubles, add the following line:

```
typedef reray<double> doublereray;
```

Chapter 3

General Types

This part describes the types that play a role in PlaGeo but that are not limited to planar geometry. First the type for coordinates is described. This type will also be used for higher dimensional geometry. Second a general container type is introduced which is needed in PlaGeo but can be very useful for other applications as well.

3.1 Base Type

First of all a choice was made how coordinates are represented. The type for this is `s_coord`, which stands for single coordinate. In the header file there is a line

```
typedef float s_coord;
```

but the user should and need not use this information. Actually, if this single declaration is changed to

```
typedef double s_coord;
```

and everything is recompiled, all should work fine.[1]

When the library is installed, this choice may have been made, thus offering some space and execution speed for higher precision.

The rest of this section describes how this was achieved and can be skipped if one is not interested in the internals.

The overloading mechanism of C++ was used to make it easy to choose between the low or high precision variant. For example, there are two functions in the math library to compute the square root, one for doubles and one for floats. Their signatures are:

```
double sqrt(double x);
float sqrtf(float x);
```

In order to choose the right one automatically, a new header file (math++.h) was created. In this file new functions are defined for all mathematical functions that have a float and a double version. The names are made by appending the suffix `_pp` to the name of the function. Now

[1] Everything means here everything that depends on this header file: not only the user files but also the library itself.

2.4 How classes are described

In the sequel all classes and related routines will be described. The description of a class is composed of a few sections. Not all sections need be present in all class descriptions.

First there is a formal description of the class interface. This is a condensed form of the description in the header file (plageo.h), from which irrelevant information has been removed. Then there is a short description in words what concept the class represents.

Next the class functions are described, first the constructors, then the normal functions. In this section the variable `c` always stands for a variable of the current class.

In the next section functions are described that do not belong to the class, but are related to it. Those functions do not follow the dot syntax, but are called like a normal C function. After this may come some remarks, for example about the efficiency of the operations, and a section with examples.

The description of routines as given in this manual may differ a bit from the actual definitions, as they are given in the header files. Especially, a function parameter or result may be described here as being a `T`, while it is in fact a `const T &` (a const reference parameter). There are several reasons for this:

- the semantics of the two alternatives is almost the same
- the format with the const reference is harder to read, especially for inexperienced C++ users.
- it is mainly a matter of performance which of the two versions is chosen. The convention that we use leaves the freedom to change the interface a bit without having to revise the documentation.

2.5 Changes

There is a base type (`pl_object`) of which most geometric types are now derived. This object defines a common interface for all those geometric objects (so, for instance, they can all be transformed in the same way).

There are more transformations defined, all derived from the new base type `pl_transform`.

Rerays have some new features.

The treatment of vectors and points has changed considerably. The type `pl_vertex` has disappeared, the types `pl_point` and `pl_fixedvec` have been introduced. Previous programs will most likely work correctly if the two following lines are in the source code:

```
typedef pl_vec pl_vertex;
typedef pl_vecreray pl_vertexreray;
```

Some routines have been renamed:

old name	new name
supportingline	supporting_line
approximation	approximation_d
project	placeproject

```
void operator*=(pl_vec &vec, s_coord l)
```

Here the infix operator `*=` is defined for a vector and a length. This operator can be used in either of the two following ways to make a vector three times as long:

```
vec *= 3;
operator*=(vec, 3);
```

The precedence of operators cannot be changed, neither can new operators be defined.

The declaration of this operator contains one strange feature: the ampersand in the parameter list. This is another extension of C++, a call by reference mechanism. The parameter `vec` behaves the same as a var-parameter in Pascal. As can be seen in the example above, the operator is called with `vec`, not with `&vec`.

Call by reference is used a lot in the Geo libraries. In fact, in the edge class, the actual declaration of transform is:

```
void        transform(const pl_vec &displacement)
```

so the vector is passed by reference. The keyword `const` says that the function will not change this parameter. The semantics are therefore the same as call by value. The reason for passing a lot of parameters by const reference is efficiency; usually it is cheaper to pass a reference than to copy an object.

The use of malloc and free in C++ is not advisable. Instead there are the language constructs new and delete. There are two ways of using those constructs, depending on whether a single object is created or several. Here follows an example of the dynamic creation of line segments:

```
pl_edge *edge_pt;
edge_pt = new pl_edge;
delete edge_pt;
edge_pt = new pl_edge[11];
delete [] edge_pt;
```

Which can be compared with the C code:

```
pl_edge *edge_pt;
edge_pt = (pl_edge *) malloc(sizeof(pl_edge));
free(edge_pt);
edge_pt = (pl_edge *) malloc(11*sizeof(pl_edge));
free(edge_pt);
```

As can be seen, the syntax of `new` is somewhat simpler than the syntax of malloc. More important though is that `new` always calls a constructor for every object that is created, so that every object is initialized in a valid state. This prevents strange errors that might happen otherwise.

The two different forms of `delete` sometimes cause confusion. Remember: when square brackets are used with new, square brackets should be used with delete.[4]

[4] Except for freeing the space used by the object, delete also calls destructors for all objects. Destructors play a role when objects cease to exist. Because this happens transparently to the user, they are not treated here.

Except for the two explicitly mentioned constructors, a third one can be used:

```
pl_edge e3(e2);
```

This constructor initializes `e3` as a copy of `e2`. In C++ such a copy constructor exists for each class, that is why it is not mentioned explicitly in the class descriptions.

After the declaration of an object, the class functions for that object can be called. A class function is called for an object by placing the function name behind the object name, separated by a dot. For instance:

```
l = e2.length();
vt3 = e2.begin_vt();
e2.transform(vec);
```

Here the length of `e2` is put in `l`, the begin vertex in `vt3`. After that `e2` is translated over a vector `vec`.

Now let's look more closely to the declarations of the class functions. The function `begin_vt` in line 3 returns a `pl_vec` and takes no arguments. The keyword `const` after the declaration means that the object (e2 in the example) does not change because of this call.[3]

The function `transform` takes a `pl_vec` as parameter. This function translates the edge over a vector. Note that the declaration is not followed by `const`, because the edge is altered.

Objects of a class can be assigned to variables of that class. Because this is always possible, this is not mentioned for every individual class.

```
e1 = e2;
e2 = pl_edge(vt4, vt5);
```

In the second line a constructor is used in the same way as a constant of a standard type (compare: `i2 = 42;`). In fact, a temporary object of type `pl_edge` is created and assigned to `e2`. The syntax for this use of a constructor differs from the syntax used for declaration; in the former case the arguments follow the class name, in the latter they follow the variable name.

What apparently is missing, is a description of how the data are represented internally. This is hidden from the user, thus making it impossible to directly inspect or alter the data. This leaves the implementor the freedom to choose for a different implementation, leaving the interface the same. For example, an edge might be represented by its begin- and endpoint or alternatively by a begin-point, a direction and a length.

2.3 Other aspects of C++

Apart from the class concept C++ has several other features. In C two functions having the same name is not allowed. C++ permits this, if the parameter lists differ. This is called overloading. For example, in this library the function `clearance`, which gives the smallest distance between two geometric objects, is defined for all kinds of pairs of objects.

Except functions, also operators may be defined. This is a special case of overloading. There is, for instance, an operator:

[3] Actually, the object may change internally, but this is invisible when using only the public interface to the class. This allows for caching things for more complicated objects.

2.2 The Use of C++ Classes

The most important feature that is added in C++ is the *class*. A class can be seen as a new, user defined type. As such the class resembles a struct. An important difference is that a class, apart from being a repository for data, may also contain a number of operations on those data. Those operations are called the *class functions* in the sequel.

A class can hide some data and operations from the users of the class, thus making it possible to separate implementation and interface. Another aspect of classes is inheritance. In this case a class (called the superclass) is built on top of other classes (the subclasses) and inherits data and operations from the subclasses. This means that everywhere where a subclass is expected, a superclass may be provided instead. Inheritance is not much used in the Geo libraries.

Example of a class

To make it somewhat clearer how classes are described and how they can be used, we give a description of the edge class here. Somewhat simplified, the class can be described as follows:

```
class pl_edge {
                pl_edge()                                       (1)
                pl_edge(pl_vec vt1, pl_vec vt2)                 (2)
    pl_vec      begin_vt() const                                (3)
    pl_vec      end_vt() const                                  (4)
    s_coord     length() const                                  (5)
    void        transform(pl_vec displacement)                  (6)
};
```

The class is named `pl_edge` and is used to represent line segments. Almost all classes in PlaGeo have the prefix `pl_`. On the first two lines there are two class functions with the same name as the class. Those special functions are called *constructors* and play a part in the declaration of variables of this class. The next four lines contain normal class functions. The types `pl_vec` (vector) and `s_coord` (length unit) are defined elsewhere. In the following examples, the names `vt`, `vec` and `l` will be used for variables of those types.

Constructors give the possibility to initialize an object during declaration. In this case there are two (explicitly named) constructors. This gives the following possibilities to declare a variable of type `pl_edge`:

```
pl_edge e1;
pl_edge e2(vt1, vt2);
```

In the first declaration the first constructor is used. There are no parameters; the line segment will be initialized in a default way. In the second declaration two vertices are given as parameter: the begin and endpoint of the edge. Note that when no parameters are required, no parentheses should be used. Those two ways of declaring can be compared to:

```
int i1;
int i2 = 42;
```

Chapter 2

Use

The Geo libraries are implemented in C++. The reason for this choice is that C++ supports the implementation of abstract data types in a better way than C.[1] A disadvantage is that C++ is more complicated than C. The user of the Geo libraries needs no thorough knowledge of C++: in the sequel of this section the necessary basics are explained. Because C is a subset of C++, the user can write an ordinary C-program, using only some C++-syntax for calling library routines. This program can be translated by the C++ compiler. This documentation supposes the reader has some knowledge of C, not of C++.

2.1 Compilation

Programs using the Geo-libraries should be compiled with a C++ compiler that supports templates. Suppose that a source file `file.C` uses PlaGeo-routines. In that case `file.C` should contain the line

```
#include <plageo.h>
```

This include-file can be found in the directory `/usr/staff/include`. The library itself resides in `/usr/staff/lib`.[2] Compiling and linking should be done as follows:

```
CC -I/usr/staff/include -c file.C
CC file.o -I/usr/staff/include -L/usr/staff/lib -lplageo -lm
```

For the use of SpaGeo use the following:

```
#include <spageo.h>

CC -I/usr/staff/include -c file.C
CC file.o -I/usr/staff/include -L/usr/staff/lib -lspageo -lplageo -lm
```

Note that the SpaGeo library itself makes use of PlaGeo. This library should therefore always be linked.

[1]The reasons for considering only C and C++ were availability and familiarity.
[2]The location of the files describes the situation at Utrecht University.

The Geo-libraries are in no way complete. Routines are normally only written when there is a need for them. There are for example no circles or spheres. It is, however, not difficult to add new objects and operations. Because the objects can be accessed on a low level, it is easy for instance to write a routine that, given a new type for circles, computes the distance between a line segment and a circle.

An important point is, finally, the underlying computation model. PlaGeo and SpaGeo compute with floating point numbers. This means that operations are not exact. The Geo-libraries have the philosophy: we do not compute with infinite precision, so for questions where this is needed we have no answer. As a consequence there are no routines to decide if a point lies on a line or if two lines coincide. In section 7.2 an example is given how, using the low level functions, routines can be written to approximate an answer for such questions.

As was mentioned before, the implementation of geometric algorithms is often a lot easier when rounding errors need not be considered. Therefore it might be interesting to build libraries like these based on exact arithmetic.

Bugs, Extensions, etc.

Although PlaGeo has gone through extensive testing some bugs may still be present. Your comments would be greatly appreciated. Please report any bug reports or comments to me using the email address **geert@cs.ruu.nl**. Also, if you make additions to PlaGeo that could be of more general interest, please mail them to me so that I might add them to a future release.

The library can be ftp-ed from **archive.cs.ruu.nl** in directory pub/SGI/GEO.

Notice

Acknowledgements

The work on this package was partially supported by the Dutch Organization for Scientific Research (N.W.O) and by the ESPRIT III BRA Project 6546 (PROMotion).

Chapter 1

Introduction

When implementing algorithms from computational geometry, some basic questions have to be attacked:

- how are geometric objects such as points, lines and polygons represented.
- how to compute intersections and distances.

A surprisingly large amount of the time spent on a project is often devoted to this kind of details.

The two libraries PlaGeo (planar geometry) and SpaGeo (spatial geometry) have been developed to provide assistance in this field. As the names suggest, the former deals with two-dimensional space, the latter with three-dimensional space.

Those libraries contain several things. First of all they provide types for all kinds of geometric objects and routines to create those objects. Furthermore there are routines to inspect those objects to answer questions like 'What are the coordinates of this point', 'What are the endpoints of this edge' and 'How many sides has this polygon'. On a higher level there are routines to decide if two objects intersect or what the distance between two objects is.

There are a lot of advantages of using those libraries over developing your own routines. We mention:

ease of use No time needs to be spent on design, implementation and debugging of basic operations.

less errors The routines have been used before. The probability of finding bugs is therefore reduced. Bugs can be very subtle. For instance, algorithms can be sensible to rounding errors in floating point computations. Such algorithms, that almost always but not always lead to the correct result, make debugging a very painful business.

efficiency Most routines in the library have been optimized. For example bounding box tests are used whenever possible. The routines are general purpose, though. It is very well possible that in specific cases things can be done more efficiently. In those cases it should be possible to write a special routine, using the low level functions.

common base When comparing different algorithms for the same problem, it is only fair to let them use the same base operations.

Part I

Intro

Contents

PlaGeo

A Library for Planar Geometry

version 2.0

and

SpaGeo

A Library for Spatial Geometry

version 2.0

Geert-Jan Giezeman

January 21, 1994