User's Guide to the SOLID Collision Detection Library

last updated 22 July 2003
for version 3.5

Gino van den Bergen

SOLID 3 uses Qhull from The Geometry Center of the University of Minnesota. Qhull is copyrighted as noted below. Qhull is free software and may be obtained via anonymous ftp from geom.umn.edu.

The National Science and Technology Research Center for
Computation and Visualization of Geometric Structures
(The Geometry Center)
University of Minnesota
400 Lind Hall
207 Church Street S.E.
Minneapolis, MN 55455 USA

email: qhull@geom.umn.edu

1 License

This open-source edition of SOLID version 3 is released under the terms of either the GNU General Public License (GPL) or the Q Public License (QPL). This means that for software created with SOLID version 3 you must comply with the terms of either one of these licenses. See appendices for a complete list of terms and conditions of these licenses

For enquiries about commercial use of SOLID, please contact info@dtecta.com.

2 Introduction

SOLID is a software library containing functions for performing intersection tests and proximity queries that are useful in the context of collision detection. Collision detection is the process of detecting pairs of geometric objects that are intersecting or are within a given proximity of each other. In particular, SOLID is useful for detecting collisions between objects that are moving relatively of each other over time. The motions of objects are controlled by the client application, and are not determined or affected by SOLID.

Furthermore, SOLID provides functionality for determining geometric data pertaining to a pair of objects that is used by the client application for computing the appropriate response to a collision. This data, referred to as response data, is passed to the client application by means of a callback mechanism or by direct queries from the client application.

2.1 Overview

SOLID's functionality is subdivided into the following categories:

1. Shape definition
2. Object placement and motion
3. Scene management
4. Response definition
5. Global actions
6. Broad phase

2.1.1 Shape Definition

The shape of a geometric object is defined relative to its local coordinate system. A shape can be a simple geometric primitive, e.g., a sphere or a line segment, or a complex shape composed of multiple primitives. Shapes defined in terms of vertex positions, e.g., a triangle mesh, may use vertex data that is stored in memory maintained by the client application.

2.1.2 Object Placement and Motion

A geometric object is defined by a shape, an affine transformation, and a margin. The shape is placed in the world coordinate system by specifying the object's local coordinate system represented by an affine transformation. The actual object is the set of points in world coordinates whose distance to the shape is at most the margin. Motion of an object can be defined by changing the placement of the local coordinate system, the margin, or the positions of the vertices of the shape.

2.1.3 Scene Management

Collections of objects on which collision detection has to be performed are placed in a scene. SOLID is capable of maintaining multiple scenes. Objects can be shared by multiple scenes. A scene maintains cached data pertaining to the objects in the scene in order to speed-up consecutive collision queries that are performed on the scene.

2.1.4 Response Definition

In SOLID, collision response is handled by callback functions. The types of response and the callback functions that needs to be executed for each pair of intersecting objects is stored in a response table. Currently, there are three types of response:

1. Simple response: no response data is returned.
2. Witnessed response: a common point of the intersecting objects is returned.
3. Depth response: the penetration depth of the intersecting objects is returned. The penetration depth is the shortest vector over which one of the objects needs to be translated in order to bring the objects into touching contact.

Response tables can be shared by multiple scenes, and per scene multiple response tables can be used. To each object that is considered for collision detection in a response table, a response class is assigned. Responses can be defined per pair of response classes, on all pairs containing a specific response class, or as a default for all pairs of response classes. On a pair of response classes multiple responses can be defined. During the lifetime of an object, it is possible to reassign the response class associated with the object per response table.

2.1.5 Global Actions

The main functionality of SOLID is to perform collision tests. Given a scene and a response table, a collision test computes for all pairs of colliding objects on which a response is defined the required response data and passes these data together with the colliding pair to the callback. The response actions are defined by the client application in the callback and fall outside the scope of SOLID.

The client application may also obtain response data for a given object or pair of objects by direct queries. These direct queries are useful for static interference checks or for tracking the closest points of a pair of objects.

2.1.6 Broad Phase

The broad phase detects changes in the overlap status of pairs of axis-aligned boxes. The overlap status of a pair of boxes changes whenever the boxes start or cease to overlap. The boxes are organized in scenes similar to the scenes for objects. For each pair of boxes in a scene whose overlap status is changed a callback is called. The client defines a callback for pairs that start to overlap and one for pairs that cease to overlap. The broad phase is actually a sub-layer of the SOLID library, however the API can be directly accessed by the client application.

2.2 Software Package

Currently, the SOLID package consists of three separate layers:

1. MT: The Mathematics Toolkit. This is a set of C++ classes containing abstract data types for scalars, vectors, points, quaternions, matrices, coordinate systems, and bounding boxes. Global names in this layer are prefixed with "MT_".

2. The broad phase: A set of C++ classes wrapped by a C API. The broad phase detects changes in the overlap status of pairs of axis-aligned boxes (pairs of boxes that start or cease to overlap). Global names in this layer are prefixed with "BP_".
3. The narrow phase: A set of C++ classes wrapped by a C API. The narrow phase performs exact collision tests for pairs of objects, and computes response data for the colliding pairs of objects. Global names in this layer are prefixed with "DT_".

2.3 New Features of SOLID version 3

Since the previous version 2.0 of SOLID, which was released in June 1998 under the terms of the GNU Library General Public License by the Department of Mathematics & Computing Science of Eindhoven University of Technology, SOLID has evolved and matured considerably. New features as well as improvements for robustness and performance have been added. The most important changes in SOLID 3 are:

- Use of single-precision floating-point numbers. The use of a 32-bit floating-point format is a requirement for games and other interactive applications that run on current PC graphics hardware and consoles.
- Penetration depth computation as a new response type. The penetration depth of a pair of intersecting objects is the shortest vector over which one of the objects needs to be translated in order to bring the objects in touching contact. The penetration depth can be used as an approximation of the contact point and contact plane, which are necessary for physics-based simulation. The depth response replaces the smart response in SOLID 2.0. Smart response uses the configuration of the previous time frame for finding the contact points and plane, which could give bad results when objects were interpenetrating over a number of frames.
- SOLID 3 maintains multiple scenes. This feature is useful when collision detection is required for multiple tasks. For instance, it is possible to maintain at the same time a sound scene, a scene used for visibility culling, and a scene for physics simulations, without the objects in different scenes interfering with each other.
- Stridden vertex arrays can be used for defining complex shapes.
- Objects can be expanded spherically, i.e., a margin that defines the radius of the sphere that is 'added' can be set for each object. The object is the set of points whose distance to the shape is at most the margin. Margins are useful for creating objects with rounded edges, or 'sensitive' areas around an object.
- Functions for direct computation of response data have been added to the API. This is useful, since it allows the client to check overlap status, distance, etc., without having to perform a global collision test.
- A ray cast has been added to SOLID. The ray cast returns the object in a scene that the ray hits first. Also, the hit spot and surface normal to this object are returned.
- Response callbacks are defined in a response table independent of a scene.
- Response callbacks are defined per pair of response classes rather than per object pair. With each object for which a response is defined in a response table, a response class is associated. The response class of an object may change over time.

- It is now possible to define multiple response callbacks per object pair. This is useful for performing several actions for a single collision (play sound, apply impulse, update statistics).

3 Installing the SOLID SDK

3.1 Requirements

The SOLID library (libsolid) and the broad-phase library (libbroad) have a standard C API and can be linked to both C and C++ applications. Note that libsolid and libbroad are internally coded in C++ and thus need the libstdc++ library on Unix platforms (link using g++ rather than gcc). The mathematics toolkit (MT) is coded in standard C++ and uses templates extensively. The source code compiles under GNU g++ version 2.95 and higher and Microsoft Visual C++ version 6.0 SP4 and higher.

3.2 Installation

All header files that export types and functions are stored in the '`include`' directory. Source files are stored in the '`src`' directory. On a Unix-like operating system, a '`configure`' script generated by the GNU build tools (automake, autoconf, libtool) is used for creating the Makefiles. Simply type '`./configure`' followed by '`make`' in the root of the SOLID distribution directory to build the SOLID lib and example programs. Typing '`make install`' installs the SOLID library header files and binaries as well as this documentation in '`/usr/local`'. Note that you usually must have root privileges for adding items to '`/usr/local`'. See '`./configure --help`' for installing the library in a different location.

The SOLID library can be built under Microsoft Visual C++ 6.0 and higher using the workspace (.dsw) and project (.dsp) files found in the '`VisualC6`' directory. For use with Visual C++ 7.0 and higher, the workspace and project files must first be converted to the newer solution (.sln) and project (.vcproj) formats. On a Win32 platform, the simplest way to make the SOLID SDK accessible in your applications is to add the SOLID '`include`' and '`lib`' directories to respectively the include and link path used by the compiler. For running executables built using SOLID, the '`solid.dll`' should be contained in the executable search path. This is achieved most easily by copying these DLLs either to the directory containing the executable or to the '`WINxxxx/system32`' directory.

SOLID uses the Qhull library for computing convex hulls of sets of points. In case the Qhull library is not available for your platform, SOLID can still be built. However, in that case, queries on convex hulls built with `DT_NewPolytope` (see below) fall back to brute force algorithms, and are therefore much slower. In order to build a SOLID library without Qhull, make sure that the preprocessing flag '`-DQHULL`' is not set.

The core of SOLID may use either single or double precision floating-point arithmetic. The default option is single-precision. In order to build a double-precision SOLID core, use '`configure`' with the '`--enable-doubles`' option. Under Visual C++, doubles can be enabled using the '`-DUSE_DOUBLES`' preprocessing flag. Note that this flag only affects the floating-point numbers that are used internally in SOLID. The API functions always use single-precision floating-point numbers.

In the same way, the SOLID core can be built using a tracer class for scalars. See the file '`MT_ScalarTracer.h`' in the '`include`' directory. A scalar tracer is used for tracing rounding errors in results of floating-point operations. An object of the type `MT_ScalarTracer` has a value field and an error field. The value field holds the result of an operation, and

the error field multiplied by the machine epsilon gives an estimated upper bound for the relative rounding error. The `MT_ScalarTracer` class behaves as the primitive scalar types `float` and `double`, however, constants of this type have to be constructed explicitly. In order to build a SOLID core that uses scalar tracers, create Makefiles using '`configure`' with the '`--enable-tracer`' option. Under Visual C++, the tracers are enabled using the '`-DUSE_TRACER`' preprocessing flag. This option is very useful for debugging purposes. Makefiles for building debug binaries of SOLILD are created using the '`--enable-debug`' option.

4 The SOLID API

The SOLID API is a set of C functions. All API functions, also referred to as commands, use arguments that have primitive types, such as, integers, floats, and arrays of floats, or handles (type-mangled pointers) to internal objects of SOLID. The types `DT_Scalar`, `DT_Vector` and `DT_Quaternion` are simply typedefs:

```
typedef float     DT_Scalar;
typedef DT_Scalar DT_Vector3[3];
typedef DT_Scalar DT_Quaternion[4];
```

The MT C++ classes can be used for representing geometric data such as vectors and quaternions as they are implicitly casted to arrays of floats, however the use of these classes is not required for calling SOLID functions. SOLID API functions can be called using your own or third-party 3D geometry objects if you stick with the following rules:

- All used scalar types are of the type `float`.
- Quaternions store their imaginary vector part before the real scalar part. Thus, for an array `float q[4]` that represents a quaternion, `q[3]` must be the scalar part.
- Transformations are specified using arrays of 16 floating-point numbers representing a 4x4 column-major matrix as used in OpenGL. This matrix representation is discussed below.

4.1 Building Shapes

The commands for creating and destroying shapes are

```
DT_ShapeHandle DT_NewBox(DT_Scalar x, DT_Scalar y, DT_Scalar z);
DT_ShapeHandle DT_NewCone(DT_Scalar radius, DT_Scalar height);
DT_ShapeHandle DT_NewCylinder(DT_Scalar radius, DT_Scalar height);
DT_ShapeHandle DT_NewSphere(DT_Scalar radius);
DT_ShapeHandle DT_NewPoint(const DT_Vector3 point);
DT_ShapeHandle DT_NewLineSegment(const DT_Vector3 source,
                                 const DT_Vector3 target);

DT_ShapeHandle DT_NewMinkowski(DT_ShapeHandle shape1,
                               DT_ShapeHandle shape2);
DT_ShapeHandle DT_NewHull(DT_ShapeHandle shape1,
                          DT_ShapeHandle shape2);

void           DT_DeleteShape(DT_ShapeHandle shape);
```

Shapes are referred to by values of `DT_ShapeHandle`. The command `DT_NewBox` creates a rectangular parallelepiped centered at the origin and aligned with the axes of the shape's local coordinate system. The parameters specify its extent along the respective coordinate axes. The commands `DT_NewCone` and `DT_NewCylinder` create respectively a cone and a cylinder centered at the origin and whose central axis is aligned with the y-axis of the local

coordinate system. The cone's apex is at $y = height/2$. The command `DT_NewSphere` creates a sphere centered at the origin of the local coordinate system. The command `DT_NewPoint` creates a single point. The command `DT_NewLineSegment` creates a single line segment in a similar way.

Any pair of convex shapes (including general polytopes) can be combined to form compound shapes using the commands `DT_NewMinkowski` and `DT_NewHull`. The command `DT_NewMinkowski` 'adds' the two shapes by sweeping one shape along the other. For instance, the Minkowski addition of a sphere and a line segment creates a hot dog. The command `DT_NewHull` creates a shape that represents the exact convex hull of the two shapes.

Complex shape types composed of simple polytopes (polytope soups) are created using the `DT_NewComplexShape` command. Here, a simple polytope is a d-dimensional polytopes, where d is at most 3. A simple d-polytope can be a simplex (point, line segment, triangle, tetrahedron), a convex polygon, or a convex polyhedron.

There are no topological constraints on the set of vertices of a polytope. In particular, the vertices of a polytope need not be affinely independent, and need not be extreme vertices of the convex hull. However, convex polytopes with many vertices may deteriorate the performance. Such complex polytopes should be created using the `DT_NewPolytope` command. Make sure that in that case, SOLID is built using Qhull.

For constructing complex shapes the following commands are used:

```
DT_VertexBaseHandle DT_NewVertexBase(const void *pointer,
                                     DT_Size stride);
void DT_DeleteVertexBase(DT_VertexBaseHandle vertexBase);
void DT_ChangeVertexBase(DT_VertexBaseHandle vertexBase,
                         const void *pointer);

DT_ShapeHandle DT_NewComplexShape(DT_VertexBaseHandle vertexBase);
void           DT_EndComplexShape();

DT_ShapeHandle DT_NewPolytope(DT_VertexBaseHandle vertexBase);
void           DT_EndPolytope();

void DT_Begin();
void DT_End();

void DT_Vertex(const DT_Vector3 vertex);
void DT_VertexIndex(DT_Index index);

void DT_VertexIndices(DT_Count count, const DT_Index *indices);
void DT_VertexRange(DT_Index first, DT_Count count);
```

A d-polytope is specified by enumerating its vertices. This can be done in two ways. In the first way, the vertices are specified by value, using the `DT_Vertex` command. The following example shows how the faces of a pyramid are specified.

```
DT_Vector3 float verts[] = {
   {  1.0f,  0.0f,  1.0f },
   {  1.0f,  0.0f, -1.0f },
```

```
    { -1.0f,  0.0f, -1.0f },
    { -1.0f,  0.0f,  1.0f },
    {  0.0f, 1.27f,  0.0f }
};

DT_ShapeHandle pyramid = DT_NewComplexShape(NULL);

DT_Begin();
DT_Vertex(verts[0]);
DT_Vertex(verts[1]);
DT_Vertex(verts[2]);
DT_Vertex(verts[3]);
DT_End();

DT_Begin();
DT_Vertex(verts[0]);
DT_Vertex(verts[1]);
DT_Vertex(verts[4]);
DT_End();

...

DT_EndComplexShape();
```

Here, an argument of `NULL` in `DT_NewComplexShape` denotes that the complex shape does not use an external vertex array.

In the second method, the vertices are referred to by indices. For each complex shape, we specify a single array of vertices. Vertex arrays are maintained by the client application and can be accessed directly by SOLID. Vertex arrays are accessed via vertex bases. De command `DT_NewVertexBase` creates a vertex base for the array given by the argument `pointer`. The client must maintain vertex data in single-precision floating-point format. The client is free to store vertex data using arbitrary spacing in-between the individual array items. The spacing is specified using the `DT_Size stride` parameter. For instance, the client maintains an array of vertices of the type:

```
struct Vertex {
    float xyz[3];
    float uv[2];
    float normal[3];
};

struct Vertex verts[NUM_VERTICES];
```

When specifying a complex shape you can use this data as follows

```
DT_VertexBaseHandle base = DT_NewVertexBase(verts[0].xyz,
                                            sizeof(struct Vertex));
```

A stride of zero denotes that the vertex coordinate data is packed in a separate array, thus

```
DT_Vector3 verts[NUM_VERTICES];

DT_VertexBaseHandle base = DT_NewVertexBase(verts[0], 0);
```

Each time the vertices are updated, or a new vertex base is assigned, to a complex shape, for instance, when using a deformable triangle mesh, the client needs to notify SOLID of a changed vertex array by calling `DT_ChangeVertexBase`. We discuss the use of this command further on.

The handle to the vertex base is passed as argument to `DT_NewComplexShape`. The command `DT_VertexIndex` is used for specifying vertices. See the following example:

```
DT_ShapeHandle pyramid = DT_NewComplexShape(base);

DT_Begin();
DT_VertexIndex(0);
DT_VertexIndex(1);
DT_VertexIndex(2);
DT_VertexIndex(3);
DT_End();

DT_Begin();
DT_VertexIndex(0);
DT_VertexIndex(1);
DT_VertexIndex(4);
DT_End();

...

DT_EndComplexShape();
```

Alternatively, the indices can be placed into an array and specified using the command `DT_VertexIndices`, as in the following example:

```
DT_Index face0[4] = { 0, 1, 2, 3 };
DT_Index face1[3] = { 0, 1, 4 };

...

DT_VertexIndices(4, face0);
DT_VertexIndices(3, face1);
```

Finally, a polytope can be specified from a range of vertices using the command `DT_VertexRange`. The range is specified by the first index and the number of vertices. In the

following example a pyramid is constructed as a convex polyhedron, which is the convex hull of the vertices in the array.

```
DT_ShapeHandle pyramid = DT_NewComplexShape(base);
DT_VertexRange(0, 5);
DT_EndComplexShape();
```

The same shape can be built using the `DT_NewPolytope` command:

```
DT_ShapeHandle pyramid = DT_NewPolytope(base);
DT_VertexRange(0, 5);
DT_EndPolytope();
```

Note that within a `DT_NewPolytope` construction all the vertex array commands can be used to specify vertices. The commands `DT_Begin` and `DT_End` are ignored for polytopes. Convex polytopes constructed using the `DT_NewPolytope` command are preprocessed by SOLID in order to allow for faster testing, and should be used when the number of vertices is large.

4.2 Creating and Moving Objects

An object is an instance of a shape. The commands for creating, moving and deleting objects are

```
DT_ObjectHandle DT_CreateObject(void *client_object, DT_ShapeHandle shape);
void            DT_DestroyObject(DT_ObjectHandle object);

void DT_SetPosition(DT_ObjectHandle object, const DT_Vector3 position);
void DT_SetOrientation(DT_ObjectHandle object,
                       const DT_Quaternion orientation);
void DT_SetScaling(DT_ObjectHandle object, const DT_Vector3 scaling);

void DT_SetMatrixf(DT_ObjectHandle object, const float *m);
void DT_SetMatrixd(DT_ObjectHandle object, const double *m);

void DT_SetMargin(DT_ObjectHandle object, DT_Scalar margin);
```

An object is referred to by a `DT_ObjectHandle`. The first parameter `void *client_object` is a pointer to an arbitrary structure in the client application. This pointer is passed as parameter to the callback function in case of a collision, and can be used for collision handling. In general, a pointer to a structure in the client application associated with the collision object should be used.

An object's motion is specified by changing the placement of the local coordinate system of the shape. Initially, the local coordinate system of an object coincides with the world coordinate system.

The placement of an object is changed, either by setting the position, orientation, and scaling, or by using an OpenGL 4x4 column-major matrix representing an affine transformation. Placements are specified relative to the world coordinate system. Rotations are specified using quaternions. The object's local coordinate system can be scaled non-uniformly by specifying a scale factor per coordinate axis. Following example shows how a pair of objects are given absolute placements.

```
DT_ObjectHandle objectHandle = DT_CreateObject(&myObject, shapeHandle);

float position = { 0.0f, 1.0f, 1.0f };
float orientation = { 0.0f, 0.0f, 0.0f, 0.1f };
float scaling = { 1.0f, 2.0f, 1.0f };

DT_SetPosition(objectHandle, position);
DT_SetOrientation(objectHandle, orientation);
DT_SetScaling(objectHandle, scaling);
```

For scalings along axes that are not coordinate axes, such as shears, you should construct a 4x4 column-major matrix representation of the local coordinate system and use `DT_SetMatrix` to specify the object placement.

The x-axis of the local coordinate system relative to the world coordinate system is the vector (m[0], m[1], m[2]), the y-axis is (m[4], m[5], m[6]), the z-axis is (m[8], m[9], m[10]), and the local origin is (m[12], m[13], m[14]). The elements m[3], m[7], m[11], and m[15] are ignored. These values are assumed to be 0, 0, 0, and 1, respectively. Thus, only affine transformations are allowed.

By setting a positive margin using `DT_SetMargin` you can spherically expand an object. The actual collision object is the set of points whose distance to the transformed shape is at most the margin. For instance, a hot dog or capsule can be created using

```
DT_Vector3 source = { 0.0f, 0.0f, 0.0f }
DT_Vector3 target = { 0.0f, -1.5f, 0.0f }

DT_ShapeHandle line = DT_NewLineSegment(source, target);

DT_ObjectHandle object = DT_CreateObject(&myHotDog, line);
DT_SetMargin(object, 0.3f);
```

This object is useful for navigating along walls and over terrains Positions, orientations, scalings, and margins may all be changed during the life time of an object.

4.2.1 Who's Afraid of Quaternions?

A quaternion is a four-dimensional vector. The set of quaternions of length one (points on a four-dimensional sphere) map to the set of orientations in three-dimensional space. Since in many applications an orientation defined by either a rotation axis and angle or by a triple of Euler angles is more convenient, the package includes code for quaternion operations. The code is found in the mathematics toolkit (MT).

The quaternion class is located in the file 'MT_Quaternion.h'. The class has constructors and methods for setting a quaternion. For example

```
MT_Quaternion q1(axis, angle);
MT_Quaternion q2(yaw, pitch, roll);

...

q1.setRotation(axis, angle);
q2.setEuler(yaw, pitch, roll);

...

DT_SetOrientation(objectHandle, q1);
```

Also included is a static method `MT_Quaternion::random()`, which returns a random orientation.

4.2.2 Proximity Queries

Objects can also be queried directly using the commands

```
DT_Scalar DT_GetClosestPair(DT_ObjectHandle object1, DT_ObjectHandle object2,
                            DT_Vector3 point1, DT_Vector3 point2);

DT_Bool   DT_GetCommonPoint(DT_ObjectHandle object1, DT_ObjectHandle object2,
                            DT_Vector3 point);

DT_Bool   DT_GetPenDepth(DT_ObjectHandle object1, DT_ObjectHandle object2,
                         DT_Vector3 point1, DT_Vector3 point2);
```

The command `DT_GetClosestPair` returns the distance between `object1` and `object2`, and a pair of closest points `point1` and `point2` given in world coordinates The command `DT_GetCommonPoint` returns a boolean that denotes whether the objects `object1` and `object2` intersect, and, in case of an intersection, returns a common point `point` in world coordinates. The command `DT_GetPenDepth` also returns a boolean that denotes whether the objects `object1` and `object2` intersect, and, in case of an intersection, returns a pair of witness points of the penetration depth `point1` and `point2` in world coordinates.

The maximum relative error in the closest points and penetration depth computation can be set using

```
void DT_SetAccuracy(DT_Scalar max_error);
```

The default for `max_error` is 1.0e-3. Larger errors result in better performance. Non-positive error tolerances are ignored.

The maximum tolerance on relative errors due to rounding is set using

```
void DT_SetTolerance(DT_Scalar tol_error);
```

This value is the estimated relative rounding error in complex computations and is used for determining whether a floating-point number should be regarded as zero or not. The default value for 'tol_error' is the machine epsilon, which is `FLT_EPSILON` when floats are used, and `DBL_EPSILON` when double-precision floating-point numbers are used internally. Very large tolerances result in false collisions. Setting tol_error too small results in missed collisions. Non-positive error tolerances are ignored.

Furthermore, objects can be queried to return data maintained internally. The world-axes aligned bounding box of an object is returned using

```
void DT_GetBBox(DT_ObjectHandle object, DT_Vector3 min, DT_Vector3 max);
```

Here, `min` and `max` are the vertices of the box with respectively the least and greatest world coordinates. The local-to-world transformation of an object can be returned using

```
void DT_GetMatrixf(DT_ObjectHandle object, float *m);
void DT_GetMatrixd(DT_ObjectHandle object, double *m);
```

The arguments of these commands are again arrays of 16 floating-point numbers that represent a 4x4 column-major matrix as discussed above.

4.3 Scenes

For scenes with many objects the number of pairwise intersection queries can become quite large. To overcome this bottleneck, objects are maintained in scenes. The commands for construction and destroying scenes are:

```
DT_SceneHandle DT_CreateScene();
void           DT_DestroyScene(DT_SceneHandle scene);
void           DT_AddObject(DT_SceneHandle scene,
                            DT_ObjectHandle object);
void           DT_RemoveObject(DT_SceneHandle scene,
                               DT_ObjectHandle object);
```

Objects can be shared by multiple scenes. Each scene tracks the changes of placement and deformations of its objects, and updates its cached data accordingly. In this way, global collision queries using `DT_Test` (see below) can be processed much faster.

4.4 Response Handling

Collision response in SOLID is handled by means of callback functions. The callback functions have the type `DT_ResponseCallback` defined by

```
typedef DT_Bool (*DT_ResponseCallback)(void *client_data,
```

```
                                        void *client_object1,
                                        void *client_object2,
                                        const DT_CollData *coll_data);
```

Here, `client_data` is a pointer to an arbitrary structure in the client application, `client_object1` and `client_object2` are the pointers to structures in the client application specified in `DT_CreateObject`, and `coll_data` is the response data computed by SOLID. The Boolean value returned by a callback functions indicates whether further processing of callbacks is needed. If `DT_FALSE` or `DT_CONTINUE` is returned, then the remaining colliding object pairs are processed. If `DT_TRUE` or `DT_DONE` is returned, then the call to `DT_Test` is exited without further processing. We discuss the `DT_Test` further on.

Currently, there are three types of response: *simple*, *depth* and *witnessed* response. For simple response the value of `coll_data` is `NULL`. For depth and witnessed response `coll_data` points to the following structure

```
typedef struct DT_CollData {
    DT_Vector3 point1;
    DT_Vector3 point2;
    DT_Vector3 normal;
} DT_CollData;
```

An object of this type represents a pair of points of the respective objects. The points `point1` and `point2` are given relative to the world coordinate system. The `normal` field is used for depth response only.

For witnessed response, the points represent a witness of the collision. Both points are contained in the intersection of the colliding objects. Note that the witness points are not necessarily equal. For depth response, the `normal` represent the penetration depth vector. The penetration depth vector is the shortest vector over which one object needs to be translated in order to bring the two objects into touching contact. The `point1` and `point2` fields contain the witness points of the penetration depth, thus `normal = point2 - point1`.

Response callbacks are managed in *response tables*. Response tables are defined independent of the scenes in which they are used. Multiple response tables can be used in one scene, and a response table can be shared among scenes. Responses are defined on (pairs of) response classes. Each response table maintains its set of response classes. A response table is created and destroyed using the commands

```
DT_RespTableHandle DT_CreateRespTable();
void               DT_DestroyRespTable(DT_RespTableHandle respTable);
```

A response class for a response table is generated using

```
DT_ResponseClass DT_GenResponseClass(DT_RespTableHandle respTable);
```

To each object for which a response is defined in the response table a response class needs to be assigned. This is done using

```
void DT_SetResponseClass(DT_RespTableHandle respTable,
                         DT_ObjectHandle object,
                         DT_ResponseClass responseClass);
```

For each pair of objects multiple responses can be defined. A response is a callback together with its response type and client data. The `DT_ResponseType` is defined by

```
typedef enum DT_ResponseType {
    DT_NO_RESPONSE,
    DT_SIMPLE_RESPONSE,
    DT_WITNESSED_RESPONSE
    DT_DEPTH_RESPONSE,
} DT_ResponseType;
```

Responses can be defined for all pairs of response classes...

```
void DT_AddDefaultResponse(DT_RespTableHandle respTable,
                           DT_ResponseCallback response,
                           DT_ResponseType type,
                           void *client_data);

void DT_RemoveDefaultResponse(DT_RespTableHandle respTable,
                              DT_ResponseCallback response);
```

...per response class...

```
void DT_AddClassResponse(DT_RespTableHandle respTable,
                         DT_ResponseClass responseClass,
                         DT_ResponseCallback response,
                         DT_ResponseType type,
                         void *client_data);

void DT_RemoveClassResponse(DT_RespTableHandle respTable,
                            DT_ResponseClass responseClass,
                            DT_ResponseCallback response);
```

... and per pair of response classes...

```
void DT_AddPairResponse(DT_RespTableHandle respTable,
                        DT_ResponseClass responseClass1,
                        DT_ResponseClass responseClass2,
                        DT_ResponseCallback response,
                        DT_ResponseType type,
                        void *client_data);
```

```
void DT_RemovePairResponse(DT_RespTableHandle respTable,
                           DT_ResponseClass responseClass1,
                           DT_ResponseClass responseClass2,
                           DT_ResponseCallback response);
```

If for an object pair, one of the objects has a class object response defined, that needs to be overruled by a pair response, then you should remove the callback defined in the class response for the pair and add the pair response, thus

```
DT_AddClassResponse(respTable, class1, classResponse,
                    DT_SIMPLE_RESPONSE, client_data);

DT_RemovePairResponse(respTable, class1, class2, classResponse);

DT_AddPairResponse(respTable, class1, class2, pairResponse,
                   DT_DEPTH_RESPONSE, client_data);
```

In the same way, a default response can be overruled by a class or pair response.

The response callback functions are executed by calling

```
DT_Count DT_Test(DT_SceneHandle scene, DT_RespTableHandle respTable);
```

This command calls for each colliding pair of objects the corresponding callback function until all pairs are processed or until a callback function returns `DT_TRUE` or `DT_DONE`. It returns the number of object pairs for which callback functions have been executed.

Note: If the response classes of the objects in a callback differ, then `client_object1` has a 'lower' response class than `client_object2`. That is, the response class of `client_object1` is generated before the response class of `client_object2`.

4.5 Deformable Models

SOLID handles deformations of complex shapes. In this context deformations are specified by changes of vertex positions. Complex shapes that are defined using a vertex array in the client application may be deformed by changing the array elements, or specifying a new array. SOLID is notified of a change of vertices by the command

```
void DT_ChangeVertexBase(DT_VertexBaseHandle vertexBase,
                         const void *pointer);
```

Note that polytopes constructed from a vertex base using `DT_NewPolytope` are not affected by a change of vertices.

4.6 Ray Cast

NOTE: This feature is currently implemented for spheres, boxes, triangles, and triangle meshes only. Also, margins are ignored for ray casts.

The commands for performing ray casts are

```
void *DT_RayCast(DT_SceneHandle scene, void *ignore_client,
                 const DT_Vector3 source,
                 const DT_Vector3 target,
                 DT_Scalar max_param,
                 DT_Scalar *param, DT_Vector3 normal);

DT_Bool DT_ObjectRayCast(DT_ObjectHandle object,
                         const DT_Vector3 source,
                         const DT_Vector3 target,
                         DT_Scalar max_param,
                         DT_Scalar *param, DT_Vector3 normal);
```

The ray is given by `source`, `target`, and `max_param`. It represents the line segment `source` + (`target` − `source`) $* t$, where t is a member of the interval $[0, \texttt{max\_param}]$. So, if `max_param` is 1, then the ray is simply the line segment from `source` to `target`, whereas if `max_param` is equal to `FLT_MAX`, then the ray is 'infinite'.

`DT_RayCast` returns a pointer to the client object of an object in `scene` that is hit first by the ray, or `NULL` if no object is hit. `DT_ObjectRayCast` performs a ray cast on a single object and returns a Boolean indicating a hit. In case of a hit, `param` points to the t of the hit spot, and `normal` is a normal to the object's surface in world coordinates. The normal always points towards the `source`. An object can be made transparent for the ray cast by specifying the object's client object as `ignore_client`. This is useful if you need to ignore hits of the ray with the source object of the ray. For instance terrain following can be implemented by casting a ray down and setting the moving object at a distance above the spot. In this case, you are probably interested in hits with the terrain only, and do not need reports of hits with the moving object.

5 Projects and other things left to do

5.1 Coming Attractions

SOLID 4 will have the following added features:

1. Compressed AABB trees for reducing the memory footprint of triangle meshes to roughly 18 bytes per triangle.
2. A general ray cast for all shape types.
3. Shape casting: returning the first collision of a shape that is translated along a ray.
4. Scene graphs for managing complex shapes.
5. A binary format for streaming of shapes.

6 Bug Reports

Please send remarks, questions, and bug reports to gino@dtecta.com.

NOTE: There is no termination after a fixed maximum number of iterations of GJK in this version of SOLID, since I believe in an industrial-strength GJK without resorting to tolerance tweaking or forced termination after a certain number of iterations. I welcome any reports of SOLID misbehaving.

Appendix A GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software

which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> *one line to give the program's name and a brief idea of what it does.*
> Copyright (C) *yyyy name of author*
>
> This program is free software; you can redistribute it and/or modify
> it under the terms of the GNU General Public License as published by
> the Free Software Foundation; either version 2 of the License, or
> (at your option) any later version.
>
> This program is distributed in the hope that it will be useful,
> but WITHOUT ANY WARRANTY; without even the implied warranty of
> MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
> GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License
> along with this program; if not, write to the Free Software
> Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19*yy name of author*
> Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
> This is free software, and you are welcome to redistribute it
> under certain conditions; type 'show c' for details.

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program
> 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> *signature of Ty Coon*, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Appendix B THE Q PUBLIC LICENSE

version 1.0

The intent of this license is to establish freedom to share and change the software regulated by this license under the open source model.

This license applies to any software containing a notice placed by the copyright holder saying that it may be distributed under the terms of the Q Public License version 1.0. Such software is herein referred to as the Software. This license covers modification and distribution of the Software, use of third-party application programs based on the Software, and development of free software which uses the Software.

Granted Rights

1. You are granted the non-exclusive rights set forth in this license provided you agree to and comply with any and all conditions in this license. Whole or partial distribution of the Software, or software items that link with the Software, in any form signifies acceptance of this license.
2. You may copy and distribute the Software in unmodified form provided that the entire package, including - but not restricted to - copyright, trademark notices and disclaimers, as released by the initial developer of the Software, is distributed.
3. You may make modifications to the Software and distribute your modifications, in a form that is separate from the Software, such as patches. The following restrictions apply to modifications:
   a. Modifications must not alter or remove any copyright notices in the Software.
   b. When modifications to the Software are released under this license, a non-exclusive royalty-free right is granted to the initial developer of the Software to distribute your modification in future versions of the Software provided such versions remain available under these terms in addition to any other license(s) of the initial developer.
4. You may distribute machine-executable forms of the Software or machine-executable forms of modified versions of the Software, provided that you meet these restrictions:
   a. You must include this license document in the distribution.
   b. You must ensure that all recipients of the machine-executable forms are also able to receive the complete machine-readable source code to the distributed Software, including all modifications, without any charge beyond the costs of data transfer, and place prominent notices in the distribution explaining this.
   c. You must ensure that all modifications included in the machine-executable forms are available under the terms of this license.
5. You may use the original or modified versions of the Software to compile, link and run application programs legally developed by you or by others.

6. You may develop application programs, reusable components and other software items that link with the original or modified versions of the Software. These items, when distributed, are subject to the following requirements:
   a. You must ensure that all recipients of machine-executable forms of these items are also able to receive and use the complete machine-readable source code to the items without any charge beyond the costs of data transfer.
   b. You must explicitly license all recipients of your items to use and re-distribute original and modified versions of the items in both machine-executable and source code forms. The recipients must be able to do so without any charges whatsoever, and they must be able to re-distribute to anyone they choose.
   c. If the items are not available to the general public, and the initial developer of the Software requests a copy of the items, then you must supply one.

Limitations of Liability

In no event shall the initial developers or copyright holders be liable for any damages whatsoever, including - but not restricted to - lost revenue or profits or other direct, indirect, special, incidental or consequential damages, even if they have been advised of the possibility of such damages, except to the extent invariable law, if any, provides otherwise.

No Warranty

The Software and this license document are provided AS IS with NO WARRANTY OF ANY KIND, INCLUDING THE WARRANTY OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Choice of Law

This license is governed by the Laws of Norway. Disputes shall be settled by Oslo City Court.

Table of Contents