OPCODE User Manual

Pierre Terdiman (p.terdiman@codercorner.com)

Last update: august, 18, 2002

I. Introduction

OPCODE is a collision detection library, freely available here:
www.codercorner.com/Opcode.htm

This is the user manual, teaching you how to perform supported queries:
- Mesh-mesh
- Sphere-mesh
- Ray-mesh
- AABB-mesh
- OBB-mesh
- Planes-mesh

We call mesh a surface made of vertices and triangles.
A ray can be a segment or a half-line, not a line.
An AABB is an Axis Aligned Bounding Box.
An OBB is an Oriented Bounding Box.

What you need to do collision queries :
- A collision structure (a.k.a. collision tree) for each mesh
- A collider for each type of queries
- Various caches

We'll describe these things successively in the rest of the document. Frequently asked questions have been appended at the end.

II. Collision trees

II.1. Source trees and optimized trees

To perform fast collision queries on a geometric surface, you need to create a corresponding collision structure, or collision tree, for each surface (mesh).

OPCODE creates collision trees in two passes:
- first it creates a generic source tree, discarded in the end
- then it transforms the source tree, creating an optimized tree used in collision queries

Four different optimized trees are supported :
- Normal trees (2*N-1 nodes, full size)
- No-leaf trees (N-1 nodes, full size)
- Quantized trees (2*N-1 nodes, half size)

- Quantized no-leaf trees (N-1 nodes, half size)

All of them are exposed to clients through an *OPCODE_Model*.

II.2. Building an OPCODE_Model

An OPCODE_Model is a wrapper for collision trees and various additional data. This is the main interface between the user and the library. You need to create one OPCODE_Model for each master mesh in your scene. If your engine supports instancing and your master mesh has several instances, you only need to create an OPCODE_Model for the master mesh. Instances will have their own transform matrix, that will be used later with the master OPCODE_Model to perform correct collision queries.

For now, here's our OPCODE_Model :

```
OPCODE_Model Sample;
```

Before using it for collision queries, you need to build (initialize) it using an *OPCODECREATE* creation structure :

```
// 1) Initialize the creation structure
OPCODECREATE OPCC;
// Surface data
OPCC.NbTris          = ...;
OPCC.NbVerts         = ...;
OPCC.Tris            = ...;
OPCC.Verts           = ...;
// Tree building settings
OPCC.Rules           = ...;
OPCC.NoLeaf          = ...;
OPCC.Quantized       = ...;
// Debug
OPCC.KeepOriginal    = ...;

// 2) Build the model
bool Status = Sample.Build(OPCC);
```

The creation structure is temporary and can be discarded after the OPCODE_Model has been built.

Let's have a closer look at the creation structure and what it contains. We can split the structure members in three parts : *Surface data*, *Tree building settings* and *Debug*.

Surface data

The first members describe the surface (i.e. the mesh) :

```
NbTris = a number of triangles, MUST be greater than 1.
NbVerts = a number of vertices.
```

Since we provide both, i.e. since NbVerts != NbTris * 3, it means we're dealing with an indexed surface. That's why we further provide :

```
Tris = an array of triangle indices
Verts = an array of vertices
```

This is fairly standard and similar to the way indexed primitives are defined in *Direct3D*. The only difference is that we expect 32-bits indices here. So you can't directly use the indices from your index buffers here, since they're usually 16-bits.

Tree building settings

The surface data (source) will be transformed to create the OPCODE_Model. The transform has some parameters defined here.

```
Rules = splitting rules
```

This is a bitmask, actually a combination of *SplittingRules* flags, defined in OPC_TreeBuilders.h. They define the way splitting will be performed while creating the trees. Please refer to the aforementioned file for more info. In OPCODE, the SPLIT_COMPLETE flag is mandatory since we only support complete optimized trees. Hence the usual best splitting rules are the following :

```
Rules = SPLIT_COMPLETE | SPLIT_SPLATTERPOINTS | SPLIT_GEOMCENTER;
```

Finally:

```
NoLeaf  = ...;
Quantized = ...;
```

Those two flags define the type of tree used for the OPCODE_Model. You'll have to stay consistent there, since mesh-mesh queries require the models to be of similar type.

Debug

The last parameter, KeepOriginal, is here for debug-purpose and you shouldn't care about it. In short, it keeps an internal copy of the source tree used to build optimized trees.

III. Colliders and collision queries

III.1. Preliminaries

Each collision query is handled by a different *Collider*. You only need one Collider for each type of query, not one per model. So for example, in your application you only need one *SphereCollider* to collide any sphere you want against any mesh you want. Per-model data are captured in *caches* (in our example in a *SphereCache*), not in Colliders. Caches are briefly described in part IV.

Collision queries can be of two kinds : *First Contact* and *All Contacts*. In "First Contact" mode, the query ends as soon as overlap is detected, i.e. as soon as the first contact is found. In "All Contacts", the query continues until all overlapping primitives have been found.

Collision queries can also use or not use *temporal coherence*. When you perform successive collision queries (say each frame) using for example a sphere against a mesh, most of the time the list of returned faces for frame N will be very similar to the previously returned list for frame N-1. (To the limit, the two lists will be the same.) This is called temporal coherence.

OPCODE takes advantage of temporal coherence in two different ways :

- for "First Contact" queries
- for "All Contacts" queries

In "First Contact" queries, clients are only interested in a boolean answer : does X overlap Y, yes or no ? They usually don't care about the exact list of overlapping entities (else they would have selected "All Contacts"). Temporal coherence in that case is implemented by caching the one-and-only previously touched primitive. Then, before everything else, the cached primitive is tested for overlap in subsequent frames. If it's still overlapping, we can return immediately without doing the actual query, which saves a lot of time. Else we do a normal query and cache the newly touched primitive (if it exists) for next frames.

In "All Contacts" queries, temporal coherence currently only works for several volume queries: Sphere-vs-mesh and AABB-vs-mesh. The idea is to use a larger ("fat") volume for collision queries, and compare normal ones against previously used fat volumes. If the normal volume is included in the fat one, we simply return the previous list of collided primitives. This is a conservative test : some returned primitives won't actually overlap the normal volume. Yet it provides O(1) queries most of the time, and can be very useful.

III.2. Mesh vs mesh collision queries

Introduced in	OPCODE 1.0
Corresponding collider	AABBTreeCollider
Brief description	Mesh P Mesh Q Returns overlapping triangles { TP(i) ; TQ(i) } where TP(i) belongs to P, and TQ(i) belongs to Q.
Useful for	- virtual prototyping - rigid body and mechanical simulation

Steps:

1) Build an OPCODE_Model using a creation structure.

2) Create a tree collider and setup it:

```
AABBTreeCollider TC;
TC.SetFirstContact(...);
TC.SetFullBoxBoxTest(...);
TC.SetFullPrimBoxTest(...);
TC.SetTemporalCoherence(...);
```

3) Setup object callbacks or pointers:

Geometry and topology are *not* stored in the collision system, in order to save some ram. So, when the system needs them to perform accurate intersection tests, you're requested to provide the triangle-vertices corresponding to a given face index.

This can be done through a callback :

```
static void ColCallback(udword triangleindex, VertexPointers& triangle, udword user_data)
{
	// Get back Mesh0 or Mesh1 (you also can use 2 different callbacks)
	Mesh* MyMesh = (Mesh*)user_data;
	// Get correct triangle in the app-controlled database
	const Triangle* Tri = MyMesh->GetTriangle(triangleindex);
	// Setup pointers to vertices for the collision system
	triangle.Vertex[0] = MyMesh->GetVertex(Tri->mVRef[0]);
	triangle.Vertex[1] = MyMesh->GetVertex(Tri->mVRef[1]);
	triangle.Vertex[2] = MyMesh->GetVertex(Tri->mVRef[2]);
}

// Setup callbacks
TC.SetCallback0(ColCallback, udword(Mesh0));
TC.SetCallback1(ColCallback, udword(Mesh1));
```

Of course, you should make this callback as fast as possible. And you're also not supposed to modify the geometry *after* the collision trees have been built. The alternative was to store the geometry and topology in the collision system as well (as in *RAPID*) but we have found this approach to waste a lot of ram in many cases.

Since version 1.2 you can also use plain pointers. It's a tiny bit faster, but not as safe.

```
TC.SetPointers0(Mesh0->GetFaces(), Mesh0->GetVerts());
TC.SetPointers1(Mesh1->GetFaces(), Mesh1->GetVerts());
```

You should setup callbacks or pointers each time, before doing the collision query. That way you can use the same collider for all meshes.

4) Perform a collision query

```
// Setup cache
static BVTCache ColCache;
ColCache.Model0 = &Model0;
ColCache.Model1 = &Model1;

// Collision query
bool IsOk = TC.Collide(ColCache, World0, World1);
```

World0 and World1 are world matrices for Model0 and Model1. If a world matrix is identity, you can pass a null pointer as a parameter instead of the matrix. Then internally the library will use a special code-path with less transforms and less computations.

Returned bool just says everything was ok. It's not the collision status, which is :

```
// Get collision status => if true, objects overlap
BOOL Status = TC.GetContactStatus();
```

If objects overlap, you can further get the colliding pairs using :

```
// Number of colliding pairs and list of pairs
udword NbPairs = TC.GetNbPairs();
const Pair* CollidingPairs = TC.GetPairs()
```

III.3. Sphere vs mesh collision queries

Introduced in	OPCODE 1.1
Corresponding collider	SphereCollider
Brief description	Sphere S Mesh M Returns triangles from M touching (or included in) S.
Useful for	- camera-vs-world collision - rigid body simulation - finding faces lit by a point light

Steps :

1) Build an OPCODE_Model using a creation structure.

2) Create a sphere collider and setup it:

```
SphereCollider SC;
SC.SetFirstContact(...);
SC.SetTemporalCoherence(...);
```

3) Setup object callback or pointers: (see the Mesh-mesh case for more infos)

```
SC.SetCallback(...);
SC.SetPointers(...);
```

4) Perform a collision query

```
static SphereCache Cache;
bool IsOk = SC.Collide(Cache, LocalSphere, OpcodeModel, SphereWorldMatrix, MeshWorldMatrix);
```

The contact status is given by :

```
BOOL Status = SC.GetContactStatus();
```

List of touched primitives is given by :

```
udword NbTouchedPrimitives = SC.GetNbTouchedPrimitives();
udword* TouchedPrimitives = SC.GetTouchedPrimitives();
```

III.4. Ray vs mesh collision queries

Introduced in	OPCODE 1.1
Corresponding collider	RayCollider
Brief description	Ray R

	Mesh M Returns triangles from M stabbed by R.
Useful for	- object picking - raytracing - shadow feelers - in/out tests - sweep tests in rigid body simulation

Steps :

1) Build an OPCODE_Model using a creation structure.

2) Create a ray collider and setup it:

```
RayCollider RC;
RC.SetFirstContact(...);
RC.SetTemporalCoherence(...);
RC.SetClosestHit(...);
RC.SetCulling(...);
RC.SetMaxDist(...);

CollisionFaces CF;
RC.SetDestination(&CF);
```

Please refer to the comments in OPC_RayCollider.cpp for more infos.

3) Setup object callback or pointers: (see the Mesh-mesh case for more infos)

```
RC.SetCallback(...);
RC.SetPointers(...);
```

4) Perform a collision query

```
static udword Cache;
bool IsOk = RC.Collide(WorldRay, OpcodeModel, MeshWorldMatrix, &Cache);
```

Stabbed faces will be found in the user-provided CollisionFaces array.

III.5. AABB vs mesh collision queries

Introduced in	OPCODE 1.2
Corresponding collider	AABBCollider
Brief description	Box B (Axis-Aligned Bounding Box) Mesh M Returns triangles from M touching (or included in) B.
Useful for	- rigid body simulation

Steps :

1) Build an OPCODE_Model using a creation structure.

2) Create a box collider and setup it:

```
AABBCollider AC;
AC.SetFirstContact(...);
AC.SetTemporalCoherence(...);
```

3) Setup object callback or pointers: (see the Mesh-mesh case for more infos)

```
AC.SetCallback(...);
AC.SetPointers(...);
```

4) Perform a collision query

```
static AABBCache Cache;
bool IsOk = AC.Collide(Cache, LocalAABB, OpcodeModel);
```

The contact status is given by :

```
BOOL Status = AC.GetContactStatus();
```

List of touched primitives is given by :

```
udword NbTouchedPrimitives = AC.GetNbTouchedPrimitives();
udword* TouchedPrimitives = AC.GetTouchedPrimitives();
```

III.6. OBB vs mesh collision queries

Introduced in	OPCODE 1.2
Corresponding collider	OBBCollider
Brief description	Box B (Oriented Bounding Box) Mesh M Returns triangles from M touching (or included in) B.
Useful for	- rigid body simulation

Steps :

1) Build an OPCODE_Model using a creation structure.

2) Create a box collider and setup it:

```
OBBCollider OC;
OC.SetFirstContact(...);
OC.SetTemporalCoherence(...);
OC.SetFullBoxBoxTest(...);
```

3) Setup object callback or pointers: (see the Mesh-mesh case for more infos)

```
OC.SetCallback(...);
OC.SetPointers(...);
```

4) Perform a collision query

```
static OBBCache Cache;
bool IsOk = OC.Collide(Cache, LocalOBB, OpcodeModel, BoxWorldMatrix, MeshWorldMatrix);
```

The contact status is given by :

```
BOOL Status = OC.GetContactStatus();
```

List of touched primitives is given by :

```
udword NbTouchedPrimitives = OC.GetNbTouchedPrimitives();
udword* TouchedPrimitives = OC.GetTouchedPrimitives();
```

III.7. Planes vs mesh collision queries

Introduced in	OPCODE 1.2
Corresponding collider	PlanesCollider
Brief description	Planes P Mesh M Returns triangles from M enclosed by P. (on the positive sides of Ps)
Useful for	- view frustum culling (mainly for software rendering) - projective texturing - coarse OBB or LSS queries

Steps :

1) Build an OPCODE_Model using a creation structure.

2) Create a planes collider and setup it:

```
PlanesCollider PC;
PC.SetFirstContact(...);
PC.SetTemporalCoherence(...);
```

3) Setup object callback or pointers: (see the Mesh-mesh case for more infos)

```
PC.SetCallback(...);
PC.SetPointers(...);
```

4) Perform a collision query

```
PlanesCache Cache;
bool IsOk = PC.Collide(Cache, Planes, NbPlanes, OpcodeModel, MeshWorldMatrix);
```

The contact status is given by :

```
BOOL Status = PC.GetContactStatus();
```

List of touched primitives is given by :

```
udword NbTouchedPrimitives = PC.GetNbTouchedPrimitives();
udword* TouchedPrimitives = PC.GetTouchedPrimitives();
```

IV. Caches

Caches are per-couple entities. The client is responsible for managing them.

In theory, there should be one different cache for each possible couple of colliding entities (where a *couple* can be a given sphere vs a given mesh, or a given mesh vs another mesh). In practice, this is only true when temporal coherence is enabled. If it's not, just use a single cache structure filled with relevant data before each collision query.

When temporal coherence is enabled and caches are needed, they're mainly filled with data computed during a collision query. Those data are then reused in subsequent queries to possibly find collision results faster. Cached data are query-dependent, i.e. they're not the same for sphere-mesh or mesh-mesh queries. That's why you have SphereCache, AABBCache, etc...

V. FAQ

Q: Am I allowed to use OPCODE in a commercial program ?
A: Yes. The library is free. No GPL or LGPL, you can do whatever you want with the code. I would appreciate it if you write somewhere in your product that you're using OPCODE, but that's not mandatory. I would also appreciate a notification by mail, just to know how people are using the lib.

Q: Why callbacks ? Why not simply pointers ?
A: Since OPCODE 1.2 you can select callbacks or pointers at compile time. At first, callbacks have been used in sake of flexibility, not to force clients to use the same geometry / topology as I do. With pointers, clients *must* use indexed triangle lists with 32-bits indices. With callbacks, they can use 16-bits indices in the background or non-indexed geometry, provided they feed OPCODE with *VertexPointers* in the end.

Q: Why not an extra callback to get triangle normals instead of computing them on-the-fly in overlap tests ?
A: Simply because that's the way I did it in the first place. Since the goal was to save memory, keeping an array of normals around didn't seem like a good idea. Now an extra callback (or extra pointers) could be added, but gains would be minimal (if any) so I probably won't bother. But you can try it if you want and share your results with everyone....

Q: How does OPCODE perform compared to other libs ?
A: Well, it's always difficult to say since most of the time the same library is not *always* the fastest. (I assume we're speaking about speed here, not memory usage). In any case, I'd say it performs well, as it seems to be (overall) faster than SOLID 2.0, RAPID and ColDet. I didn't try against other libs (QuickCD, PQP, Swift++, etc). Give it a try and tell me !

Q: Why AABB trees ?
A: Because they're easy to deal with, efficient and memory-friendly. Spheres are more memory-friendly but really too coarse, and sphere-trees are painful to build. OBBs need more memory and are not that faster (see OPCODE 1.0 vs RAPID). Other volumes (k-Dops, etc)

are interesting theoretically but practically they're not worth it. So AABBs are the best compromise, I think.

Q: What about SIMD optimizations ?
A: Sure, gimme a P3.

Q: Have you thought about adding an option for building the AABB trees bottom up?
A: Yes, and I think it would provide better trees and faster queries. But I haven't had the time to play with them so far. Later, maybe..... In any case, that's definitely something to test.

Q: If you're using OPCODE in physics do you have a neat solution for turning intersecting triangle pairs into contact points/normals for use in the physics?
A: No. I don't have a neat solution.

Q: Did you read the article about compressed AABB trees in *Game Programming Gems 2* ?
A: Yes, and it's very nice. It takes advantage of what I used to call "tree coherence", even if they don't name it that way. It was planned for OPCODE at first, but I would have to use (Min, Max) boxes whereas I currently use (Center, Extents) ones, so I need to rewrite many things here, and right now I can't. The trees in GPG2 are more compressed than the ones in OPCODE. I don't know how they behave as far as speed is concerned, since there's no ready-to-use source code provided in the book.

Q: I would like to port it to Linux, any suggestions?
A: Several people have done it, and reported the port to be easy. Two places to check :
- The Flat Four engine:
 http://www.379.com/f4/
- The ODE library and its TriangleCollider contribution:
 http://q12.org/cgi-bin/wiki.pl?TriangleCollider

Q: There is a check against exception handling (#if defined(_CPPUNWIND)) what is the reason for that? I have commented it out, but wonder if that might break anything.
A: No reason, it's just that OPCODE uses some low-level libraries of mine (basically "my engine"), and at some point I added the check because -*in my engine*- I wanted to disable exceptions. As far as OPCODE is concerned, you can put them back.

Q: Some precision problems appeared in the SphereCollider, when a sphere was collided against really big triangles.
A: It happens when triangles are *really* too big. It has been reported that using doubles instead of floats in the sphere-triangle overlap test helped. Tesselating your big triangles a bit is probably a better solution.

Q: How do I deformable models ? Dynamic trees ?
A: You don't. Seriously, OPCODE is not very well-suited to dynamic trees. There are two different trees to begin with : the source tree and the optimized tree. Updating the former is relatively easy. But updating the latter doesn't sound very exciting as it means re-quantizing, etc. It can work, but it probably won't be efficient.

Two notes :

- if you need it to be fast (say for deformable objects like cloth), the best way is to update the source tree in the same way as Gino van den Bergen did in SOLID 2.0. (or Thomas Larsson in his articles). Check out his code, it's pretty easy. Then, *maybe* rebuilding the optimized tree from there is an option. Else you can also use the source tree directly - after all that's what SOLID does. But then use SOLID, not OPCODE.

- if you don't need it to be fast (say it only happens from time to time), *maybe* you can simply discard the old trees and rebuild everything from scratch with the new geometry.

Q: How do I do LSS-vs-mesh queries ?
A: There's no direct LSS query since LSS-vs-box is too slow. Instead I use a box query, gather touched faces, then do a LSS-triangle collision response on that. But this is done out of OPCODE since it's not directly relevant to the lib.

Actually I may even be using a plane-query instead of a box-query. It's usually faster, yet more conservative.

Side note: LSS = Line Swept Sphere = a capsule.

Q: What are future features ?
A: I have many things on my TODO list already :

- implementing "All Contacts" temporal coherence for other volumes (OBB, maybe planes or ray)
- moving some AABBTree queries from another part of my code to OPCODE. (queries on "vanilla AABB trees", not only on "optimized" trees).
- distance queries (already working but really too slow)
- bottom-up tree building
- implementing tree-coherence as in GPG2
- trying the P3's prefetch instructions to speed up recursive queries
- doing some OpenGL samples, better docs, tutorials...
- trying a huge list of remaining tricks to compress the trees better
- etc