Sweep-and-prune

Version 0.2
Pierre Terdiman – september 11, 2007

I) Single SAP

1) Presentation

The sweep-and-prune (SAP) [8] is a broad-phase algorithm whose goal is to determine overlapping pairs of objects in a 3D world. An object in the SAP is defined by an axis-aligned bounding box (AABB). Given a set of N boxes, the goal is to find the subset of overlapping pairs.

A SAP implementation can provide *direct* or *incremental* results. A direct SAP provides the full set of overlapping pairs, each frame. An incremental SAP provides lists of new and deleted pairs for current frame. A related but different concept is the way the SAP operates internally: by starting from scratch each time, or by updating internal structures, persistent over the SAP's lifetime. We will call the first type *brute-force* and the second type *persistent*. This can be a bit confusing since our persistent SAPs are sometimes called incremental in other articles, but this is really a different notion: for example the initial SAP implementation from *V-Collide* [5] is *persistent* but still provides *direct* results.

This paper focuses on persistent SAPs that provide incremental results. Nonetheless we will also use *box-pruning* along the way, which is a brute-force SAP that provide direct results.

2) Algorithm overview

The basic idea behind the algorithm is this: two AABBs overlap if and only if their projections on the X, Y and Z coordinate axes overlap.

The projection of each AABB gives three [min; max] intervals and each interval is defined by two "end points": one min, one max. Those end points are stored in a structure, and they are kept sorted at any given time. There is one structure per axis.

Each *EndPoint* has been generated from an initial box, and we keep a reference to this box inside the *EndPoint* structure. We also have a collection of "box objects", where each box keeps a reference to its end points, and possible user-defined data associated with the box.

Alongside there is an independent, separate structure, keeping track of overlapping pairs of boxes - this is, after all, what we are after. We will call it a *pair manager* (PM), and we can assume for the moment that it is a black box. Its interface just allows us to add or remove a pair of objects.

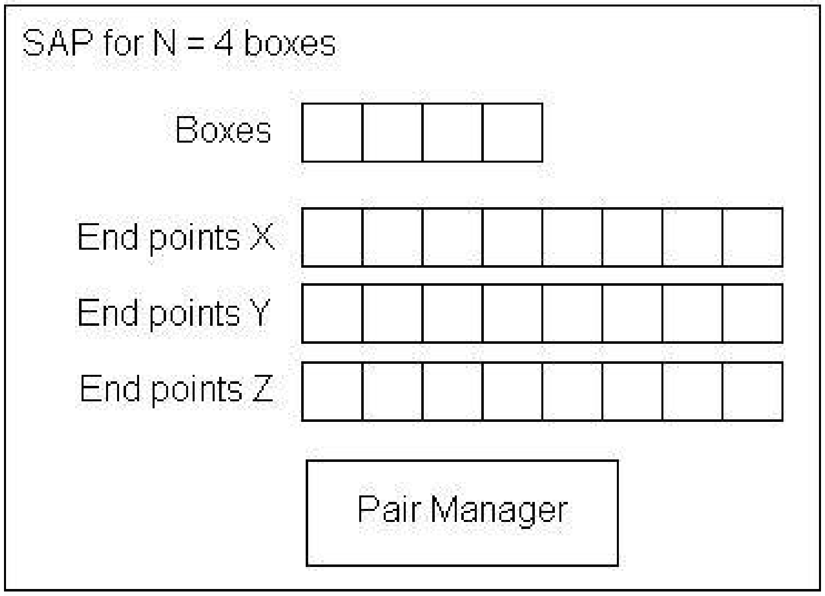

Figure 1 – main elements of a SAP

Let’s recap. So far our *EndPoint* structure looks like this:

```
struct EndPoint
{
     Box*   mOwner;     // Owner box (could be box ID)
     float  mValue;     // Actual value
     bool   mIsMin;     // Value is min or max
};
```

And our *Box* structure looks like that:

```
struct Box
{
     EndPoint* mMin[3];       // Min EndPoint over X,Y,Z
     EndPoint* mMax[3];       // Max EndPoint over X,Y,Z
     void*     mUserData;     // User-defined data
};
```

We also defined three sorted collections of N*2 end points, one collection per axis (Figure 1). There are two traditional ways to implement the sorted collection: using *linked lists*, or using *arrays*. Both have pros and cons, but nowadays arrays are better for a number of reasons that we will come back to. When using linked lists, the *EndPoint* structure must be updated so that we can walk the list:

```
struct EndPoint
{
     Box*      mOwner;    // Owner box (could be box ID)
     EndPoint* mPrevious; // Previous EndPoint whose mValue
                          // is smaller than ours, or NULL
     EndPoint* mNext;     // Next EndPoint whose mValue is
                          // greater than ours, or NULL
     float     mValue;    // Actual value
     bool      mIsMin;    // Value is min or max
};
```

Using a linked list, the *EndPoint* structure is immediately quite bigger than the initial version. However with arrays, we can use the initial structure as-is. This is the first place where an array-based implementation is better:

- it consumes less memory for the sorted collections
- it has better cache coherence since walking the list simply means reading the *EndPoint* structures in linear order (since they are sorted)

Arrays also allow extra optimizations that we will cover later. The important thing to keep in mind at this point is simply that this document mainly focuses on an array-based implementation.

Now that we defined the basics, let us see how we can support the three necessary operations that we need:
- adding an object to the SAP
- updating a moving object
- removing an object from the SAP

Our SAP interface at this point would be something like:

```
class SweepAndPrune
{
    Handle AddObject(const AABB& box, void* user_data) = 0;
    void   UpdateObject(const AABB& box, Handle handle)= 0;
    void   RemoveObject(Handle handle)                 = 0;

    <plus some functions to retrieve the set of overlapping
     pairs>
};
```

For reasons that will become clear afterwards, we will first describe the object update.

3) Updating an object

When new bounds are provided for a given object, we need to update its end-points values (*EndPoint::mValue*), and we need to move the *EndPoint* structures in the collections, to keep them sorted. Updating the values is easy since we can directly go from a *Box* to its *EndPoints*, through the *mMin* and *mMax* members. Moving the end points within the collections is also not a big issue: we keep comparing the new *mValue* to other *mValues* in the collection, until we find our correct place. The comparisons can be either FPU or CPU-based, as described in *Appendix A*. When using linked lists, we simply walk the list until we find the correct location, then insert the current end point there. When using arrays however, the whole *EndPoint* structure has to be moved up or down the array, and the *EndPoint* references in each *Box* owner have to be updated. This means a lot of memory has to be touched, and implementing this as efficiently as possible is key to SAP performance. Reducing the size of the *EndPoint* structure is important here, and a typical optimization is to pack the *mIsMin* boolean along with the *mOwner* box reference (either by using the last bit of the pointer, or better, by using a 31-bit box index and a single bit to define a min or a max). If we do this with arrays, we obtain the following “canonical” (*) *EndPoint* structure:

```
struct EndPoint
{
     int    mData;     // Owner box ID | MinMax flag
     float  mValue;    // Min or max value
};
```

(*) PhysX, ICE and Bullet all use a similar end point structure.

Notice that the structure is only 8-bytes, which means it is as small as (or smaller than) just the linked-list pointers in a linked-list-based implementation. In other words, swapping two array entries does not actually touch more memory than moving a linked-list element. Of course with an array, all elements between the initial and final position of an end point have to be shifted, but when the motions are reasonable objects do not move that much in the collections - and we do not touch that much memory compared to a linked list. In any case, properly implemented array implementations have been found to outperform the linked list counterparts, so arrays are the preferred design choice. Finally, it is worth reporting that the code to walk the collections – regardless of their nature - can be simplified a bit by using *sentinels*. This is described in *Appendix B*.

Keeping the collection sorted is only half the job. At the *same time* we must keep the set of active pairs updated in the pair manager. It is important to realize that both sorting the structure and adding / removing overlapping pairs happen at the same time. This is the strength of the algorithm, and what makes the *persistent* mode worth it. If an object only moves a little bit, not changing its position in the sorted structure, we immediately know that no pair has been created or deleted, and the PM is not touched. At the limit, when no object moves in the world, no time at all is spent in a persistent SAP (at least with good implementations). A brute-force SAP however would re-compute everything from scratch and waste a valuable amount of time finding that nothing changed.

Update rules

But let us focus on something more interesting than a dead world with no-one moving, and pretend that they do. We will examine the PM update rules with a few practical examples.

We work in 1D because it is easier, but the ideas are the same in 3D – only done three times on three different axes. Pretend we have three intervals, i.e. six end-points marked as (*), the number is the order in the sorted structure; the [number] is the index of the parent box.

```
0   [0]    2         4   [1]    5
*---------*          *---------*
       *---------*
       1   [2]   3
```

Say box [2] is moving right:

- its Min (1) becomes greater
- if it passes a Max (2) ➔ [0] and [2] stop overlapping
- if it passes a Min (4) ➔ nothing special to report

- its Max (3) becomes greater

- if it passes a Min (4) ➔ [1] and [2] start overlapping
- if it passes a Max (5) ➔ nothing special to report

Now with this configuration:

```
0   [0]    2    3   [1]    5
*----------*    *---------*
          *---------*
          1   [2]    4
```

Say box [2] is moving left:

- its Min (1) becomes smaller
- if it passes a Max (2) ➔ [0] and [2] start overlapping
- if it passes a Min (0) ➔ nothing special to report

- its Max (4) becomes smaller
- if it passes a Min (3) ➔ [1] and [2] stop overlapping
- if it passes a Max (2) ➔ nothing special to report

Now let us examine this special case: [2] is moving left, passes [0], and they swap positions:

```
0   [0]    1                          0   [0]    1
*---------*              ➔            *---------*
               *---------*        *---------*
               2   [2]   4         2   [2]   4
```

In one frame:

- Min (2) becomes smaller than Max (1) ➔ [0] and [2] start overlapping
- Min (2) becomes smaller than Min (0) ➔ nothing special
- Max (4) becomes smaller than Max (1) ➔ nothing special
- Max (4) becomes smaller than Min (0) ➔ [0] and [2] stop overlapping

We must do "start" before "stop" since:

- start(x, y) then stop(x, y) ➔ nothing
- stop(x, y) then start(x, y) ➔ pair (x, y) is created, which is wrong

If this happens on all three axes, it means we can have, in one run:

- 3 calls to add pair (x, y)
- 3 calls to remove pair (x, y)

Of course this supposes that we create the pair as soon as it starts overlapping in one dimension, because our example is 1D. In 3D we do not create the pair immediately: we have to make sure the boxes overlap on the three axes before creating it. There are two main ways to do it:

- By keeping an overlap counter for each pair, and only “really” reporting the overlap when the counter reaches 3. This is not very practical since the counter has to be stored somewhere, and referenced by the pair itself.

- By performing the full 3D overlap test between the owner boxes to validate pair creation. Since the test is very quick, this is usually the preferred approach.

Overlap test optimization

The 3D overlap test can be greatly sped up with array-based implementation. Recall that in the array version end-points are simply stored in sorted order, in a linear array. It means that we do not really need to compare the actual end-point values together (*EndPoint::mValue*), it is enough to simply compare the *EndPoint indices* (*Box::mMin* and *Box::mMax*), i.e. their positions within the sorted array. This is faster for two reasons:

- *mMin* and *mMax* are integers (pointers, 32-bit, or 16-bit indices), and comparing integers is faster than comparing potentially floating-point values

- We do not need to touch the sorted collection at all, only the memory associated with the array of boxes. This reduces cache misses a lot.

Note however that this optimization is only possible on two axes: the axes not currently being updated by the SAP update code. On the current axis, our max end point for example might not have been moved already when we process our min. So it might be located at an incorrect position, giving incorrect overlap results. However it is worth mentioning that a single “real” comparison is needed on this axis – the other one, either min or max, is already taken care of by the SAP update code itself, when searching for our correct position in the collection. Finally, note again that this optimization only works with array-based versions, which is also one reason why they end up being faster than their linked-lists counterparts.

Overlap test for deletions

When removing a pair, the overlap test is often useless: if the pair did not actually exist, the pair manager is usually able to detect it, and early exit – so the extra overlap test looks like a waste of CPU time. However there are two reasons why performing the overlap test prior to any removal attempt is a good idea:

- The overlap test is fast, and it might be faster than the code removing a pair from the PM. It all depends on how the PM is implemented, so your mileage may vary.

- The number of “real” removals (concerning actually existing pairs) can be a *lot* smaller than the number of apparent removals issued by the SAP code. Figures like 10.000 apparent removals filtered down to 3 real removals by the overlap test have been recorded. And in this case, the overlap test saves a significant amount of time overall.

4) Adding an object

a) Simple approach

To create a new object in the SAP we need to do three things:

- Allocate a new box in our collection, and new end-points for this new box

- Make sure the end-points are inserted at the right place in the sorted collections

- If this new object touches already existing boxes, make sure the correct pairs are created in the pair manager

The easiest way to implement the two last parts is to do them at the same time, reusing the update function we already discussed – and this is why we discussed it first.

The idea is simply to create the box very far away, to ensure it does not collide with anything else, and then to move it to the right place in a second pass. That is, we want the new end-points to be initially located at the tail of the linked lists or the end of the arrays. Creating them this way is very easy, since we do not need to find their correct location immediately – we can just append them to our structures. The correct locations are found just afterwards by the update function, called with the real box position this time.

This method is simple to implement but it is not very efficient. First, when adding an object to the SAP we do *not* need to perform all the overlap tests contained in the update function. We only need to do them on the last axis (Z). There is an intuitive reason for this: performing a SAP update where a box moves freely from one position to another in 3D, is the same as performing three simpler updates for three simpler motions along coordinate axes. In other words, if the center of the box is (xc; yc; zc), doing this:

(xc; yc; zc) ➔ (xc’; yc’; zc’)

…is the same as doing:

(xc; yc; zc) ➔ (xc’; yc; zc) ➔ (xc’; yc’; zc) ➔ (xc’; yc’; zc’) i.e. moving the object along X first, then along Y, then along Z, with three different calls to the update function. The key point to realize is that any overlap you get when moving along the X and Y axes is meaningless: on the Z axis the object is still very far away, not touching anything, so all the overlap tests will return false (no overlap). Hence it is better to just skip them, and only perform them when moving along the last axis. To support this, we may modify the update function to conditionally disable the overlap test (but this introduces an overhead), or to duplicate the whole update function, one with overlap tests, one without (but this produces code bloat).

Whatever we choose, the process is still not very efficient: the end-point is added at the start (or end) of the collection, and bubbled up (or down) the list until the correct position is reached. This simple approach is O(N) by nature, and it can be quite time consuming because:

- This is done three times, once per axis
- It potentially touches the memory for all objects in the SAP structure

Several methods exist to improve the speed of SAP insertion (stabbing numbers, markers, etc), but we will not discuss them all in this document, as they are ultimately not as useful as the one we will now focus on: *batch insertions*.

b) Batch insertions

If multiple objects are inserted at the same time, it is possible to perform batch insertions, and improve insertion performance. The idea here is to pre-sort the end-points of *incoming* boxes, and then insert them all in the SAP's end-point arrays at the same time. It is easy to see why it is a win: if you already parsed the start of the array and found the correct insertion position for a given end-point P, you can start from this same position when inserting end-point P+1 (the next end point in the pre-sorted array) instead of starting from the start of the array.

Let us have an example, to make things perfectly clear. Pretend we have to insert two new objects, i.e. four new end-points. We are only going to examine what happens for a single axis, but of course things are exactly the same on the other ones. So, let's say our incoming end-points' values are:

42, 10, 60, 12

Pretend we have four objects, or eight end-points, already in the SAP. The already existing collection of end-points for axis X can be, for example:

-6, -1, 0, 11, 15, 20, 50, 150

Now, the first step is to sort the incoming values backwards – largest values first:

60, 42, 12, 10

Then, we start inserting the largest incoming end-point (60), from the end of the existing array (150) towards its start (-6). We quickly find that the new configuration should be:

-6, -1, 0, 11, 15, 20, 50, **60**, 150

With a classical insertion method the process would be repeated for the next incoming end-point (42), from the end of the collection again, and four comparisons would be needed (against 150, 60, 50 and 20) to find the correct position. With batch insertions however, and since our incoming values have been sorted, we know that the new end-point (42) is smaller than the previous one, so we can simply start from where we were (60) instead of restarting from the end of the array. Hence, we only need two comparisons (against 50 and 20) to find the correct position:

-6, -1, 0, 11, 15, 20, **42**, 50, 60, 150

In practice, batch insertions are thus a lot faster than multiple single insertions. This is why the “markers” method becomes useless: in a way the first end-point we insert (60) becomes the marker for the next end-point. Traditional markers can hence only speed up insertion of the first end-point, and overall this is simply not worth it (since the artificial markers otherwise pollute the collections of end-points and consume extra memory).

c) Box pruning

When batch-inserting the new objects, we still need to create new pairs of overlapping objects – only when processing the last axis though, as seen before. This can be done by keeping a list of open intervals and performing overlap tests on corresponding boxes, all of that while parsing the sorted collections and inserting the new end-points. However the code is not very easy to write, not very good looking, and ultimately not very efficient.

A better idea is to use *box-pruning* to compute the new overlapping pairs in a second pass. Box-pruning is a broad-phase algorithm itself, which could be used all alone to solve our initial problem. [4] However it differs from the typical sweep-and-prune algorithm on several points:

- It uses a single axis instead of three. Typically the axis with largest variance is used. Sometimes the best axis is computed at runtime (it does not have to be a coordinate axis)

- It is not output sensitive. It roughly always consumes the same amount of time to compute the set of overlapping pairs. This time is not too big (it typically outperforms a normal SAP when all objects are moving), but not too small either (otherwise we would just use this approach all the time…)

- It does not provide incremental results. It only provides the full set of overlapping pairs, i.e. direct results.

- It does not use persistent structures. It always re-computes everything from scratch, making it quite efficient for one-shot queries. As a side effect, it also does not consume much memory.

- It comes in two flavors: “complete” or “bipartite” [6]. The “complete” version finds pairs of overlapping boxes within a single set. The “bipartite” version finds pairs of overlapping boxes between two sets A and B, where one box belongs to A and another belongs to B. Intersections between boxes of the same set are not reported in the bipartite version.

It turns out that using box pruning in a second pass to compute the new overlapping pairs is a lot faster than doing it on-the-fly while inserting the new end-points. There are two separate calls: one “complete” box pruning to find overlapping boxes amongst the incoming objects, and one “bipartite” box pruning to find overlaps between the new objects and the old ones – previously contained in the SAP. As a bonus, when using a separate box-pruning pass we do not need to worry anymore about duplicating the insertion code (as discussed in 4a): we only need the version without the overlap tests.

5) Removing an object

a) Simple approach

To remove an object from the SAP we need to:

- Remove all pairs involving this object from the PM

- Remove the object’s *EndPoint* structures

- Remove the object from our collection of boxes

The easiest way to remove pairs involving our object is to use the same strategy as when creating it: move the object very far away using the update function. The function will automatically remove the pairs from the PM as existing overlaps vanish when the object goes to infinity. Afterwards, removing the *EndPoint* and *Box* objects is a simple book-keeping matter. Note that with this approach, arrays or linked lists are really the same: our *EndPoint* end up either at the end of the array, or as the tail of the linked lists. In both cases, removing them is not an issue.

Now, if the PM quickly supports removals of all pairs involving a given object, then it is a good idea to use that feature. In this case there is no need to move the object to infinity, and one can immediately remove its end-points from the collections. Linked lists are a bit faster here – arrays need shifting and pointer fixing, which can be quite time consuming even for a single delete.

In any case, the resulting code will not be very efficient anyway. In the same way we used batch insertions to quickly add multiple objects to the SAP, we need to use *batch deletions* to quickly remove them for it.

b) Batch deletions

Batch deletions use the same idea as batch insertions: we do not want to start parsing the collections of end-points from scratch, for each object. So the code here usually works in two passes:

- In a first pass over the removed boxes, their end-points are tagged for removal within the sorted collections (for example using a special value for the *EndPoint::mData* member)

- In a second pass over the collection of end-points, marked ones are actually removed.

With arrays, there is little to optimize here: the entire array has to be potentially shifted, and pointers or links from the box objects to their end-points have to be updated. However with batch insertions, the arrays are only roughly shifted *once*, not once per end-point…

6) Pair manager and incremental results

So far we have assumed the pair manager was a black box. The only thing we know is its interface, allowing us to either add or remove a pair from the structure. Typically a pair is defined as a couple of 16-bit or 32-bit indices, each index being assigned to one of the boxes in the SAP. Using indices is better than using the box pointers for a few reasons:

- They can use less memory (especially on 64-bit machines)

- They are easier to hash, if needed

- They do not become invalid when the box array is resized (in which case the pointers stored in the PM would need remapping. Indices do not.)

There are multiple ways to implement the pair manager, all of them with different performance and memory trade-offs. Here we will just briefly mention a few possible ways:

- A simple 2D array, but it usually consumes way too much memory.

- An array of linked lists, indexed by the min of the two indices of the pair. This is what was used in the original V-Collide [5], but it is not very good.

- A hash-based linear array, as used in ICE [3].

- An STL map

To provide direct results, no extra work is necessary: one can simply reports all the pairs stored in the PM. This is where the ICE solution shines, since all the active pairs are directly available as a contiguous linear array, with no holes or invalid pairs. Solutions using 2D arrays or bitmaps are not so easy to parse, and require more complex iterator code to skip invalid entries.

Deriving incremental results is easy to do once the direct results are available, provided we can store some persistent information with each active pair. Here is one way to do it. The usual function to add a pair to the PM during the SAP updates looks like:

```
void AddPair(const void* object0, const void* object1, uword id0, uword
id1)
{
    // mPairs is a pair manager structure containing active pairs. The
    // following call adds the pair to the structure. If the pair is a
    // new one, a new internal "SAP_Pair" structure is allocated for
    // it, and returned. If the pair was already there before, its
    // previously allocated "SAP_Pair" is returned.

    SAP_Pair* UP = mPairs.addPair(id0, id1, null, null);
    assert(UP);

    // The "object0" and "object1" pointers are null for newly created
    // pairs.
    if(UP->object0)
    {
        // This is a persistent pair => nothing to do (we don't need to
        // notify the user)
    }
    else
    {
        // This is a new pair. First, we update object pointers.
        UP->object0 = object0;
        UP->object1 = object1;
        // Then we set a flag saying that the pair is in the "mKeys"
        // array (and we add the pair to the array).
        // mKeys will be processed later in a second pass.
        //
        // We store pair indices in mKeys because pointers would be
        // invalidated when the array is resized in "addPair"
        // operations. The indices are guaranteed to stay valid because
        // we never remove pairs here (it is done later in a second
        // pass)
        UP->SetInArray();
        mKeys.Add(mPairs.getPairIndex(UP));

        // Mark the pair as new
        UP->SetNew();
    }
    // Mark the pair as added
    UP->ClearRemoved();
}
```

And the counterpart function to remove a pair looks like:

```
void  RemovePair(const void* object0, const void* object1, uword id0,
uword id1)
{
    // We try to find the pair in the array of active pairs
    SAP_Pair* UP = mPairs.findPair(id0, id1);
    if(UP)
    {
        // The pair was found in the array => we have to remove it. If
        // the pair is not already in mKeys, add it to this array and
        // update the flag. If it's already marked as "in the array",
        // it might be a double removal of the same pair => nothing to
        // do then, it has already been taken care of the first time.
        if(!UP->IsInArray())
        {
            // Setup flag saying that pair is in "mKeys", then add to
            // this array
            UP->SetInArray();
            mKeys.Add(mPairs.getPairIndex(UP));
        }
        // Mark the pair as removed
        UP->SetRemoved();
    }
    // else the pair was not in the array => nothing to do
}
```

So, we update the SAP for all objects, things get added to *mKeys*. After all the updates, we dump incremental results using the following function. There are two callbacks: one for newly created pairs, one for deleted pairs. The list of currently active pairs is always available in a contiguous linear array, whose start is *mPairs.activePairs*.

```
udword DumpPairs(SAP_CreatePair create_cb, SAP_DeletePair delete_cb,
void* cb_user_data)
{
    // Parse all the entries in mKeys
    const udword* Entries = mKeys.GetEntries();
    const udword* Last = Entries + mKeys.GetNbEntries();
    mKeys.Reset();    // Reset for next time

    udword* ToRemove = (udword*)Entries;
    while(Entries!=Last)
    {
        udword ID = *Entries++;    // Stored before as "getPairIndex"

        SAP_Pair* UP = mPairs.activePairs + ID;  // Fetch pair back

        assert(UP->IsInArray());   // Comes from mKeys, must be true

        // The pairs here have been either added or removed. We
        // previously marked added ones with "ClearRemoved" and removed
        // ones with "SetRemoved". Use this flag now.
        if(UP->IsRemoved())
        {
            // No need to call "ClearInArray" in this case, since the
            // pair will get removed anyway

            // Remove callback
            if(!UP->IsNew() && delete_cb)
            {
                (delete_cb)(UP->GetObject0(), UP->GetObject1(),
                                           cb_user_data, UP->userData);
            }

            // We don't remove the pair immediately to make sure the
            // indices we're parsing are still valid. So we will
            // actually remove the pairs in a third pass below. We
            // reuse the same buffer here, to save some memory.
            *ToRemove++ = udword(UP->id0)<<16|UP->id1;
        }
        else
        {
            // This is a new pair. It is not in the mKeys array anymore
            // after processing, so we clear the flag.
            UP->ClearInArray();

            // Add callback
            if(UP->IsNew() && create_cb)
            {
                UP->userData = (create_cb)(UP->GetObject0(),
                                       UP->GetObject1(), cb_user_data);
            }
            UP->ClearNew();
        }
    }

    // 3rd pass to do the actual removals.
    Entries = mKeys.GetEntries();
    while(Entries!=ToRemove)
    {
        const udword ID = *Entries++;
        const udword id0 = ID>>16;      // ID of first pair
        const udword id1 = ID&0xffff;   // ID of second pair
        bool Status = mPairs.removePair(id0, id1);
        assert(Status);
    }

    // Return number of active pairs
    return mPairs.nbActivePairs;
}
```

And that's about it. *mPairs* is an ICE *PairManager* [3], a bit modified to hold object pointers and user-data. *mKeys* is a simple "*Container*" (like an STL-vector of ints). Everything is O(1), there are no linked lists, only linear arrays, and memory usage is quite small (O(number of active pairs) for the pair manager, O(number of changed pairs) for *mKeys*).

This is just one way to implement this, maybe not the most efficient or anything, but quite solid. For example it has recently been used in Erin Catto's "*Box 2D*" simulator. [9]

II) Multi-SAP

1) Issues with the typical SAP

The typical SAP works great in a vast number of cases, and often outperforms alternative solutions like grids, especially when clients are interested in incremental results (as defined at the start of this article). However it does not scale very well, and its performance decreases when the SAP contains a lot of objects. In particular, it has two main problems: interactions with far away objects and insertion of new objects.

a) useless interactions with far away objects

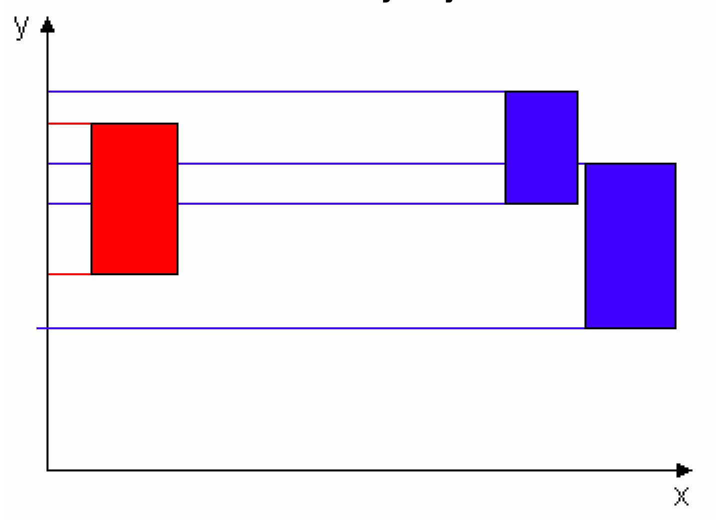

Fig 2. Cluttering with far away objects

Consider figure 2. It has a red object moving vertically (along Y), and two far away objects (in blue), not moving. The blue objects are not overlapping the red one, and are potentially very far from it along the X axis. However, their projections on Y overlap the projection of the red box on this axis. So, when the red box moves vertically, its end-points interact with the end-points from the blue boxes, which mean that some work will be performed to swap the end-points on this axis and keep the collection sorted (*). In other words those far away blue boxes, despite being very far away, interact with the red box within the SAP structure. *The more objects you have in the SAP, the more time consuming it is to update one of them, even though they might not be touching anything.*

(*) This is especially true along the up axis, when the game world is flat and almost 2D. Imagine a flat plane with a lot of boxes located all over the place: no matter how far away they are from each other, it is likely that their projections on the up axis overlap a lot.

b) Insertion of new objects

In the first part of the document we saw that batch insertions and batch deletions greatly increased performance. Nonetheless, batching does not help when a *single* object has to be created or deleted. Moreover performance still drops when the number of objects in the SAP increases, because the collections of end-points potentially need to be fully parsed during insertions and deletions. Said a different way, adding to or removing an object from the SAP is still not fast enough.

2) The Multi-SAP approach

a) Basic idea

A natural way to deal with the aforementioned SAP issues is to use multiple SAPs instead of a single one. The idea is simply that using several SAPs limits the number of objects N in each of them; hence it makes updates and insertions faster. Updates are faster because far away objects are likely to be in another SAP, so the useless far-away interactions are reduced. Insertions are faster simply because N is smaller.

In this document we consider the simplest way to approach this idea: a grid of SAPs. The grid could be 2D or 3D, but in the remaining parts of the document we will consider a 2D grid. (Fig 3)

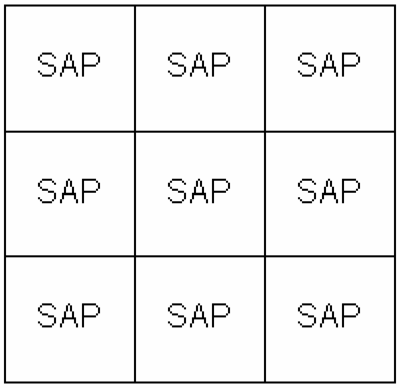

Fig 3. A grid of 3*3 SAPs

The world is divided in N*N *non-overlapping* grid-cells and a different SAP is allocated in each cell. Each game object overlaps one or several SAPs, and it is added to each of them. We must make sure that a pair of objects (A, B) existing in different cells (i.e. different SAPs) is only reported once. For example in figure

4, the red and blue objects overlap, and they belong to both SAP 3 and SAP 4. We must make sure that the (Red, Blue) pair of objects is only reported once to the client.

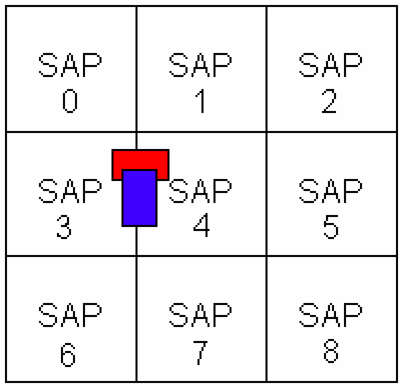

Fig 4. Same pair in different SAPs

b) Requirements

The multi-SAP version has extra requirements compared to the single SAP version.

- Since we use a grid, we need extra parameters to define it. In particular we need to define the grid-cell size, either directly or indirectly. The easiest way is probably to define the world bounds, and the number of SAPs we want to allocate along each axis.

- We may want to add extra constraints on the size of dynamic objects, so that they are guaranteed not to overlap more than 4 cells at the same time (or 8 in 3D). It is both for efficiency reasons and for ease of implementation.

c) Static objects

Static objects can be an issue because they potentially cover the whole world. At least two approaches are possible:

- Static objects are simply added to all cells if needed. It is simple, but it really multiplies the number of SAP entries if static objects are big.

- They can be stored in a different structure completely, like an AABB-tree, and the static-vs-dynamic interactions might use another code path. (This approach was suggested by Erwin Coumans).

The first approach is easier to implement and gives good results in practice, although the implementation can be quite subtle – if you want it to run fast.

3) Implementation

a) Grid design choices

The grid can be either a fixed grid of N*M cells or a hashed grid of N entries. Both are pretty much equivalent in terms of implementation, and conceptually the same when one sees the index computation in a fixed grid as a particular hash function. The fixed grid is usually wrapped, meaning that any object passing its bounds is re-introduced at the other end of the grid.

b) Sharing the pair manager

As we previously saw, the typical SAP implementation uses some kind of pair manager (PM) structure to keep currently overlapping pairs. A key point to realize is that *this structure already filters out redundant pairs*, because the same pair (A, B) can be added to the PM multiple times, from different axes, *even in a single SAP version*. We covered this before, in the paragraph dedicated to object updates.

As a consequence of this, adding another SAP in the mix *can simply be seen as adding another set of axes in the single version*. If we decouple the PM structure from the SAP, and have all SAPs in our grid reference this same PM, then duplicate pairs existing in different SAPs are *automatically* filtered out. There is no extra code needed, no "boundary management", nothing like this.

c) Object management for multiple SAPs

This is the only place where real new code is needed. The multi-SAP should have the same interface as the single-SAP, and make sure it calls the right functions on underlying SAPs in the grid.

Adding an object

Adding an object to the structure is done by rasterizing its bounds into the grid to find touched cells. Then we add the object to the SAPs of touched cells. A handle object is allocated and returned, containing indices of touched cells and handles of internal SAP boxes associated with our Multi-SAP-level object.

Removing an object

To remove an object, we use the list of touched cells given by the handle, and we simply remove the object from all the cells it was currently touching.

Updating an object

To update an object, we first rasterize the new bounds into the grid. The set of touched cells is compared to the set of *previously* touched cells, given by the handle. Then, depending on the comparison results, the appropriate action is performed:

- If a touched cell was not in the handle, it is a new cell and we add the object to its SAP

- If a touched cell was in the handle, the object is updated in this cell's SAP

- After all touched cells have been processed, object is removed from previously touched cells that have not been touched this frame

Updating an object this way, we do not need the AABBs around the internal SAPs, and we do not need to test the object's bounds against them.

d) Parallel version

Sharing the PM is the easiest way to implement the multi-SAP, but it might not be the most parallel-friendly version. A parallel version is one where each SAP is truly independent, not sharing anything, and all the insertions, updates or deletions to each of them can be performed in parallel – say with one core dedicated to each grid cell. This can be achieved relatively easily, like this:

- The PM is not shared. We really use an unmodified single-SAP in each grid cell. In a way, this is easier than the non-parallel version.

- (Serial part) First, objects are rasterized into the grid and necessary Add / Remove / Update operations are recorded (not performed) in different queues (one per grid cell).

- (Parallel part) Queues are processed and SAPs from different cells are all updated at the same time.

- (Serial part) A synchronization part is performed: PMs from all SAPs are merged together, duplicate pairs filtered out, results presented to users.

In practice, the synchronization part is very cheap, so more advanced approaches are usually not needed. Nonetheless it is possible to modify the update rules of each single SAP, to give the pairs a "SAP owner" and filter out duplicate pairs right away. This gets rid of the synchronization part, but the code becomes a lot more complex, uses more memory, and overall this approach is not worth the hassle.

Appendix A - Integers or floats comparisons

The SAP main loop spends a lot of time comparing end-point values. Unfortunately FPU comparisons are often quite slow, in particular on PC and Xbox 360. So, it is usually a good idea to use CPU comparisons instead.

One way to do that is to quantize the floating-point values and cast them to integer, in a way similar to how bounding boxes are quantized in *Opcode*. [10] However this is not a great approach because:

- You typically need to impose max bounds for your world to derive quantization coefficients. This is an un-necessary constraint that can be quite painful for users.

- Quantized boxes are not fully accurate (although conservative), leading to false positives (overlaps reported between pairs of objects that are not really touching)

The *Bullet* library [7] in particular, suffers from those problems (at least at the time of writing, i.e. up to Bullet 2.55)

A better way would be to simply read the floats as binary values, like in radix-sorting. However it is well known that simply reading floats as integers do not allow one to sort them correctly, when positive and negative values are involved. As described in [1], one has to "fix the wrong place" and "fix the wrong order" to make it work. In [1] we did that on-the-fly in the fourth radix pass, but there is a better way explained in [2] (*). The idea is simply to realize that the IEEE format is arbitrary, totally artificial, and we are free to transform it to something else if it fits our purposes. In particular, if we can apply a 1-to-1 mapping to our IEEE floats and end up with something "sort-compliant", this is all we really need since the values will otherwise not be used in any arithmetic operations. As it turns out, this mapping is really easy, we just "fix the wrong place" and "fix the wrong order" directly in the float itself. This is really the same as in [1], done a different way.

Negative floats are seen as big integer values (see [1]). In reality they are of course smaller than any positive numbers, so we just invert the signs to "fix the wrong place": positive floats have their sign bit set, negative floats have their sign bit cleared. If the float is negative, we also have to "fix the wrong order" by simply reverting the sequence of negative numbers, using a NOT. The final code to encode a floating-point value as a sortable integer is:

(*) …and suggested to me long before by Piotr Zambrzycki, although at the time I did not understand what he meant.

```
	inline_ udword EncodeFloat(const float newPos)
	{
		//we may need to check on -0 and 0
		//But it should make no practical difference.
		udword ir = IR(newPos);

		if(ir & 0x80000000) //negative?
			ir = ~ir;//reverse sequence of negative numbers
		else
			ir |= 0x80000000; // flip sign

		return ir;
	}
```

As a bonus and although we do not need it in the SAP implementation, the reverse transform is:

```
	inline_ float DecodeFloat(udword ir)
	{
		udword rv;

		if(ir & 0x80000000) //positive?
			rv = ir & ~0x80000000; //flip sign
		else
			rv = ~ir; //undo reversal

		return FR(rv);
	}
```

Once our floats are properly encoded, we can simply store them as integers in our *EndPoint* structures, and perform CPU comparisons on them as if we were using the original floating point values. It just works.

Appendix B - Using sentinels

The SAP main loops uses either linked lists or arrays. In both cases, we iterate over the structure (left or right), and we have to make sure we do not go past its start or end. So for example with linked lists the typical SAP update loop has a line like this:

```
EndPoint* tmp;
EndPoint* current;

while((tmp = current->mNext) && tmp->mValue < Limit)
```

The first part of the test is only needed to detect the end of the list, and it does not do anything "useful" as far as the SAP is concerned. So it would be good to get rid of it, and reduce the line to:

```
while(tmp->mValue < Limit)
```

This is exactly what the sentinels are there for. Sentinels are dummy, artificial end-points that do not belong to a real box. They are only here to mark the start and end of the sorted structures. Typically they contain a very large negative (for the start) or positive (for the end) value, like –FLT_MAX and FLT_MAX. This way the SAP loop is guaranteed to stop before passing the end of the collection, since any valid value coming from a real bounding box should be smaller than FLT_MAX. Just to make sure everything is clear, let us have a small example. This could be a list of end-points values, guarded by the two sentinels:

-FLT_MAX, -20, -14, -2, 4, 13, 28, FLT_MAX

If end-point 13 becomes greater, say increases to 50, its natural place should be the end of the array. With the original code, it would go past 28 and the code would stop because the end of the structure is reached. With sentinels, it stops because 50 is still smaller than FLT_MAX, and we end up with:

-FLT_MAX, -20, -14, -2, 4, 28, **50**, FLT_MAX

This is rather easy and an obvious way to optimize the SAP loops. There are really only a couple of minor issues with this approach:

- Each part of the SAP has to know about sentinels and handle them as a special case (for example when adding a new object, its new initial position is not *really* the end of the array, rather the before-last position, just before the sentinel)

- Incoming bounding boxes should not use FLT_MAX, otherwise the sentinels' logic does not work anymore. It is usually not an issue, except for infinite planes (whose bounds should then use FLT_MAX / 2 or something), and for objects falling through the ground towards infinity. If those maverick objects end up really far, so much that their bounding boxes end up with a +INF value in them, it might very well kill the SAP. Yes, those things happen.

Overall though, those are very minor problems that can be easily prevented, and using sentinels is pretty much always a good idea.

References:

[1] Pierre Terdiman, *Radix Sort Revisited*,
http://www.codercorner.com/RadixSortRevisited.htm

[2] Michael Herf, *Radix Tricks*, http://www.stereopsis.com/radix.html

[3] www.codercorner.com/PairManager.rar
http://www.codercorner.com/Ice.htm

[4] www.codercorner.com/BoxPruning.zip

[5] V-Collide collision detection library:
http://www.cs.sunysb.edu/~algorith/implement/V_COLLIDE/implement.shtml
http://www.cs.unc.edu/~geom/V_COLLIDE/

[6] Afra Zomorodian, *Fast Software for Box Intersection***.** with Herbert Edelsbrunner
International Journal of Computational Geometry and Applications, 2002 *(invited)*
16th ACM Symposium on Computational Geometry, Hong Kong, 2000

[7] Bullet library: http://www.bulletphysics.com/Bullet/

[8] Also called “Sort and Sweep”, see e.g:
http://www.gamedev.net/community/forums/topic.asp?topic_id=389367

[9] Erin Catto’s *Box 2D*: http://www.box2d.org/features.html

[10] Opcode library: http://www.codercorner.com/Opcode.htm