4.1

Comparison of VIPM Methods

Tom Forsyth, Mucky Foot Productions

tomf@muckyfoot.com

View-Independent Progressive Meshing (VIPM) has moved from the status of an interesting research project, to promising new technology, to sensible addition to all the best engines, and now into the Direct3D graphics API itself. It is now becoming almost required for any engine, and its inclusion in the Direct3DX library means that one form of VIPM is relatively easy to add.

However, in an effort to push the performance of VIPM, and in particular to drive the hardware as efficiently as possible, several new forms have been developed, each with their own tradeoffs and characteristics. This gem is intended as a guide to some of the more promising versions, and should help people decide which of the many variants to use in particular situations.

This gem does assume a basic familiarity with VIPM, and there is no space for a thorough introduction here. However, there are several good guides both in print and online. The two best known are Jan Svarovsky's gem in *Game Programming Gems* [SvarovskyOO] and Charles Bloom's Web site [BloomOl], both of which have excellent step-by-step guides to implementations of the "vanilla" VIPM method. All of the methods discussed here use the same basic collapse/split algorithm, but implement it in different ways.

Considerations

There are a few main points on which the various methods need to be judged. Different situations demand different choices, and the different ways each object type in a game is used may mean that different methods of VIPM are used. Things to consider include:

- **Global memory cost.** How much memory is taken up just by the mesh representation of the model? This memory is shared between all onscreen instances.
- **Instance memory cost.** How much memory is used for each instance of the object drawn onscreen? This memory is duplicated for each instance and cannot be shared.
- **Streaming or referenced memory cost.** This is the amount of data actually referenced on each frame. There may be a large amount of existing data for an

object that is mainly left on CD or swapped out to a hard drive by virtual memory. However, on each frame the actual amount of data referenced may be small, allowing the data to be streamed and/or handled efficiently by the virtual memory system. This is especially important for consoles that typically have limited memory.

- **CPU cost.** How many clock cycles does the algorithm take, in terms of user code? This includes both single-frame rendering costs and the cost of changing the level of detail from frame to frame.
- **API interface efficiency.** How many CPU cycles are used in driver and API interfaces getting data down to the graphics card?
- **Bus bandwidth.** How much data must be sent to the graphics card? On a PC, this means the AGP bus bandwidth.
- **Vertex—cache coherency.** Modern graphics cards try to fetch, transform, and light each vertex only once, even though the vertex will be used by multiple triangles. To do this, they have a vertex cache that holds the most recently used vertices, and applications need to try to use vertices in this cache as often as possible to get maximum performance. An algorithm that uses more triangles than another may still be faster because it has a higher vertex cache hit rate.

Vertex cache coherency will be quoted in terms of the number of vertices loaded or processed per triangle drawn, or "vertices per triangle." Current triangle reordering algorithms for static (i.e., non-VIPM) meshes using modern vertex caches of around 16 entries can get numbers down to around 0.65. For an example, see [Hoppe99]. This gives suitable benchmark figures to compare efficiencies when the mesh is converted to a VIPM one. Also note that when calculating the vertices per triangle using triangle strips, only drawn triangles should be counted, not degenerate ones. The degenerate triangles are a necessary evil—they add nothing to the scene.

Algorithms that are good at streaming allow the application to draw huge worlds that are mostly stored on disk, and to degrade image quality gracefully if the streaming of data hits a limit somewhere along the way, such as available disk bandwidth or available memory on the machine.

This also helps systems with virtual memory; if the data is accessed linearly, the virtual memory manager can swap out data that has yet to be accessed, or has not been accessed for a long time. Static data can be optimized even further and made into a read-only memory-mapped file. This also ensures that irritating "loading level" messages are no more tedious than absolutely necessary. The object data does not all need to be loaded at the beginning; the player can start playing the level with low-resolution data and as the detailed models are needed, they will be loaded.

All the methods discussed here are based around implementations of the same fundamental algorithm. Single operations are done that collapse a single vertex onto another vertex along one of its triangle edges. No new "average" vertex is generated, and no collapses between vertices that do not share an edge are allowed. These are worth looking into; however, the current consensus is that they involve a higher run-

time cost for equivalent error levels on most current hardware. Of course, things change, and new algorithms are always being invented.

A note on the terminology used: The *resolution* of a mesh is proportional to the number of triangles in it. Thus, a high-resolution mesh undergoes edge collapses and becomes a lower-resolution mesh. The opposite of an edge collapse is an edge *split,* where a single vertex splits into two separate vertices. For a given edge collapse, there is a *kept* vertex and a *binned* vertex. The binned vertex is not used in any lower-resolution meshes, whereas the kept vertex is. For a given edge collapse, there are two types of triangles. Those that use the edge being collapsed will not be in any lower-resolution mesh and are binned. For a typical collapse, there are two binned triangles, although there may be more or less for complex mesh topologies. Those that are not binned but use the binned vertex are "changed" triangles, and changed so that they use the kept vertex instead of the binned vertex. When performing an edge split, the previously binned vertex and triangles are "new," although they are often still called binned because there are typically no split data structures, just collapse data structures that are done in reverse. Most of the perspective is in the collapsing direction, so words like *first, next, before,* and *after* are used assuming collapses from a high-triangle mesh to a low-triangle mesh. Again, splits are done by undoing collapses.

This gem will be talking in a very PC and DirectX-centric way about CPUs, AGP buses, graphics cards ("the card"), system/video/AGP memory, index, and vertex buffers. This is generally just a convenience—most consoles have equivalent units and concepts. Where there is a significant difference, it will be highlighted. The one term that may be unfamiliar to the reader is the *AGP bus;* this is the bus between the main system memory (and the CPU) and the graphics card with its memory. There are various speeds, but this bus is typically capable of around 500Mbytes/sec, which makes it considerably smaller than the buses between system memory and the CPU, and between the graphics chip and its video memory. Some consoles have a similar bottleneck; others use a unified memory scheme that avoids it. In many cases, this is the limiting factor in PC graphics.

Vanilla VIPM

This is the best-known version of VIPM, and the version used by the Direct3DX8 library. It has a global list of static vertices, arranged in order from last binned to first binned. Each time a collapse is done, the vertex being binned by the collapse is the one at the end of the list, and the number of vertices used is decremented by one. This ensures that the used vertices are always in a single continuous block at the start of the vertex buffer, which means that linear software T&L pipelines always process only the vertices in use.

The triangles are also ordered from last binned to first binned. Each edge collapse generally removes two triangles, although they may actually remove anywhere from zero upward for meshes with complex topologies.

Triangles that are not binned but are changed during a collapse simply have the index to the binned vertex changed to that of the kept vertex. Since the index list changes as the level of detail changes, the triangle index buffer is stored as per-instance data. The index buffer is comprised of indexed triangle lists (each triangle defined by three separate indices), rather than indexed triangle strips.

Each record of collapse data has the following format:

```
struct VanillaCollapseRecord
{
    // The offset of the vertex that doesn't vanish/appear.
    unsigned short wKeptVert;
    // Number of tris removed/added.
    unsigned char  bNumTris;
    // How many entries in wIndex0ffset[] .
    unsigned char  bNumChanges;
    // How many entries in wIndex0ffset[] in the previous action.
    unsigned char  bPrevNumChanges;
    // Packing to get correct short alignment.
    unsigned char  bPadding[1];

    // The offsets of the indices to change.
    // This will be of actual length bNumChanges,
    // then immediately after in memory will be the next record.
    unsigned short wIndex0ffset[] ;
};
```

This structure is not a fixed length — wIndexOf f set [ ] grows to the number of vertices that need changing. This complicates the access functions slightly, but ensures that when performing collapses or splits, all the collapse data is in sequential memory addresses, which allows cache lines and cache prefetching algorithms to work .efficiently. It also allows the application to stream or demand-load the collapse data off a disk very easily. Because it is static and global, it can also be made into a read-only memory-mapped file, which under many operating systems is extremely efficient.

Although at first glance bPrevNumChanges doesn't seem to be needed for collapses, it is needed when doing splits and going back up the list — the number of wIndexOff - set[] entries in the previous structure is needed so they can be skipped over. Although this makes for convoluted-looking C, the assembly code produced is actually very simple.

To perform a collapse, the number of vertices used is decremented since the binned vertex is always the one on the end. The number of triangles is reduced by bNumTris; again, the binned triangles are always the ones on the end of the list.

The changed triangles all need to be redirected to use the kept vertex instead of the binned one. The offsets of the indices that refer to the binned point are held in wIndexOffset[]. Each one references an index that needs to be changed from the binned vertex's index (which will always be the last one) to the kept vertex's index — wKeptVert.

```
VanillaCollapseRecord *pVCRCur = the current collapse;
iCurNumVerts-;
iCurNumTris -= pVCRCur->bNumTris;

unsigned short *pwlndices;
// Get the pointer to the instance index buffer.
p!ndexBuffer->Lock ( &pwlndices );
for ( int i = 0; i < pVCRCur->bNumChanges; i++ )
{
    ASSERT ( pwIndices[pVCRCur->w!ndexOffset[i]] ==
        (unsigned shortJiCurNumVerts );
    pwIndices[pVCRCur->w!ndexOffset[i]] = pVCRCur->wKeptVert;
}
// Give the index buffer back to the hardware.
p!ndexBuffer->Unlock();
// Remember, it's not a simple ++
// (though the operator could be overloaded).
pVCRCur = pVCRCur->Next();
```

Note that reading from hardware index buffers can be a bad idea on some architectures, so be careful of exactly what that ASSERT () is doing—it is mainly for illustration purposes (Figure 4.1.1).

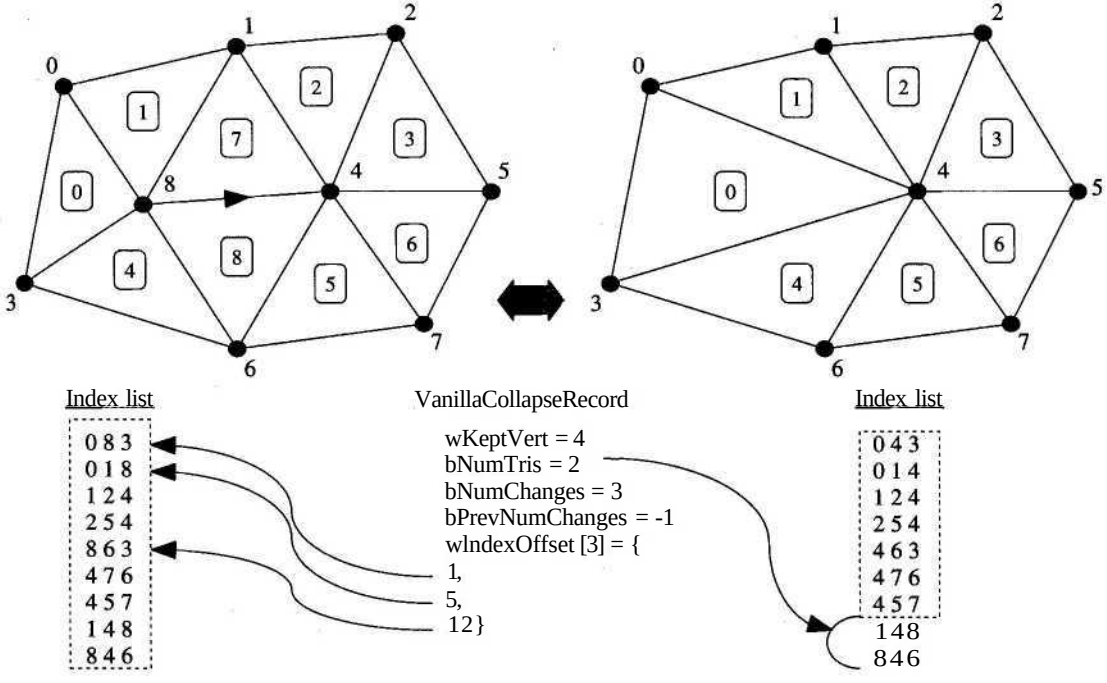

FIGURE 4.1.1 *An edge collapse with before and after index lists and the* VanillaCollapseRecord.

Doing a split is simply a matter of reversing the process.

```
VanillaCollapseRecord *pVCRCur = the current collapse;
pVCRCur = pVCRCur->Prev();
unsigned short *pwIndices;
pIndexBuffer->Lock ( &pwIndices );
for ( int i = 0; i < pVCRCur->bNumChanges; i++ )
{
     ASSERT ( pwIndices[pVCRCur->wIndexOffset[i]] ==
          pVCRCur->wKeptVert );
     pwIndices[pVCRCur->wIndexOffset[i]] =
          (unsigned shortJiCurNumVerts;
}
iCurNumTris += pVCRCur->bNumTris;
iCurNumVerts++;

pIndexBuffer->Unlock();
```

Note that in practice, and for arbitrary historical reasons, in the sample code the VertexCollapseRecords are stored last first, so the `Prev()` and `Next()` calls are swapped.

Vanilla VIPM is simple, easy to code, and has decent speed. It should probably be the first version used for any evaluation of VIPM, because it is so simple, and even this will give good scalability, streaming, and so on.

The good thing about vanilla VIPM is that it streams very well. Collapse information and index buffer data is completely linear in memory and ordered by collapse, so implementing a streaming system with fallbacks for when data is not immediately available is extremely easy.

However, there are many bad things about vanilla VIPM. Vertex cache coherency is poor. Because triangle order is strictly determined by collapse order, there is no way to reorder triangles for better vertex caching.

Another problem is the relatively large per-instance memory use. The whole index data chunk needs to be replicated for each instance. This can be reduced by only allocating as many indices as are actually currently being used, and growing or shrinking as needed (along with a bit of hysteresis to prevent calling `malice()` and `free()` all the time), but it is still large if there are lots of objects onscreen.

Finally, vanilla VIPM only works with indexed triangle lists, which can be a poor choice for hardware that prefers strips.

Skip Strips

Skip strips is a slightly overloaded name. It was borrowed from a paper on View-Dependent Progressive Meshing (VDPM) [El-Sana99]. VDPM is significantly more complex and requires some fairly extensive data structures to achieve good efficiency, and a *skip list* is one of those data structures. However, the section that inspired this VIPM method was the bit that noted that to bin a triangle, it does not have to fall off the end of the index list, as in vanilla. There is not much wrong with simply making it degenerate by moving one of its vertices (usually the binned vertex) to another one

(usually the kept vertex), and leaving it in the list of drawn triangles. Hardware is very good at spotting degenerate triangles, and throws them away very quickly without trying to draw any pixels.

This means that the order of triangles is no longer determined by collapse order; they can be ordered using some other criteria. The cunning thing that the original skip strips paper pointed out is that triangles can now be ordered into strip order, and indeed converted into strips. This is great for hardware that prefers its data in strip order. Since this VIPM method was inspired by the paper, it inherited the name, despite it being somewhat inaccurate.

The ability to reorder triangles increases vertex cache coherency. Strips are naturally good at this—they have an implicit 1.0 vertices per triangle efficiency (for long strips with no degenerates), and with the right ordering and a decent-sized vertex cache, they can get much lower values.

One cunning thing about the implementation is that the collapse/split routines and data structures are virtually identical to vanilla VIPM. The only change is that the number of drawn triangles does not change with collapses and splits. Triangles simply become degenerate; they do not fall off the end of the list.

However, this shows a big problem with skip strips. After many collapses, there are many degenerate triangles in the list. Although they are rejected by the hardware quickly, they still take some time to reject, and their index data still has to be sent to the card. This eats into the bus bandwidth, and lowers the visible triangle throughput in triangles/second.

After many collapses, the vertex cache efficiency also drops. The nice neat strips will have been bent and broken by the collapses, which disrupts the cache efficiency. Moreover, as triangles become degenerate, the number of indices referring to one of the remaining vertices increases. A collapse that bins that vertex must change all the indices that refer to it, including the degenerate triangles. Therefore, the more collapses that are done, the more expensive each collapse becomes, because the size of `wIndexOffsett]` grows. This does not scale with the number of triangles drawn, which is no good since that is the whole point of VIPM—things at lower detail should take less time to render.

Multilevel Skip Strips

Fortunately, there is a solution to most of skip strip's woes. After a certain number of collapses, simply stop, take the current geometry with all of its collapses done, throw away the degenerate triangles, and start making a completely new skip strip from scratch. Continue collapses with this new skip strip until it too becomes inefficient, and so on.

When creating each new skip strip level, all of the degenerate triangles are thrown away, which reduces the number of triangles (both visible and degenerate) that are sent to the card. The triangles are also reordered to make lists that are again vertex-cache optimal. New collapses don't need to change lots of degenerate triangle indices

each time, each instance only needs to copy the skip strip level that it actually uses, and they become shorter with decreasing detail.

The different index lists can be stored globally since when switching to a new list, a new copy is taken and then refined with collapses to exactly the number of triangles wanted. Therefore, the fact that there are now multiple index lists is not too bad—its global data. This also restores some of the nice streaming friendliness that the vanilla method has. The granularity is a bit coarser; the whole of an index list must be grabbed before anything can be rendered using that level, but at least it's no longer an all-or-nothing thing, and the lower-resolution index lists are actually very small.

For a bit more efficiency, two versions of the index lists can be stored in global space: fully collapsed (before switching to a lower-resolution list, that is) and fully uncollapsed. This means that a single-collapse oscillation across the boundary between two index lists is still fairly efficient. If only the uncollapsed versions are held, each time the level of detail increases, the higher-resolution index list must be copied, and then all of its collapses need to be performed to draw the next frame. Having the collapsed versions stored as well means that a change in the level of detail of *n* collapses only actually requires *n* collapses (and sometimes fewer).

The actual collapse/split code and structures are the same as for standard skip strips, except that there is a global array of structures holding the premade index lists, the collapse lists for each one, and the number of collapses in each. Before doing any collapses or splits, the code checks to see if it needs to change levels, and if so, copies the new level's index list and starts doing collapses/splits until it reaches the right level of detail within that level.

So, this has fixed all the bad things about skip strips when compared to vanilla in exchange for an increase in global (but easily streamed or swapped) memory.

Skip strips also have an equivalent using triangle lists instead of triangle strips. The principle is exactly the same, but use a different primitive. Some algorithms require lists rather than strips, and some vertex cache routines can obtain slightly higher caching rates with lists than strips. No separate implementation was done in the sample code, because they are so similar.

Mixed-Mode VIPM

One of the problems with the types of VIPM mentioned so far is that the whole index list needs to be copied for each instance of the object. This can be quite a burden in some cases, especially on machines with limited memory, notably consoles, where everything has to be shoehorned into memory that is usually half the size that the programmers would like, even before VIPM is mentioned. It would be excellent if some of this index list could be moved to global (i.e., static and shared between instances) memory instead of having to be copied for each instance.

On a multilevel skip strip, many of the triangles are not affected, even when that level is fully collapsed. Therefore, there is no need to copy those triangles per instance;

they can be global and shared between instances. In fact, for this algorithm, indexed lists are used—the indexed strip case will be discussed later as a variant. At each level, the triangles are split into four lists:

- The triangles that are not affected by any collapses.
- The triangles that are binned by collapses, but not modified by any before they are binned.
- The triangles that are modified by collapses, but not binned.
- The triangles that are first modified by one or more collapses and then binned.

Lists 2 and 4 are each sorted by bin order, just as for vanilla VIPM. Lists 1 and *3* are sorted into whatever order gives the highest vertex cache efficiency. Then list 2 is appended to list 1, and the combined list is put into a global index buffer that is static and shared by all instances. List 4 is appended to list 3, and the combined dynamic list is copied into instances when they use that level. This list is then modified at runtime using exactly the same modification algorithm as vanilla VIPM.

To draw the mesh, the required collapses and splits are done to the dynamic per-instance list, and the list is drawn. Then the associated level's static list is drawn, with the only modification being that the number of triangles drawn will change as static triangles are collapsed.

The code and structures needed are based on the multilevel skip list, except that for each level there are two lists: the copied dynamic one and the shared static one. The other change is that there are two triangle counts, one for each list, and a collapse may alter either or both of these numbers. Therefore, the `bNumTris` member is replaced by `bNumStaticTris` and `bNumDynamicTris`, and the appropriate increments and decrements are added.

This means that a large proportion of each mesh is being drawn from a static index buffer that is tuned for vertex cache coherency (list 1). It is not quite as good as it could be, since the triangles in this list only make up part of the object. There will be "holes" in the mesh where triangles have been moved to the other three lists, and this decreases both the maximum and the actual vertex per-triangle numbers that are obtained. Some of the dynamic buffer is also ordered for optimal vertex cache behavior (list 3), although collapses can interfere with this efficiency, and the mesh for list 3 is usually far from usefully connected, so there is a limit to what any reordering can do.

Like all multilevel methods, it is streaming friendly; although in this case, since the lists are ordered by collapse order, the granularity is even finer at the triangle level, not just the list level. Whether this is terribly exciting is a different question—the finer control is probably not going to make much of a difference in performance.

This does require two `DrawIndexedPrimitive()` calls to Direct3D (or equivalent API), although on most platforms, this is not a bottleneck and does not affect rendering speed. It may be important for very low-triangle meshes, and for these, switching to another method may be appropriate.

Mixed-Mode Skip Strips

Mixed-mode skip strips are identical to mixed-mode lists, except that strips are used, and instead of the dynamic list being done with vanilla VIPM, it is done using the skip strips algorithm. As with skip strips, using strips means that ordering by collapse order is too inefficient, and diis means that list 2 triangles now have to be binned by being made degenerate. This forces them to become dynamic instead of static, and they join lists 3 and 4. The triangles from these three lists are merged and treated as a skip strips —reordered for optimal vertex cache efficiency, copied for each instance, and modified by collapse information.

The disadvantages with this method are that there is now more data being copied for each instance, and because the triangles are ordered by strip order and not collapse order, triangles cannot be binned entirely by simply dropping them off the end of the index list. However, both these factors are only mildly worse than the list version, and if the hardware needs to be fed strips, this is still an excellent method.

Sliding Window

Sliding window VIPM introduces the idea of fully static and global index buffers, with no editing of indices, and therefore a tiny amount of per-instance memory.

Sliding window notes that when a collapse happens, there are two classes of triangles: binned triangles and modified triangles. However, there is no real need for the modified triangles to actually be at the same physical position in the index buffer before and after the collapse. The old version of the triangles could simply drop off the end of the index buffer along with the binned triangles, and the new versions added on at the other end.

Therefore, instead of an example collapse binning two triangles and editing three others, it actually bins five triangles and adds three new ones. Both operations are performed by just changing the first and last indices used for rendering—sliding a "rendering window" along the index buffer (Figure 4.1.2).

The index buffer is split into three sections. At the beginning are triangles added as a result of changes, in reverse collapse order. In the middle are triangles not affected by collapses, in any (vertex cache-optimal) order. At the end are triangles binned or changed by collapses, again ordered in reverse collapse order—first collapse at the end. Note that a triangle modified as the result of a collapse cannot then be involved (either binned or changed) in another collapse. To be modified by a second collapse would mean that triangle would have to fall off the end of the index buffer. It has already been added to the beginning so it cannot then also fall off the end—the chance of the ordering being just right to allow this are incredibly slim.

Once a triangle has been modified by a collapse, the only way it can be involved in another collapse is if a new index buffer is started that has all the same triangles as the previous (collapsed) one. The ordering of this new one is not constrained by the previous collapses, and so can be sorted by new collapses. Again, the multilevel con-

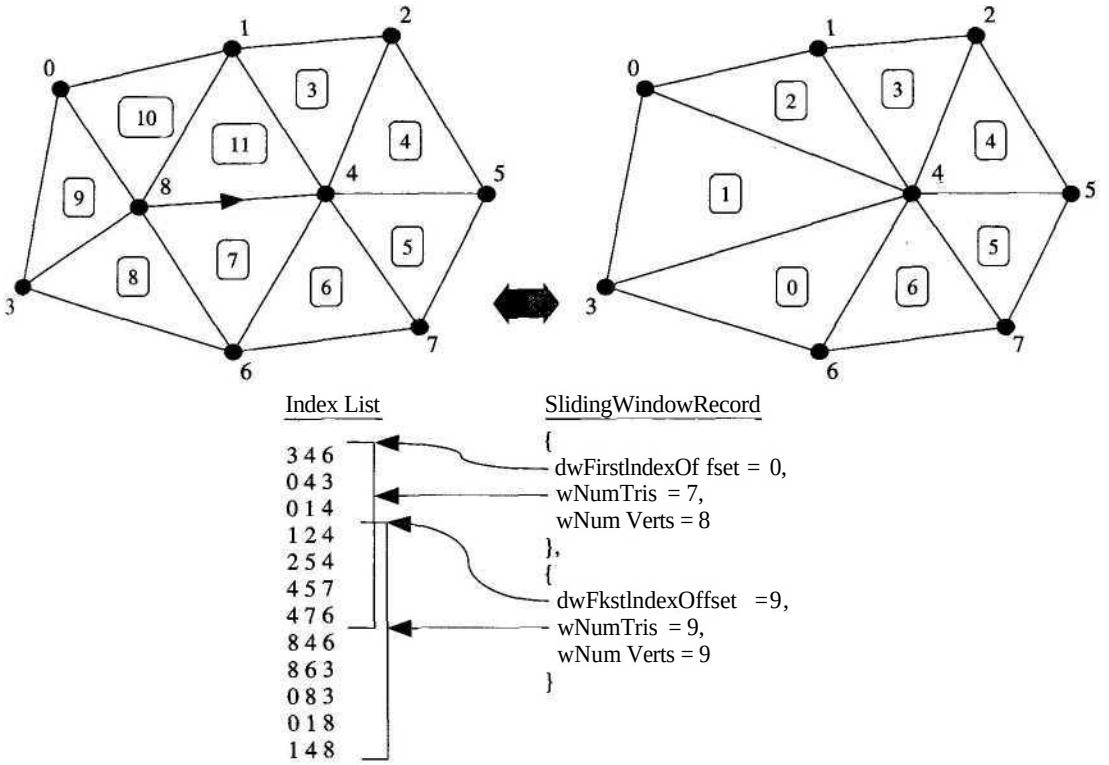

FIGURE **4.1.2** *A collapse showing the index list and the two windows.*

cept is used, but in this case because further collapses cannot happen without it, not simply for efficiency.

The problem with this at face value is that algorithms such as QEM give an ordering for collapses. If this ordering is strictly followed, the QEM frequently wants to do a new collapse that involves a triangle that has already been modified by a previous collapse. This forces a new level to be made, and the index buffer needs to be copied. Since only a few collapses have been done, this copy is almost as big as the original. If only a few collapses are done before having to make a copy, the memory used for all the index buffers is going to be huge.

However, there is actually no need to strictly follow the order of collapses that QEM decides. Progressive meshing is not an exact science, since it ignores everything but the distance of the camera from the object, and the whole point is to simply be "good enough" to fool the eye. Therefore, diere is no real need to precisely follow the collapse order that QEM decides—it can be manipulated a bit.

The way to do this is to follow the QEM collapse order until it decides to do a collapse that involves triangles that have already been modified. Doing this collapse would force a new level, and so this is put off for as long as possible. For the moment

this collapse *is* ignored, and the best one that can be done without creating a new level is found. The errors of the two collapses are compared, and if they are within a certain tolerance, then doing them out of strict order is not going to affect visual quality all that much, and the collapse that will not force a new level is done.

Once the difference in error levels is too great, then doing the wrong collapse first is going to affect image quality significantly, and the algorithm bites the bullet and creates a new level. There have now been a decent number of collapses done before this copy happens, the triangle count has been significantly reduced, and thus far, fewer levels are needed before they collapse down to the minimum level of detail.

The sample code uses a fairly small tolerance level of 10 percent of the average collapse error, and even this small tolerance reduces the number of levels dramatically. Using a larger error tolerance can reduce the memory use even more, although only to a point. After a while, the algorithm simply runs out of triangles that have not already been involved in a collapse. Most meshes can only lose around 20 percent of their triangles before this happens, but this still keeps memory use at sensible levels.

Since no runtime modification is made to the index or vertex lists, all the data can be made global, and there is almost zero per-instance memory use. There is also almost zero CPU time used to change level of detail—each time, a simple table look-up is made to decide the index list to use, the start and end index to draw from that index list, and how many vertices are used. In practice, the index lists are concatenated together, so that the first index also implies the index list to use. The table is composed of this structure:

```
struct SlidingWindowRecord
{
    unsigned int   dwFirstlndexOffset;
    unsigned short wNumTris;
    unsigned short wNumVerts;
};
```

Although the number of triangles and vertices is known to be less than 64k (this is a limit in all currently known hardware), because the index list is a concatenation of many lists, it may easily be greater than 64k indices in length, so 32 bits are required for it. This does mean that the structure is nicely padded to 8-byte alignment, though. The rendering code is amazingly simple:

```
SlidingWindowRecord &pswr = swrRecords[iLoD];
d3ddevice->DrawIndexedPrimitive (
    D3DPT_TRIANGLELIST,  // Primitive type
    0,                   // First used vertex
    pswr->wNumVerts,     // Number of used vertices
    pswr->dwFirst!ndexOffset,// First index
    pswr->wNumTris );    // Number of triangles
```

There is no code to do splits or collapses as with all the other methods—the current level of detail is just looked up in die SlidingWindowRecord table each time the

object is rendered. This also means that with hardware transform and lighting cards, the CPU time required to render objects is fixed and constant per object, whatever their level of detail. The phrase "constant time" is always a good one to find lurking in any algorithm.

The major problem with sliding window VIPM is that it forces the ordering of the triangles at the beginning and end of each level's index lists. This has two effects: it makes strips hard to use—only triangle lists really handle fixed ordering well—and vertex cache efficiency is affected.

Fortunately, it is not as bad as it first seems. When an edge collapse is performed, all of the triangles that use the binned vertex are removed, so they all go on the end of the triangle list. This is typically from five to seven triangles, and they form a triangle fan around the binned vertex. Then the new versions of the triangles are added. These need to go together at the beginning of the index list, there are typically three to five of them, and they form a triangle fan around the kept vertex. These fans can be ordered within themselves to get the best cache coherency. The middle of the index list that is not affected, and thus has no set order, can be reordered for the vertex cache. This gets much better cache coherency than vanilla. Although it is still quite a bit short of the theoretical ideal, it is not unreasonably poor.

Vertex cache coherency can be raised by having a larger middle index list section in each level—by having fewer collapses per level. This takes more memory, but the extra performance may be worth it, especially as it is global memory.

Hardware that requires strips rather than lists can still use this method, although it does require many degenerate triangles to join the different parts. In practice, this does not increase the number of indices required, it actually reduces it—strips have one index per triangle, compared to a list's three. The vertex cache efficiency per drawn triangle is exactly the same. The raw triangle throughput is increased a lot (roughly doubled), but since all of these extra triangles are just degenerate, most hardware will reject them very quickly. If there is a choice, which of the two primitives used depends on whether the hardware is limited by index bandwidth (in which case, strips are optimal) or triangle throughput (in which case, lists are optimal).

Summary

VIPM seems to be coming of age. It is now mainstream, it has been incorporated into a major API, and for discrete objects it has beaten off VDPM and static level of detail methods for the most visual bang for the CPU buck (although it is worth noting that VDPM methods are still challengers for large landscapes, especially regular-height-field ones). In addition, it now has a plethora of methods from which to choose, each with its own advantages and disadvantages. Innovation certainly won't stop there—there are already some interesting paths for future investigation, but this roundup should give a fairly good guide to some of the issues and options when choosing which VIPM method to implement.

Table 4.1.1 Summary of Strengths and Weaknesses of Each VIPM Method

	Vanilla	Skip Strips	Mixed-Mode	Sliding Window
Vertex cache use	Poor	Excellent	Good	Good
Global memory use	Low	Medium	Medium	High
Instance memory use	High	High	Medium	Low
LoD-change CPU cost	Medium	Medium	Medium	Tiny
API efficiency	Good	Good	Good	Excellent
List efficiency	Poor	Excellent	Good	Good

Table 4.1.1 shows the results of each method with their relative strengths and weaknesses. Note that "skip strips" refers to multilevel skip strips—the single-level version is not actually a sensible method in practice, for the reasons given.

References

[SvarovskyOO] Svarovsky, Jan, "View-Independent Progressive Meshing," *Game Programming Gems,* Charles River Media, 2000, pp. 454-464.

[BloomOl] Bloom, Charles, VIPM tutorial, and various VIPM thoughts gathered from many sources, www.cbloom.com/3d/index.html.

[Hoppe99] Hoppe, Hugues, "Optimization of Mesh Locality for Transparent Vertex Caching," Computer Graphics (SIGGRAPH 1999 proceedings) pp. 269-276. See also www.research.microsoft.com/-hoppe/.

[El-Sana99] J. El-Sana, F. Evans, A. Varshney, S. Skiena, E. Azanli, "Efficiently Computing and Updating Triangle Strips for View-Dependent Rendering," *The Journal of Computer Aided Design,* vol. 32, no. 13, pp. 753-772. See also www.cs.bgu.ac.il/-el-sana/publication.html.