DOOM 3 BFG Technical Note

J.M.P. van Waveren

October 2, 2013

Abstract

This technical note describes some of the performance issues encountered during the development of DOOM 3 BFG.

1. Introduction

DOOM 3 BFG Edition is a remastered version of DOOM 3, the third installment of the genre defining franchise. This remastered version extends and enhances the full experience of the original DOOM 3, and makes the game available on all current major platforms (PC, Xbox 360, PlayStation 3) with a silky smooth frame rate and highly responsive controls.

While it may seem that DOOM 3 BFG is a fairly straight forward port of the original game, significant effort went into re-optimizing the game for current hardware. At the time of release (2004), the original DOOM 3 displayed about 20 frames per second at a resolution of 640x480 with medium to low quality settings on the minimum-specification PC. The goal for DOOM 3 BFG was to run at a solid 60 frames per second at a resolution of at least 1280x720 with the highest quality settings on all current (2012) hardware. That is 3 times the number of rendered pixels that are updated 3 times more frequently. Of course 1280x720 is not even considered a high resolution anymore. Everyone wants to run at a higher resolution with some form of MSAA which easily adds another couple of factors to the equation. In other words, the goal for DOOM 3 BFG was to run *at least* 10 times faster on current hardware. Obviously computer hardware improved significantly since the release of the original DOOM 3. However, a factor of 10 or more performance improvement turned out to not come for free with 8 years of improvements in computer hardware (8 years on the PC while the Xbox 360 and PlayStation 3 benefit from only a couple of years of hardware improvements).

Central Processing Units (CPUs) have become a lot more efficient at processing instructions from a single thread of execution. In more technical terms, the number of clock cycles per instruction (CPI) has gone down. However, the operating frequency of CPUs has not gone up much, if at all. In other words, while the number of instructions that can be executed per clock cycle has gone up, the number of clock cycles per second has not increased. Instead most CPUs now implement multiple cores that can execute code in parallel and on various processors each core can simultaneously advance multiple threads of execution. This translates into a large potential performance improvement, but only if software is written to take advantage of multiple cores by implementing multiple threads of execution that can run in parallel. Unfortunately, the original DOOM 3 [10] is predominantly single threaded and all the heavy lifting is done on a single thread of execution.

The performance of Graphics Processing Units (GPUs) has increased dramatically. GPUs have always been designed to be highly parallel machines and over time there has been a significant increase in the number of threads of execution. GPUs are complex machines with various hard-wired high performance components. Interestingly, some of these components were just coming online at the time of the original DOOM 3 release.

The coarse Z-Cull / Hierarchical-Z is one such component which has the potential to dramatically improve rendering performance. While DOOM 3 BFG benefits significantly from this component, taking full advantage of the coarse Z-Cull / Hierarchical-Z turns out to be an interesting challenge when rendering stencil shadow volumes, one of the important rendering features of DOOM 3 [1, 2].

While the potential CPU and GPU performance has improved significantly over the years, the performance of random access memory (RAM) has not improved at the same rate. In particular, memory bandwidth and memory latency have not improved as much. As a result the number of instructions that can be executed per memory access has increased. This change presents another set of challenges and forced many of the optimizations from the original DOOM 3 [3, 4, 5, 6, 7, 8, 9] to be redone with this new balance between compute and memory access in mind.

2. Memory Constraints

In some of the worst case scenes, DOOM 3 BFG suffers from cache thrashing on PC systems with a modest amount of cache. In particular, there can be a bad interaction between the renderer backend/driver and dynamic shadow volume construction code (all running in different threads). This really is not anything new because cache thrashing and memory bandwidth were a problem in the original DOOM 3 as well. Even though significant improvements were made for DOOM 3 BFG, the game now also performs significantly more CPU work while still touching and generating a lot of data, partly to offload the GPU and to avoid Z-Cull / Hi-Z issues on some graphics hardware.

Unlike the original DOOM 3, nothing is done lazily anymore (except for generating animation frames, no more flags and tests like: "calculate this right now if it has not been calculated yet"). All the high performance CPU work now follows the streaming programming model. For dynamic shadow volumes the engine now culls triangles to the light volume, both to reduce the number of shadow volume triangles and to reduce the number of triangles that are redrawn for each light pass. The engine now also performs a precise test to determine whether the view intersects or is inside a shadow volume. This precise test is important to significantly reduce the number of cases where shadow volumes have to be rendered with Z-fail [1] because Z-fail rendering is significantly slower on a various graphics hardware (in particular at high resolutions). For this precise inside test a line-versus-expanded-triangle intersection test is performed for every single shadow volume near cap triangle where the line goes from the view origin to the light origin. The shadow volume is also transformed into clip space and the polygons are clipped to the view in homogeneous space to calculate very tight depth bounds. In particular in the cases where shadow volumes do have to be rendered with Z-fail, the depth bounds are used to get more Z-Cull / Hi-Z benefit (at least on the hardware that supports the depth bounds test).

Skinning for rendering is now done on the GPU for both the visible meshes and shadow volumes. Unlike the original DOOM 3, all animated vertices are now in static GPU buffers and vertices are never copied to GPU memory at run-time. However, a lot of source data is used and a lot of triangle indices are generated on the CPU for both shadow volumes and to be able to render just those triangles that are inside a light volume. All indices are written to a single mapped index buffer using the _mm_stream_si128() intrinsic. To generate these indices and to perform a precise inside test, the code pulls in a lot of source data (whole triangle meshes in CPU memory).

In other words, the game crunches through, and generates a lot of data on the CPU. For instance, during the Mars City 1 intro sequence, the game may go up to using the following amounts of memory per frame, just for shadow volume construction/setup and light triangle culling:

6.0 MB worth of input data for shadow volume construction/setup and light triangle culling.
2.5 MB worth of generated shadow volume and culled light triangle indices.

Keep in mind that the game code and the graphics driver are running in parallel overlapped with the shadow volume construction/setup and light triangle culling. The game code and the graphics driver are typically very good at thrashing the CPU cache on their own without outside help.

The following code was used to verify that the index buffer returned by the graphics driver lives on write-combined memory, at least in the use cases in DOOM 3 BFG:

```
bool IsWriteCombined( void * base ) {
	MEMORY_BASIC_INFORMATION info;
	SIZE_T size = VirtualQueryEx( GetCurrentProcess(), base, &info, sizeof( info ) );
	if ( size == 0 ) {
		DWORD error = GetLastError();
		error = error;
		return false;
	}
	bool isWriteCombined = ( ( info.AllocationProtect & PAGE_WRITECOMBINE ) != 0 );
	return isWriteCombined;
}

void * buffer = glMapBufferRange( GL_ARRAY_BUFFER_ARB, 0, GetAllocedSize(),
				GL_MAP_WRITE_BIT |
				GL_MAP_INVALIDATE_RANGE_BIT |
				GL_MAP_UNSYNCHRONIZED_BIT );

assert( IsWriteCombined( buffer ) );
```

In other words, the game is not thrashing the cache when writing out shadow volume and culled light triangle indices. However, the source data for the shadow volume construction/setup and light triangle culling lives on CPU cacheable memory. Overlapped streaming through a small window (usually around 4kB) is used for all source data, for both shadow volume construction/setup and light triangle culling. Everything is optimized to perform contiguous streaming where as little memory is touched as possible and memory is read as few times as possible.

In the cases where the source data is used once, and only once, during a game frame, large scale cache pollution can be avoided by using the _mm_clflush() intrinsic to immediately flush the small window of source data from the cache after it has been processed. Unfortunately, it is a bad idea to do this when the same source data is used multiple times during a game frame from different threads (for instance when a mesh interacts with multiple lights and multiple shadow volumes are constructed in parallel). It is a bad idea to flush the source data from the cache from one thread, while another thread may potentially be using the same source data at pretty much the same time. It is interesting how a shared cache between threads/cores is actually not helping performance in this case.

What this all comes down to is that even when using the streaming programming model on a high end x86 CPU, it is hard to achieve similar throughput as the Cell processor. Of course the Cell SPUs are hard to program because all the code has to be optimized pretty much from the start. Both the code and data has to fit in 256 kB of memory and it is desirable for each of the programs (or jobs) plus associated data to fit in less than 128 kB of memory to allow overlapped input and output streaming. The Cell SPU is also SIMD only so it is desirable to write SIMD code from the start to reduce the size of the binary code because having the compiler map scalar code to SIMD introduces a reasonable amount of overhead. In other words, if the programmer hours are spent then it is possible to achieve some pretty impressive throughput (for what the Cell processor is anyway, it is not a GPU).

On x86 there is the SSE4.1 intrinsic _mm_stream_load_si128(). This intrinsic loads data through "read combine" buffers instead of the cache. Unfortunately this intrinsic only works on write-combined memory. Once memory is set write-combined, all code better use this intrinsic otherwise all performance bets are off. Streaming loads through _mm_stream_load_si128() are predictive so it is possible to achieve very close to maximum theoretical bandwidth. In other words, the _mm_stream_load_si128() intrinsic provides a lot of what can be done with a DMA controller as long as the code does not require a gather / scatter operation which is something the DMA controller for a Cell SPU can actually do reasonably well. Aside from gather / scatter operations, for which an actual DMA controller is probably desired, it would be really nice if the _mm_stream_load_si128() intrinsic also worked on cacheable memory (just like _mm_stream_si128()). This would allow cacheable memory to be accessed both through the cache and the "read-combine" buffers. Obviously there is no coherency between cached reads/writes and "read-combined" reads but that is not a concern for data that is effectively read-only at run-time. Read-only run-time data is very common in many games. Think of collision detection data, sound occlusion data, render geometry construction and culling source data etc. Of course the SSE4.1 instruction is not available on all CPUs that are targeted by DOOM 3 BFG so this is still only a solution for newer CPUs that actually support SSE4.1.

Fortunately in DOOM 3 BFG most data that is written out to memory, is streamed to write-combined memory. However, it is worth noting that streaming out large amounts of data to cacheable memory requires double the bandwidth on x86/x64. If an entire cache line worth of data is going to be written to cacheable memory then it is wasteful to first fetch the cache line from memory into cache, only to completely overwrite the cache line afterwards. For this reason the Cell processor implements the 'dcbz' instruction. This instruction allocates a cache line associated with the given memory address. Instead of initializing the cache line with the contents from memory the cache line is set to all zeros. It would be useful to have a cache-line-clear instruction on x86/x64.

3. Compute vs. Memory

Another interesting (but not too surprising) difference between the original DOOM 3 and DOOM 3 BFG is the shift from memory usage to compute. Whereas in the original DOOM 3 an animated mesh for a game character was only ever skinned once per frame, in DOOM 3 BFG an animated mesh for a game character may be skinned many times per frame, simply because it is faster.

In the original DOOM 3 an animated mesh for a single instance of a game character was skinned only once per frame and the uniquely skinned version of the mesh was stored out to memory. The skinned copy in memory was then used to construct shadow volumes and it was also sent to the GPU to actually render the mesh (or shadow volumes). This all made sense 8 years ago when the balance between compute and memory bandwidth was somewhat different.

In DOOM 3 BFG an animated mesh for a single instance of a game character may be skinned many times per frame. The simple observation is that the cost of skinning a mesh may be mostly hidden behind the cost of reading/streaming the "un-skinned" source mesh from memory. In other words, reading/streaming the "un-skinned" source mesh from memory and skinning it in-place does not cost much more than reading/streaming an already skinned mesh from memory without performing any additional calculations.

The big win comes from never having to write the skinned mesh back to memory and never having to send/copy it to the GPU. In addition, far less memory is being read because while the same "un-skinned" source mesh may be used by many instances of a game character, a fully skinned mesh is unique to a single instance of a game character. Reading less memory reduces memory bandwidth if data can remain in CPU caches. It may also result in less cache thrashing, in particular when performing a lot of work in parallel where all parallel threads (or tasks) use the same source data (as opposed to unique source data).

To construct a shadow volume the source mesh is read, skinned in-place, followed by immediately calculating the facing triangles and silhouette edges. In DOOM 3 BFG a shadow volume is no more than a set of indices that reference a static "un-skinned" source mesh in GPU memory that is skinned on the GPU right before rasterization. As a result, an animating mesh for a single instance of a game character may be skinned many times per frame in DOOM 3 BFG. For instance, if an animating mesh interacts with 2 shadow casting lights (very common) then the mesh may be skinned 7 times per frame in DOOM 3 BFG.

2x CPU skinned to construct 2 shadow volumes
1x GPU skinned to render the depth pass
2x GPU skinned to render 2 shadow volumes
2x GPU skinned to render 2 light surfaces

In other words, DOOM 3 BFG may, for a typical animated mesh, use 7 times the number of FLOPS compared to the original DOOM 3. However, DOOM 3 BFG runs noticeably faster on today's hardware.

As an added bonus DOOM 3 BFG now maintains less state (no longer stores a skinned copy of a mesh per instance of a game character). The less state is maintained the easier it is to understand and reason about code and the less likely the code will have bugs. The stateless nature also allows a lot of code to run in parallel without contention over resources. All shadow volumes can now be constructed in parallel without the need for synchronization primitives like mutexes or critical sections.

4. Linked Data Structures

The DOOM 3 BFG source code [11] includes several questionable templates for linked lists and hierarchical data structures such as 'idLinkList' and 'idQueue'. The implementation of these two classes is particularly poor, but in general there are problems with generic (templatized) classes for linked lists and hierarchical data structures. These generic classes tend to be bloated with redundancy and they are either intrusive or require a separate memory allocation per "linked object" and use a pointer to the "linked object".

```
/*
	A public member variable of this type is added to the class of the "linked object".
	Not only is this class intrusive, the member variable also needs to be public.
	To fetch the "linked object" the class wastes space for a pointer to the "linked object".
	This class is also confusing because it is used both for a "link" and the "head" of the list.
*/
template< class type >
class idLinkList {
	idLinkList * head;
	idLinkList * next;
	idLinkList * prev;
	type * owner;
};
```

If the generic class is intrusive it either is bloated with redundancy (like idLinkList above which is a stunning 16 bytes on a 32-bit system) or it needs knowledge of the "linked object" (like idQueue below). In both cases the member variable of the generic class type needs to be public on the "linked object" which is clearly undesirable.

```
/*
	A public member variable of this type is added to the class of the "linked object".
	Not only is this class intrusive, the member variable also needs to be public.
*/
template< typename type >
class idQueueNode {
	type * next;
};

/*
	The idQueue class then needs to know the offset to the idQueueNode member
	variable stored on the "linked object".
*/
template< typename type, idQueueNode<type> type::*nodePtr >
class idQueue {
	type * first;
	type * last;
};
```

Intrusive linked lists and intrusive hierarchical data structures are generally questionable because they don't allow an object to be easily linked into multiple lists or hierarchies. One or more new member variables need to be added for every list or hierarchy the objects needs to be linked into. These member variables typically need to be public to allow third parties to iterate over the objects and they often also have to be mutable to keep all

the code const correct because adding/removing an object from a list or hierarchy does not necessarily change the true state of the object.

If the generic class is non-intrusive then an object of the generic class is separately allocated and there is redundancy because the generic class needs to store a pointer to the "linked object". Often the "linked object" also stores a pointer to the generic class to be able to easily unlink the object from the list or hierarchy (making it intrusive again). The 'idLinkList' class can be used in this manner where an 'idLinkList' object is allocated separately and optionally the "linked object" stores a pointer to this 'idLinkList' object.

Aside from poor code quality it is generally better to stay away from pointer based linked lists and hierarchies in run-time engine code because of the poor performance. It is usually much more efficient to use an 'idList' with pointers and/or use indices instead of pointers. Indices can often be smaller than pointers, they result in better cache locality and compilers usually generate more efficient code when offsetting a single pointer with an index as opposed to incrementing pointers or walking linked pointers. On a side note, the name 'idList' is really unfortunate considering this class implements a resizable heap array (although the name could be worse like 'vector').

To evaluate the potential performance benefits, compare the following code:

```
class idMyClass {
	bool valid;
	byte otherMembers[64];	// over a cache line worth of other members
	idMyClass * next;
};

idMyClass * myObjects;

for ( idMyClass * c = myObjects; c != NULL; c = c->next ) {
	if ( c->valid ) {
	}
}
```

with the code below:

```
class idMyClass {
	bool valid;
	byte otherMembers[64];	// over a cache line worth of other members
};

idList< idMyClass * > myObjects;

for ( int i = 0; i < myObjects.Num(); i++ ) {
	if (myObjects[i]->valid ) {
	}
}
```

In the first case there are two potential cache misses per iteration because the ‘valid’ flag and ‘next’ pointer are more than a cache line apart. Each cache miss costs hundreds of clock cycles that could instead be used to perform real work. In the case of an array with pointers there may still be a cache miss when reading the ‘valid’ flag but there will be at most one additional cache miss every 16 pointers (assuming 32-bit pointers and 64-

byte cache lines) and a modern CPU will predictively prefetch memory and cache pointers because they are stored contiguously in memory. Additionally, if the CPU does not predictively prefetch then the 'idList' with pointers allows manual prefetching while the linked list does not.

Being able to directly remove an object from a linked list is often used as a reason to use a linked list. Removing an object from a linked list using a non-intrusive linked list structure is obviously expensive because the linked list first has to be walked to find where the object is linked. On the other hand, removing an object from an intrusive linked list is thought to be cheap, while it really is not. Typical code to remove an object from a doubly linked list looks like the following:

```
// potential cache miss touching the ‘prev’ pointer
if ( object->prev != NULL ) {
	// potential cache miss touching the ‘next’ pointer on the ‘prev’ object
	object->prev->next = object->next;
} else {
	head = object->next;
}
// touching the ‘next’ pointer probably does not result in a cache miss
// because it is typically on the same cache line as the ‘prev’ pointer
if ( object->next != NULL ) {
	// potential cache miss touching the ‘prev’ pointer on the ‘next’ object
	object->next->prev = object->prev;
} else {
	tail = object->prev;
}
```

If we assume the ‘head’ and ‘tail’ pointers are in cache (or there may not even be a ‘tail’ pointer) then unlinking an object from a linked lists comes at the cost of three potential cache misses, each of which is hundreds of clock cycles. In other words, unlinking an object from a linked list may seem cheap when looking at the number of operations performed, it can be very expensive when it comes down to the number of cache misses and therefore total clock cycles.

Removing a pointer from an 'idList' with pointers sounds expensive because, without knowing the index at which the pointer is stored, the list has to be iterated to find the position of the pointer. Obviously each object in the list could store the index in the list at which a pointer to the object is stored but that would make the list effectively intrusive which is undesirable.

An 'idList' with pointers, however, can store 16 pointers per cache line (assuming 32-bit pointers and 64-byte cache lines). Comparing pointers while iterating the list does not require the pointers to be de-referenced and is no more than a couple of clock cycles because the branch is highly predictable (all pointers compare unequal except for the last). In other words, 16 pointers can be compared at a fraction of the cost of a single cache miss and many more at the cost of three cache misses (the typical cost of unlinking from a doubly linked list). A modern CPU will predict the memory access pattern and automatically prefetch pointers from the list because they are contiguous in memory. So in reality hundreds of pointers can be compared at a cost equivalent to three cache misses. As such, finding a pointer in a medium sized list is far less expensive than one would think. Obviously finding a pointer in a list with many thousands of pointers can still be expensive or may cause cache thrashing.

Actually removing a pointer from the 'idList' may be expensive if the list needs to stay sorted and the list is not supposed to have any NULL pointers. In this case all the pointers beyond the index at which the to be removed pointer is stored need to be moved down. Due to predictive prefetching on a modern CPU this is once again not as expensive as it may seem but it is not for free either. However, if the order of the pointers is not a concern then 'idList' implements two handy methods "RemoveFast" and "RemoveIndexFast". These methods replace the to be removed element with the last element in the list. This is obviously much less expensive.

So in general, instead of using a linked list, it is preferable to use an 'idList' with pointers because it is non-intrusive, does not require a lot of memory or many memory allocations, and iterating is more efficient. The 'idList' is also bounds checked and the code will generally be cleaner (no intrusive public variables, easy to add objects to multiple lists, more const correct code etc.). In many cases removing an object from an 'idList' will actually be faster than removing an object from a linked list. Looking at the DOOM 3 BFG code, lists are iterated more often than objects are removed, or the occasionally more expensive removal from an 'idList' is easily offset by not having the extra cost due to cache misses when iterating a linked list. If (and only if) the performance of object removal from an 'idList' does become a problem then using a linked list can still be considered (probably one rolled out by hand that does not suffer from the aforementioned problems with a generic template class). However, in many cases it may be preferable to instead use an 'idHashIndex' to be able to quickly find the index of a pointer in the 'idList' (for instance when dealing with very long lists). These are things to consider when parts of the code have been identified as problematic from a performance perspective. The last thing we want is premature optimizations at the cost of intrusive data structures that result in less clean code.

When it comes to hash tables, the 'idHashTable' template class is not very good for various reasons. This class is not intrusive but requires a memory allocation for every element that is added. For each element a node is allocated to link the element into the hash table. Such a node stores the hash key, the 'value' of the element and a pointer to the next element in the hash chain. Storing the hash key and the 'next' pointer adds at least 8 bytes of overhead for every element that is added and the overhead of general memory allocations will cause a lot of memory to be wasted. Memory access patterns will also be poor due to all the separate memory allocations. 'idHashTable' is also designed to be the only access to a set of elements because the 'value' of each element is stored on a node. Each node can store a pointer to an element (that is kept in a separate list), but that will further degrade performance due to additional pointer chasing and cache misses.

It is important to realize that a hash table is really no more than an acceleration data structure. Once again it is preferable to use an 'idList' to store elements and only if looking up elements in the 'idList' is identified as a performance problem then it is worth considering alternatives. If performance is important then it is generally preferred to use an 'idHashIndex' instead of using an 'idHashTable'. The 'idHashIndex' class may be somewhat non-intuitive at first but it has very good performance with very little overhead. In contrast an 'idHashTable' partly defeats the purpose because it is an acceleration data structure that does not have particularly good performance.

Unlike an 'idHashTable', an 'idHashIndex' does not replace an 'idList. Instead an 'idHashIndex' is added next to the 'idList' that stores the elements. If fast access based on multiple keys is desired then multiple 'idHashIndex' objects can be added next to a single 'idList' with elements. Next to storing a hash table, an 'idHashIndex' adds only 4 bytes of overhead per element. The 'idHashIndex' also uses only two memory allocations and the data is tightly packed to avoid any additional overhead and for improved memory access patterns.

5. Conclusion

Re-optimizing DOOM 3 for today's hardware not only reveals a change in the way high performance software is designed and implemented, it also exposes some interesting hardware limitations. Today's software has to implement multiple threads of execution that can run in parallel to take advantage of the multiple cores on current CPUs. However, if these threads touch a significant amount of memory then cache thrashing may occur while many CPUs are poorly equipped to avoid large scale cache pollution. Without additional instructions to manage or bypass the cache, a shared cache between all CPU cores can result in less than ideal performance.

Today's GPUs implement various hard-wired high performance components that can significantly improve the rendering performance. However, taking full advantage of these components, like the coarse Z-Cull / Hierarchical-Z, can be challenging, especially when a rendering algorithm requires the use of different depth tests. DOOM 3 uses stencil shadow volumes to define the regions in space that are in shadow of an occluder. If the view intersects or is inside a shadow volume then the fragments of the triangles that define the volume need to update the stencil buffer when they fail the depth test (Z-fail rendering). On various hardware this results in poor performance due to the way the coarse Z-Cull / Hierarchical-Z is implemented in hardware.

Analyzing the performance of the DOOM 3 and DOOM 3 BFG code also reveals various data structures that exhibit poor performance on today's hardware. These data structures tend to result in poor memory access patterns and excessive cache misses. To make matters worse, some of these data structures also results in poor code quality.

6. References

1. On Shadow Volumes
   John Carmack
   id Software, May 2000
2. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering
   Cass Everitt, Mark J. Kilgard
   NVIDIA developer site, March 2002
3. Optimizing the Rendering Pipeline of Animated Models Using the Intel Streaming SIMD Extensions
   J.M.P. van Waveren
   Intel Software Network, June 2006
4. Slerping Clock Cycles
   J.M.P. van Waveren
   Intel Software Network, June 2006
5. From Quaternion to Matrix and Back
   J.M.P. van Waveren
   Intel Software Network, June 2006
6. The Skeleton Assembly Line
   J.M.P. van Waveren
   Intel Software Network, June 2006
7. Fast Skinning
   J.M.P. van Waveren
   Intel Software Network, June 2006
8. Deriving Triangle Plane Equations
   J.M.P. van Waveren
   Intel Software Network, June 2006
9. Shadow Volume Construction
   J.M.P. van Waveren
   Intel Software Network, June 2006
10. The DOOM 3 Source Code
    id Software, 2011
    https://github.com/id-Software/DOOM-3
11. The DOOM 3 BFG Source Code
    id Software, 2012
    https://github.com/id-Software/DOOM-3-BFG