Fluid Simulation for Video Games (part 18)

By Dr. Michael J. Gourlay

Fluid Surface Extraction and Rendering

Liquids are fluids that have a free surface (that is, a surface whose shape is not defined by its container). This article—the 18th in the series—describes how to render a fluid surface.

Part 1 in this series summarized fluid dynamics; part 2 surveyed fluid simulation techniques. Part 3 and part 4 presented a vortex-particle fluid simulation with two-way fluid-body interactions that run in real time. Part 5 profiled and optimized that simulation code. Part 6 described a differential method for computing velocity from vorticity, and part 7 showed how to integrate a fluid simulation into a typical particle system. Part 8 explained how a vortex-based fluid simulation handles variable density in a fluid; part 9 described how to approximate buoyant and gravitational forces on a body immersed in a fluid with varying density. Part 10 described how density varies with temperature, how heat transfers throughout a fluid, and how heat transfers between bodies and fluid. Part 11 added *combustion,* a chemical reaction that generates heat. Part 12 explained how improper sampling caused unwanted jerky motion and described how to mitigate it. Part 13 added convex polytopes and lift-like forces. Part 14 modified those polytopes to represent rudimentary containers. Part 15 described a rudimentary smoothed particle hydrodynamics (SPH) fluid simulation used to model a fluid in a container. Part 16 explored one way to apply SPH formulae to the vortex particle method (VPM). Part 17 described how to identify the surface of a simulated fluid.

Using Isosurface Extraction to Find an Implicit Surface

Part 17 described the notion of identifying a fluid surface as the set of points where a volumetric function (such as the signed distance function) has a certain value (such as 0). Such a surface is also called an *implicit surface* because it is the solution to an implicit equation, $f(\vec{x}) = 0$. You want to find and describe such a surface in a form you can render, such as a polygon mesh.

Drawing a fluid surface is somewhat analogous to plotting contour lines on a topographic map, where each line is a curve of constant height. On a surface, height is defined everywhere, but on a topographic map, only certain heights are plotted, such as in the contour plot shown in Figure 1. Likewise, a fluid surface is where some volumetric function has a certain value. The volumetric function has a value everywhere in space, but you only want to see where that function has one particular value.

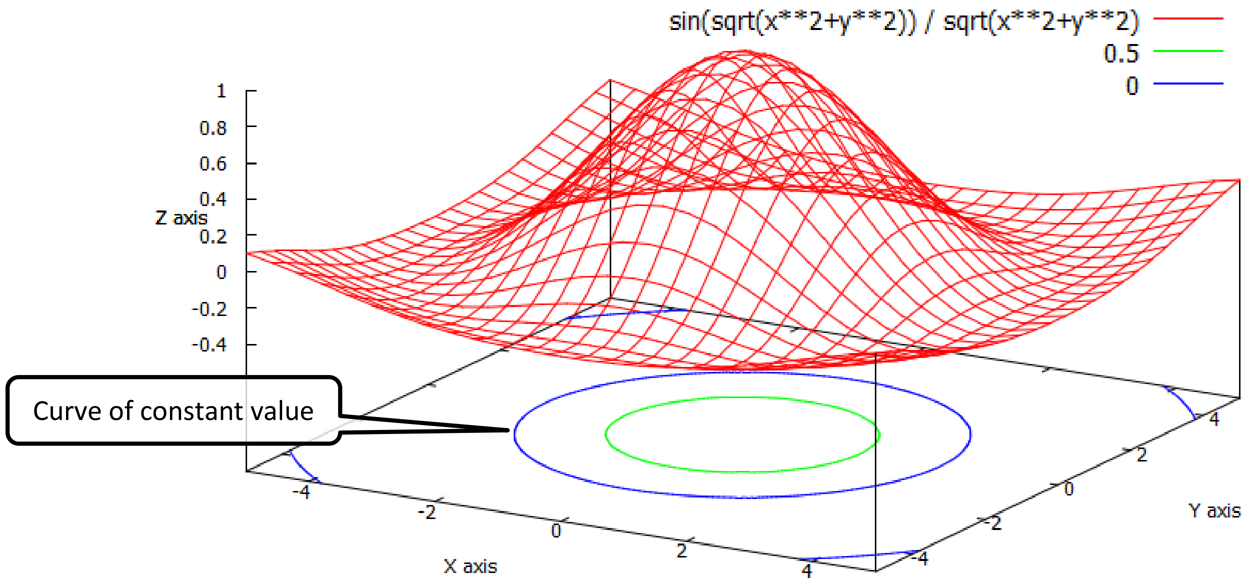

Figure 1. *Contour and surface plot. Contour lines on the bottom of the plot indicate where (in the XY plane) the function has constant values.*

This analogy breaks down somewhat because you can perceive multiple nested contours on a map; contours (such as the green and blue curves in Figure 1) do not occlude each other. But it would be more difficult to visualize multiple surfaces in a volume because each surface would occlude those inside it. Fortunately, that does not pose a real problem because you (usually) only want to render a single geometric surface at the boundary between a liquid and air.

This article assumes that the function is defined on a grid and that you just want to extract a surface of constant value, also called an *isosurface.* Refer to part 17 to see how you could define that function.

Incidentally, you could use this approach of extracting a mesh at an isosurface to visualize more than just fluid surfaces; you could use it to visualize any kind of volumetric function, for example, rendering isosurfaces of constant enstrophy to visualize a vortex tube. The vortex tube visualization in Figure 2 shows isosurfaces for vortex stretching and vorticity magnitude.

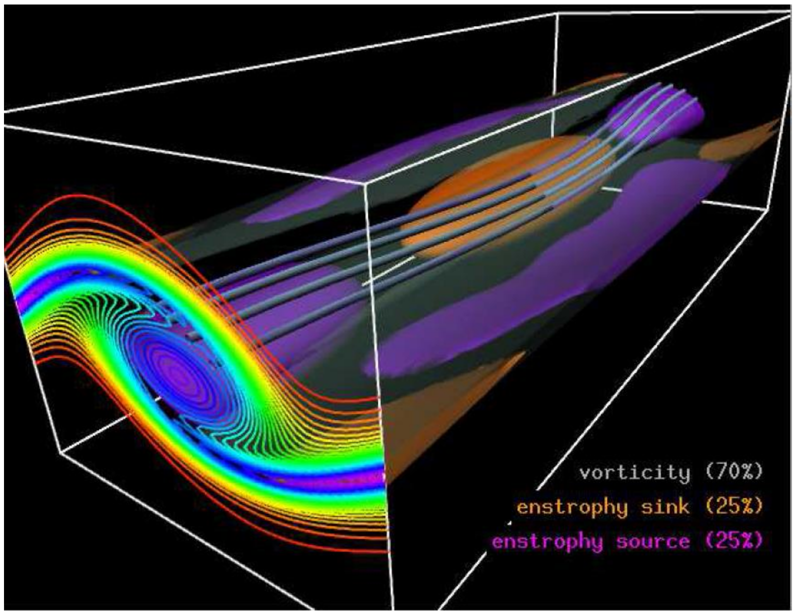

Figure 2. *Vortex tube visualization. White isosurfaces (showing the vortex tube) encompass orange and purple opaque isosurfaces, so I rendered vorticity magnitude as translucent to let you see through it. Otherwise, the white would occlude the orange and purple surfaces. Meanwhile, the contour plot on the front shows curves of constant spanwise-averaged vorticity magnitude. The contours are tantamount to cross-sections of isosurfaces. (Image from Gourlay [1999].)*

Multiple algorithms exist that can extract an isosurface from volumetric data. This article presents a classic algorithm called *Marching Cubes,* but see the “Further Reading” section at the end of this article for references to other such algorithms.

Marching Cubes

Before considering how to generate an isosurface from a volumetric function, it’s easier to understand an analogous problem in two dimensions. To generate a contour plot like the one above, start with a two-dimensional (2-D) grid of values representing a 2-D function. Each four adjacent grid points form a rectangular cell—call it a *square* for simplicity (see Figure 3). You want to render curves where the 2-D function is zero. (If you want to render curves of any other constant value, just subtract that value from the function; the rest of the algorithm applies.) The value at each point is either positive or negative. For any adjacent pair of grid points (i.e., they are joined by a square side), if the value at one point is positive and the other negative, then the function must be zero somewhere along the square edge connecting the two points. Use interpolation to find that point—the *zero-crossing*—and place a vertex there. Each square can have zero, two, or four places where the function is zero. If the square has two zeros, then connect them with a line segment. If the square has four zeros, then connect each of two pairs with two nonintersecting line segments.

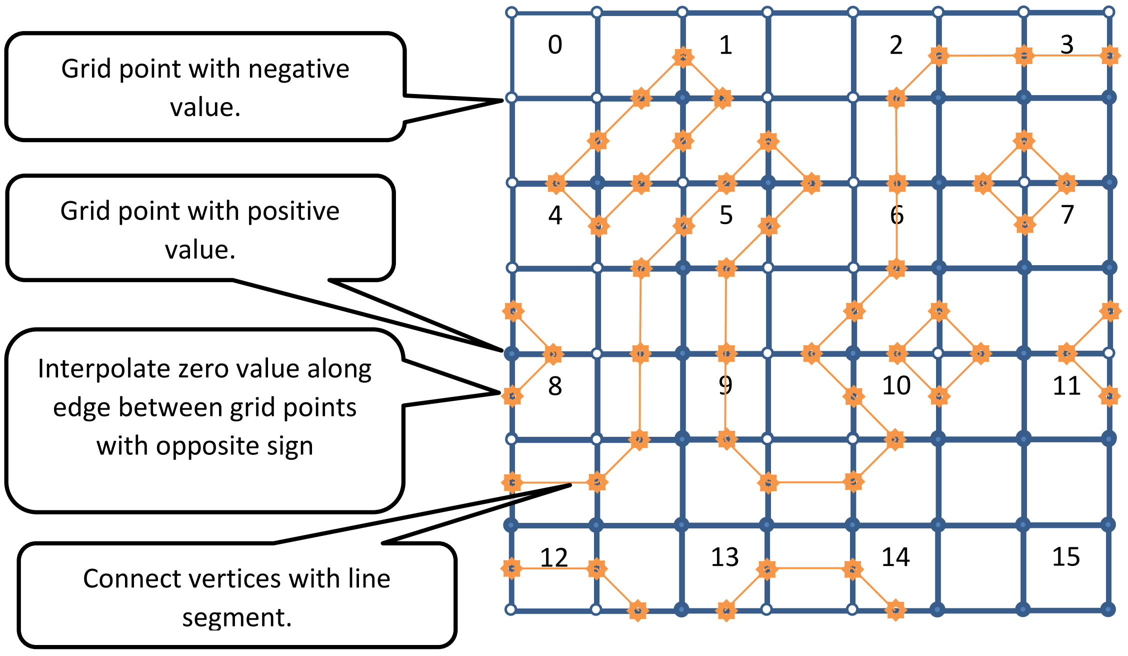
Grid point with negative value.
Grid point with positive value.
Interpolate zero value along edge between grid points with opposite sign
Connect vertices with line segment.
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Figure 3. *Marching squares cases. The grid has values at the corners. Open circles indicate negative values. Closed circles indicate positive values. Orange stars indicate zero values whose positions are interpolated along edges between corner values. Orange line segments connect zero values. Numbered cells indicate canonical cases.*

Figure 3 shows every possible combination of squares along with the resulting contour curves. (Note that in this figure, the zero vertices could be anywhere along each square edge.)

In squares with four zeros, Marching Square has some ambiguity about how to connect the dots. Look at cases 5 and 10. In Figure 3, I chose the “5” case to have diagonal lines from bottom left to top right and the “10” case to have the opposite. But that choice was arbitrary, and I could have chosen a different orientation for the lines. In fact, you can see that if I chose the “5” case to have both diagonal lines going from bottom right to top left, the graph would still “work” but would have a different (but equally plausible) topology. Ultimately, this problem has no bulletproof solution; it results from having insufficient information because of the function being discretized too coarsely for an unambiguous representation. As such, there is no unique solution to drawing the contour plot. There are, however, some conventions for how to resolve the ambiguity. One entails computing the value at the cell center by averaging the values at the four corners. Doing so effectively gives the grid another grid point (albeit a contrived one) that you can use to decide the orientation of the diagonal lines connecting the zeros.

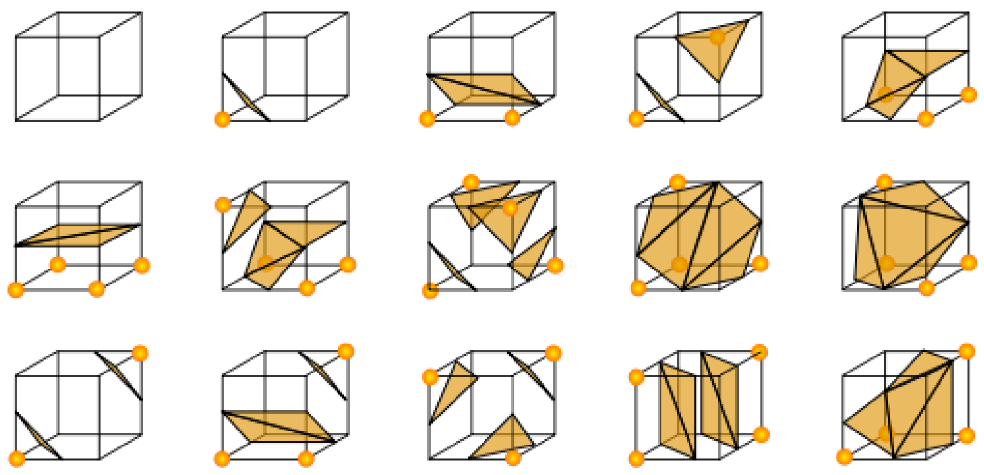

Figure 4. *Marching cubes cases. Corners with orange circles indicate grid points that have values of the opposite sign. Triangle vertices are placed at zero-crossings interpolated along cube edges between adjacent grid points that have the opposite sign. Triangle edges connect vertices. (Image from Wikipedia page on Marching Cubes.)*

The marching cubes algorithm extends the marching squares algorithm to three dimensions. The basic idea remains the same: find points along grid cell edges where the function crosses zero and place vertices there. Then, connect those vertices with triangle edges and render the triangles. Figure 4 shows the possible configurations of zero-crossings.

As with marching squares, marching cubes has ambiguous cases, and the academic literature has several (more than a hundred!) derivative algorithms that deal with that ambiguity. The "Further Reading" section includes a paper that surveys those, so feel free to take a look. Most of those algorithms share the basic structure with the classic algorithm, so the parallelization technique I describe in the next section applies just as well to them.

Generic Grid Wrapper

Marching cubes takes as input a grid of values and outputs triangles. Instead of making the marching cubes routine depend directly on any particular implementation of a grid (such as `UniformGrid`, presented in part 3), I created a lightweight generic wrapper called `GridWrapper`. `GridWrapper` makes these assumptions:

- Values in the grid lie along a set of three parallel straight lines (*x*-, *y*-, and *z*-axes).
- Each adjacent value in the grid is separated by a uniform number of bytes (the strides).
- The grid has a rectilinear topology (i.e., the grid has a uniform number of points along each axis—in other words, the grid has the same number of points along each X line, Y line, and Z line; it is not a jagged array).
- Each value in the grid is a floating-point value.

```
struct GridWrapper
{
    float * values          ;   /// values at grid points.
    size_t  number[3]       ;   /// Number of grid points along each direction.
    size_t  strides[3]      ;   /// Delta, in bytes, per index, between adjacent points in the grid.
    Vec3    minPos          ;   /// Location of first grid point, i.e. values[0]
    Vec3    directions[ 3 ] ;   /// Direction vectors corresponding to each index
} ;
```

The marching cubes routine accesses its inputs via the `GridWrapper`. The input values come from a `UniformGrid` of scalar floating-point values (but could be vectors if you wanted to plot isosurfaces of constant values of a particular vector component).

Generic Vertex Buffer Wrapper

Just as the input grid is wrapped by a lightweight structure, so is the output vertex buffer. The marching cubes routine generates triangles. Rather than copying them from an intermediate buffer, I use a wrapper to let marching cubes write triangles directly into the GPU buffer, thus avoiding a copy.

`VertexBufferWrapper` assumes the following:

- Each vertex has a position given by three adjacent floating-point values.
- Each vertex is separated from its neighbors by a constant number of bytes (the stride).
- The buffer has a fixed capacity, separate from its population.
- The buffer can optionally include normals, and their stride is the same as that for positions (such as for a vertex format that has both positions and normals). You could easily extend the wrapper to have different strides for positions and normal (e.g., if they are not interleaved).

```
struct VertexBufferWrapper
{
    float *         positions   ;   /// Array of vertex positions extracted from grid.
    float *         normals     ;   /// Array of vertex normals extracted from grid.
    TbbAtomicSizeT  count       ;   /// Number of vertices written into vertices array.
    size_t          capacity    ;   /// Maximum number of vertices that can fit into vertices array.
    size_t          stride      ;   /// Number of bytes between adjacent vertices.
} ;
```

Computing Normals

I modified the classic marching cubes implementation to generate per-face normals by computing the cross-product of triangle edge vectors. You could instead compute per-vertex normals by using the gradient of the function grid. If you already have the function gradient, this value could yield better performance, depending on which is faster: accessing memory for a second buffer or computing a cross-product.

Welding Identical Vertices

This implementation of marching cubes generates a vertex for each edge of each grid cell that has a zero-crossing. That means that for adjacent grid cells, a vertex will occur up to four times, one for each cell that shares that edge. In other words, each vertex occurs multiple times. You could avoid that if you modified the procedure to exploit knowledge of adjacent cells.

Because this implementation computes per-face normals, the resulting mesh has a faceted look, which is not desirable. To create a smoother look, you would compute per-vertex normals, which would also involve welding identical vertices.

This redundancy of identical vertices has a benefit: Because each cell is completely ignorant of its neighbors, the algorithm is (nearly) embarrassingly parallelizable. The output for one cell has (almost) no dependency on the output from any other cell, even adjacent ones. If the procedure welded vertices, adjacent cells would depend on each other.

Welding would also complicate computing normals in another way: each unique vertex would have contributions for its normal from each triangle that shares that vertex. One way to accomplish that would be to sum the normal contribution from each triangle. (You could also weight each term in that sum by the triangle size or its reciprocal.) Then, at render time, have the GPU normalize each normal.

For the sake of brevity, this article does not delve into welding or per-vertex normals, but in production code it would be important to deal with those issues both to reduce the number of vertices and to create smoother-looking surfaces.

Parallelizing Marching Cubes

The marching cubes procedure is *almost* embarrassingly parallel, except that the output of each cell goes into a single vertex buffer. If the algorithm knew ahead of time where each output triangle would go in the vertex buffer, parallelizing it would be easy: the routine could just write each vertex into its destination slot. But, alas, the destination is not cheap or easy to predict. Each cell could generate a different number of vertices (0, 3, 6, 9, or 12) and triangles (0, 1, 2, 3, 4, or 5). Determining that entails nearly all the work involved in performing extraction.

The vertex buffer is allocated ahead of time (for example, using the rendering system application programming interface [API]). You want to partition that buffer so that each thread can access its own portion, avoiding race conditions with other threads. This avoids each thread needing to synchronize access to the vertex buffer. The trick, then, is deciding how to partition the vertex buffer. Again, each thread will not know ahead of time how many vertices it will generate.

To cope with this variability, partition the vertex buffer into blocks and have each thread “allocate” one block at a time. A *block* is a contiguous segment of the vertex buffer, with capacity for multiple vertices. You use the original vertex buffer as a kind of memory resource (sort of like a heap), then allocate one block at a time. Each thread can have its own block. Only these block allocations require synchronized access to the original vertex buffer. I will describe this mini-allocation scheme in more detail shortly, but first I explain the synchronization mechanism because it has some subtlety and warrants attention.

Synchronizing with an Atomic Integer

One way to synchronize memory access entails using an atomic variable—that is, a variable whose value is accessed atomically. In this case, a single atomic integer provides all the synchronization you need for this block allocator. The “count” member of the vertex buffer object keeps track of the number of populated vertices. You just need to make that count member atomic.

Intel® Threading Building Blocks (Intel® TBB) provides atomic versions of integer types, but to let the code compile even without Intel TBB, I wrote a simple proxy class (`TbbAtomicSizeT`) that has the same methods as the `atomic`:

```
#if USE_TBB
        typedef tbb::atomic< size_t > TbbAtomicSizeT ;
#else
        class TbbAtomicSizeT
        {
        public:
            TbbAtomicSizeT() {}

            size_t fetch_and_add( size_t increment )
            {
                size_t originalValue = _value ;
                _value += increment ;
                return originalValue ;
            }

            operator size_t () const { return _value ; }

            size_t & operator=( size_t newValue ) { _value = newValue ; return _value ; }

        private:
            size_t _value ;
        } ;
#endif
```

The magic routine of `atomic` is `fetch_and_add`. Each time a thread wants to allocate a new block, it (atomically) increments that integer (the vertex count) by the number of vertices in a block, and `fetch_and_add` returns the value prior to the increment. This might seem counterintuitive: Shouldn't it return the incremented value? No, the calling thread wants to know what the value was just before the increment because that is also the value of the index into the buffer where that block starts. But couldn't the thread just read the counter value just prior to incrementing it? Shouldn't that yield the same value? Not always; if two threads accessed the counter at the same moment, then both threads could read the counter and get the same value. Then, each thread would mistakenly think it should use the same region of the vertex buffer. Using `fetch_and_add`, each thread is guaranteed a unique value.

You could use different synchronization mechanisms, such as a mutex, but using an atomic probably has lower overhead and faster performance. Mutexes involve a *system call* that can involve performing a *context switch:* processor control transfers to the kernel (which involves changing privileges), which runs the routine to obtain the lock (possibly involving conditional branches) and returns control to the calling thread (again, with the attendant privilege context switch). That overhead can cost perhaps 50 times as much as an atomic access. In contrast, an atomic operation is implemented in hardware as a single instruction and avoids a system call.

Still, using atomic instructions is expensive. To guarantee atomic access, the hardware has to lock the bus to prevent any other processor core from accessing memory during the atomic access. Exacerbating this, if the L1 cache for another core contains the same memory, the hardware needs to invalidate that cache. This requirement ensures *cache coherence,* but it's expensive. One way to avoid such issues involves making sure each core's L1 cache does not overlap that of any other core. But because the vertex count is shared, it's impossible in this case. The upshot is that you want to reduce the amount of times you increment the atomic.

Synchronized Linear Pool Allocator

In principle, you could have each thread allocate a single vertex at a time, but doing so would involve accessing the atomic count more often, which is expensive (as described above). In contrast, you could have each of the *N* threads preallocate 1/*N* of the vertex buffer, thus having at most one mini-allocation per thread. But as described earlier, threads don't know now many vertices they will generate, so that approach is likely to yield allocating the wrong number of vertices. Some threads will have too many vertices available, while other threads will not get enough. So, use a compromise: each thread allocates vertices in blocks, where the block size balances the conflicting goals of having fewer mini-allocations while having fine enough granularity so that no thread takes too many vertices.

VertexLinearPoolAllocator

Treat the original vertex buffer as a "mother" from which each thread will allocate blocks of equal size. Such a memory resources is called a *pool.*

These block allocations are never freed, and there is no need to keep track of a free list. In addition, the allocations always proceed from the start to the end. The allocated blocks are contiguous. Such an allocation scheme is called *linear.*

So, I created a linear pool allocator to wrap the mother vertex buffer. The `VertexLinearPoolAllocator` class implements this buffer:

```
class VertexLinearPoolAllocator
{
    static const size_t INVALID_INDEX               = ~ size_t( 0 ) ;   /// Special value for mIndexOfCurrentBlock.
    static const size_t NUM_VERTICES_PER_TRIANGLE   = 3 ;   /// Determined by basic geometry :)
    static const size_t NUM_TRIANGLES_PER_BLOCK     = 64 ;  /// power-of-2 to arrange for disjoint cache lines for each block.

public:

    static const size_t NUM_VERTICES_PER_BLOCK      = NUM_TRIANGLES_PER_BLOCK * NUM_VERTICES_PER_TRIANGLE ;
```

```
private:
    TbbAtomicSizeT * mVertexCounter         ; // Currently "allocated" number of vertices within the pool.
    size_t mIndexOfCurrentBlock             ; // Index, with pool, of current block used to allocate vertices.
    size_t mNumVertsAllocedInCurrentBlock   ; // Number of vertices allocated within the current block.
    size_t mTotalCapacityInVertices         ; // Total capacity of VB; used to check for overflow.
} ;
```

Remember that the hardware needs to keep memory access cache coherent. Cache is divided into *lines,* typically 64 bytes each. The hardware will invalidate an entire cache line every time any core modifies any value in that line. If any core's L1 cache overlapped with that of another core, then even if neither core accesses the same physical memory addresses, if those addresses share a cache line, the line is invalidated. This behavior can generate a lot of useless cache invalidations. Therefore, align each block so that its cache lines never overlap those of other blocks. Each block start should be cache-aligned and have a size that is a multiple of the cache line size. In principle, you would need to know the cache line size to do so. For this implementation, I assumed a cache line size of 64 bytes, which is typical for Intel® Core™ i7 processors.

Each thread has its own `VertexLinearPoolAllocator` object, which keeps track of the index of the current block for that thread and the number of vertices allocated within that block.

AllocateTriangle

The `AllocateTriangle` method allocates a triangle (i.e., three vertices) at a time. It first checks to determine whether the current block has room for another three vertices. If not, it allocates another block from the pool (i.e., from the mother vertex buffer). Remember, that involves an atomic increment to the total vertex count, which is a slow and expensive operation. But most of the time, the current block will still have room, in which case the block counter is merely incremented. Because that counter is per thread, it does not require a synchronized atomic operation and so is relatively fast—just a plain old integer increment.

Note: This linear allocator could allocate a single vertex at a time, but in this incarnation, marching cubes always allocates vertices in triples because each triangle has three vertices. Remember, if you wanted to exploit the fact that each vertex will occur multiple times—once for each triangle that connects to that vertex—you would want to modify this mini-allocator to allow allocating a single vertex at a time. But that's a topic for another potential article or an exercise for the reader.

```
size_t VertexLinearPoolAllocator::AllocateTriangle()
{
    if(      ( INVALID_INDEX == mIndexOfCurrentBlock )
        ||  ( mNumVertsAllocedInCurrentBlock + NUM_VERTICES_PER_TRIANGLE > NUM_VERTICES_PER_BLOCK ) )
    {   // There is no current block yet (i.e. no allocations occurred yet) or no room remains in this block.
        // Try to allocate a block.
        // NOTE: fetch_and_add returns the *original* value, and increments the variable.
        mIndexOfCurrentBlock = mVertexCounter->fetch_and_add( NUM_VERTICES_PER_BLOCK ) ;
        mNumVertsAllocedInCurrentBlock = 0 ;
        if( mIndexOfCurrentBlock + NUM_VERTICES_PER_BLOCK >= mTotalCapacityInVertices )
        {   // Exceeded pool capacity.
            FAIL() ;
            mIndexOfCurrentBlock = INVALID_INDEX ;  // Try to make failure obvious.
        }
    }
    // Room remains within this block.
    size_t vertexIndex = mIndexOfCurrentBlock + mNumVertsAllocedInCurrentBlock ;   // Calculate offset of next vert in block.
    mNumVertsAllocedInCurrentBlock += NUM_VERTICES_PER_TRIANGLE ; // Allocate vertices for that triangle.
    size_t triangleIndex = vertexIndex / NUM_VERTICES_PER_TRIANGLE ;
    ASSERT( triangleIndex * NUM_VERTICES_PER_TRIANGLE == vertexIndex ) ;
    return triangleIndex ;
}
```

FillBlockRemainerWithDegenerateTriangles

One minor complication of allocating vertices in blocks is that a thread could terminate without using up all the vertices in the last block it allocated. Eventually, the GPU will read the original vertex buffer as a single contiguous array of vertices, including those leftover, unused vertices in the block.

You could compact all the vertices so that the vertex buffer has no such “holes.” You could, for example, copy vertices from the end of the buffer into those holes. Maybe that would yield better overall performance, but another solution is available that is even simpler and probably has adequate (possibly even better) performance: fill the unused vertices in the block with degenerate triangles.

A *degenerate triangle* is one that has zero area. Any triangle that has at least two identical vertices has zero area. Certainly, a triangle whose vertices are all at the same position is also degenerate. That means you could simply assign those leftover vertices, all with the same value, and the resulting triangles would be degenerate. It doesn’t matter what the values are as long as they are all the same.

I chose to assign the value `FLT_MAX`, but you could also use zero or any other value. I figured `FLT_MAX` would be easier to spot in a memory view and unlikely to occur randomly as a default unassigned value, so if the code had bugs, such a memory pattern would be easier to spot.

```
void VertexLinearPoolAllocator::FillBlockRemainderWithDegenerateTriangles( float * positionsStart , size_t vertexStrideInBytes )
{
    unsigned char * positionsAsBytes = reinterpret_cast< unsigned char * >( positionsStart ) ;
    for( size_t idxVertWithinBlock = mNumVertsAllocedInCurrentBlock ; idxVertWithinBlock < NUM_VERTICES_PER_BLOCK ;
         ++ idxVertWithinBlock )
    {   // For each unallocated vertex within current block...
        const size_t    idxVertWithinMotherBuffer   = idxVertWithinBlock + mIndexOfCurrentBlock ;
        const size_t    offsetInBytes               = idxVertWithinMotherBuffer * vertexStrideInBytes ;
        float *         vertPosAsFloats             = reinterpret_cast< float * >( & positionsAsBytes[ offsetInBytes ] ) ;
        // The value here does not matter, as long as all 3 vertices in the triangle have the same coordinate,
        // or more precisely, as long as the triangle has zero area.  I chose to use an extreme value here to make
        // it more obvious, when using a memory watch, which vertices are for padding.
        vertPosAsFloats[ 0 ] =
        vertPosAsFloats[ 1 ] =
        vertPosAsFloats[ 2 ] = FLT_MAX ;
    }
}
```

Using Intel® Threading Building Blocks to Parallelize Marching Cubes

Parallelizing marching cubes entails creating a `VertexLinearPoolAllocator` per thread, having the triangle generation routine use that allocator, and partitioning the input grid uniformly across multiple threads. Intel TBB `parallel_for` handles the last part with ease.

As in previous articles in this series, you arrange for the processing routine to take begin and end indices and write a functor to wrap calling that routine. Intel TBB takes care of spawning threads and calling the functor.

The `ExtractIsoLevel_Slice` routine is the extraction loop. Here, I show only the structure of the loop; see the accompanying code to examine the loop body.

Note that this routine creates the `VertexLinearPoolAllocator` object. Because Intel TBB will have already spawned a thread to run this routine, each thread will have its own allocator object.

Also note that this routine fills the remainder of the block with degenerate triangles, as described above. When all threads finish, the vertex buffer will therefore be in a state ready for the GPU to render it directly, even though the buffer might have a few “bubbles,” with degenerate triangles interspersed among the legitimate triangles.

```
ResultCodeE ExtractIsoLevel_Slice( VertexBufferWrapper * vertexBufferWrapper , float isoLevel , const Grid * valGrid
    , size_t zStart , size_t zEnd )
{
    PERF_BLOCK( ExtractIsoLevel ) ;

    ASSERT( vertexBufferWrapper->normals != vertexBufferWrapper->positions ) ; // VB has same address for positions and normals

    VertexLinearPoolAllocator vertexAllocator( & vertexBufferWrapper->count , vertexBufferWrapper->capacity ) ;

    ...Polygonize cells and extract normals...

    // Fill remainder of block with degenerate triangles.
    vertexAllocator.FillBlockRemainderWithDegenerateTriangles( vertexBufferWrapper->positions , vertexBufferWrapper->stride ) ;

    return RESULT_OKAY ;
}
```

The `ExtractIsoLevel_TBB` class is the functor that wraps the `ExtractIsoLevel_Slice` routine. This has the structure that Intel TBB requires for use with `parallel_for` as described and used many times in previous articles in this series.

```
class ExtractIsoLevel_TBB
{
    VertexBufferWrapper * mVertexBufferWrapper ;
    float mIsoLevel ;
    const Grid * mValGrid ;
public:
    void operator() ( const tbb::blocked_range<size_t> & r ) const
    {   // Extract isosurface for subset of grid.
        SetFloatingPointControlWord( mMasterThreadFloatingPointControlWord ) ;
        SetMmxControlStatusRegister( mMasterThreadMmxControlStatusRegister ) ;
        ExtractIsoLevel_Slice( mVertexBufferWrapper , mIsoLevel , mValGrid , r.begin() , r.end() ) ;
    }

    ExtractIsoLevel_TBB( VertexBufferWrapper * vertexBufferWrapper , float isoLevel , const Grid * valGrid )
        : mVertexBufferWrapper( vertexBufferWrapper )
        , mIsoLevel( isoLevel )
        , mValGrid( valGrid )
    {
        mMasterThreadFloatingPointControlWord = GetFloatingPointControlWord() ;
        mMasterThreadMmxControlStatusRegister = GetMmxControlStatusRegister() ;
    }
private:
    WORD        mMasterThreadFloatingPointControlWord   ;
    unsigned    mMasterThreadMmxControlStatusRegister   ;
} ;
```

The `ExtractIsoLevel` routine is the outermost caller. It sets up the `parallel_for` call that also involves creating the functor.

```
ResultCodeE ExtractIsoLevel( VertexBufferWrapper * vertexBufferWrapper , float isoLevel , const Grid * valGrid )
{
    const size_t numZMinus1 = valGrid->number[2] - 1 ;
    ResultCodeE  resultCode = RESULT_OKAY ;

#if USE_TBB
    // Estimate grain size based on size of problem and number of processors.
    const size_t grainSize =  Max2( size_t( 1 ) , numZMinus1 / 2 /*/ GetNumberOfProcessors() */) ;
    // Extract isosurface using TBB
    parallel_for( tbb::blocked_range<size_t>( 0 , numZMinus1 , grainSize )
        , ExtractIsoLevel_TBB( vertexBufferWrapper , isoLevel , valGrid ) ) ;
    if( vertexBufferWrapper->count + VertexLinearPoolAllocator::NUM_VERTICES_PER_BLOCK > vertexBufferWrapper->capacity )
    {   // Vertex buffer probably had insufficient capacity.
        resultCode = RESULT_INSUFFICIENT_CAPACITY ;
    }
#else
```

```
    resultCode = ExtractIsoLevel_Slice( vertexBufferWrapper , isoLevel , valGrid , /* zStart */ 0 , numZMinus1 ) ;
#endif

    return resultCode ;
}
```

Summary

This article showed how to extract surfaces from a function represented by a grid as a polygon mesh that you can then render. Combined with previous articles in this series, that lets you visualize surfaces of simulated fluids. This article also showed how to parallelize that isosurface extraction routine. Although the routine's outputs for a given input cell do not appear to depend on those of other cells, there is a subtle dependency on *where* to write the output. To solve this question, I wrapped the output vertex buffer with a lightweight, thread-safe, linear pool allocator, thereby avoiding race conditions between threads, keeping the vertex buffer usage relatively balanced across threads and the synchronization overhead manageable. You can tune the tradeoff between balancing thread usage of the buffer and synchronization overhead by changing the block size.

Future Considerations

As mentioned above, this implementation makes no attempt to weld identical vertices, but it would be worthwhile to do so for at least two reasons. First, it would reduce the total number of vertices (thus improving rendering performance). Second, it would facilitate computing per-vertex normals (thus allowing for smoother-looking mesh surfaces). Accomplishing this in a thread-safe way that is amenable to parallelization would be an interesting and worthwhile challenge.

Because marching cubes generates triangles for each grid cell, it tends to generate a lot of triangles where a few would suffice. For example, even if the resulting surface were perfectly flat and could be represented by a small number of triangles, marching cubes will generate many triangles. You could reduce the number of triangles by simplifying the mesh as a postprocess or by incorporating a mesh extraction algorithm that automatically choses a different discretization resolution as appropriate for the level of curvature in the input data.

Further Reading

- Learn more about the classic marching cubes algorithm in W.E. Lorensen, and H.E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics* 21, no. 4 (1987): 163–169.
- For a survey of 195 papers related to the marching cubes algorithm, including several that resolve the ambiguity described in this article, see T.S. Newman, and H. Yi, "A Survey of the Marching Cubes Algorithm." *Computers & Graphics* 30(2006): 854–879.
- For an isosurface extraction algorithm that can directly generate grids for multiple resolutions, see T. Poston, Tim, T-T. Wong, and P-A. Heng, "Multiresolution Isosurface Extraction with Adaptive Skeleton Climbing." *Computer Graphics Forum* 17, no. 3(1998): 137–147.
- For more information about the dividing cubes algorithm, see H.E. Cline, W.E. Lorensen, S. Ludke, C.R. Crawford, and B.C Teeter, "Two Algorithms for Three-Dimensional Reconstruction of Tomograms," *Medical Physics* 15, no. 3 (1988):320–327.
- For a description of one way to parallelize marching cubes using a GPGPU approach, see C. Dyken, G. Ziegler, C. Theobalt, and H-P. Seidel, "High-Speed Marching Cubes Using HistoPyramids," *Computer Graphics Forum* 27, no. 8(2008): 2028–2039.
- For visualizations of vortices, including contour plots, isosurfaces, and other volumetric rendering, see M. Gourlay, "Stability and Dynamics of Stretched Fluid Shear Layers," Ph.D. thesis, 1999.

About the Author

Dr. Michael J. Gourlay works at Microsoft as a principal development lead on a new computing platform. He previously worked at Electronic Arts (EA Sports) as the software architect for the Football Sports Business Unit, as a senior lead engineer on *Madden NFL,* on character physics and the procedural animation system used by EA on *Mixed Martial Arts*, and as a lead programmer on *NASCAR.* He wrote the visual effects system used in EA games worldwide and patented algorithms for interactive, high-bandwidth online applications. He also developed curricula for and taught at the University of Central Florida, Florida Interactive Entertainment Academy, an interdisciplinary graduate program that teaches programmers, producers, and artists how to make video games and training simulations. Prior to joining EA, he performed scientific research using computational fluid dynamics and the world's largest massively parallel supercomputers. His previous research also includes nonlinear dynamics in quantum mechanical systems and atomic, molecular, and optical physics. Michael received his degrees in physics and philosophy from Georgia Tech and the University of Colorado at Boulder.