Fast Optical Monte Carlo Simulation With Surface-Based Geometries Using *Chroma*

Stanley Seibert Anthony LaTorre

July 9, 2011

Abstract

We propose an alternative approach to Monte Carlo simulation of optical photons based on a detector geometry defined by triangulated surfaces rather than constructive solid geometry. This technique allows for highly optimized track propagation techniques using bounding volume hierarchies, and is also amenable to parallelization with a graphics processing unit (GPU). We describe a working implementation of this idea, called *Chroma*. Chroma provides a very fast realtime ray-tracer as well as a full optical Monte Carlo of a detector that can be nearly 50 times faster than GEANT4, propagating more than 1 million photons per second. We show some example uses of Chroma, including visualization of PMT properties in 3D and realtime Monte Carlo PDF generation for maximum likelihood based reconstruction.

1 Introduction

The exponential growth of computing power per dollar has had a profound effect on analysis techniques in particle physics. With fast workstations and large clusters at our disposal, Monte Carlo simulation has become an extraordinarily useful tool at all stages of experiment design and data analysis. Standardized libraries like GEANT4 provide a common framework for creating particle-based simulations that has been been adopted by many recent high energy and nuclear physics experiments. With more than a decade of development, and a broad user base, GEANT4 has many features and many aspects of it have been well-verified using real detector data.

Standardization can breed some measure of complacency, however. Early design choices achieve a kind of *rigor mortis* over time, hardening into unconscious assumptions that we forget we made in the first place. Perhaps one of the more fundamental choices in GEANT4 is in the description of a detector geometry:

> *A detector is a tree of nested solids, each composed of some material and mathematically implemented by a particular C++ class.*

As will be described in the next section, this is a powerful paradigm, but one we believe has serious performance consequences. In this paper, we explore an alternative description:

> *A detector is a list of oriented triangles, each representing the boundary between an "inside" and "outside" material.*

The distinction between (and implications of) a solid-based versus a surface-based geometry system is discussed in more detail in the next section.

We want to be clear: we come not to bury GEANT4, but to complement it. Surface-based modeling has its own tradeoffs, but can be very effective for some problems. In particular, we will show that for the narrow problem of fast propagation of optical photons in a large detector, the surface-based approach can be much faster and easily parallelized on increasingly common graphics processing units (GPUs). We will describe our software implementation of this idea, called *Chroma*.

2 3D Representations

Since the invention of computer graphics in the 1960s, computer scientists have studied the representation of 3D objects in software. Modern software for manipulating 3D models generally follows one of two paradigms. The first paradigm is that of *parametric solid modeling*, and the second is that of *surface-based modeling*. The former is traditionally employed by engineering CAD software and was thus a natural choice for GEANT4 as well. The latter is more commonly used for artistic applications, such as 3D animation and games.

2.1 Solid-based Modeling

GEANT4 fundamentally uses a solid-based approach to geometry specification, along with a tree structure that is used both in a descriptive role and to accelerate track propagation. These two roles are fully separated in the surface-based modeling used by Chroma, so it is important to first dissect the GEANT4 approach.

2.1.1 Detector Definition

In solid-based modeling, a scene is composed of primitive solid volumes described by some number of parameters. For example, a sphere is a primitive solid described by a single radius parameter. The primitive solid can be displaced and rotated, and also combined with other primitive solids using boolean operations like union, intersection, and difference. More sophisticated solid primitives can be defined, such as solids of revolution, extruded 2D shapes, even arbitrary polyhedra. All solids have a closed boundary, and thus a finite volume.

A sphere is defined by more than just a single number, however. The radius must be combined with functions that can compute useful properties about the solid given that piece of information. For example, all solid types in GEANT4 must implement functions that can answer questions like:

- Is a point $\vec{x}$ contained within the solid?
- Starting at point $\vec{x}$, does a ray in the direction $\hat{u}$ intersect the surface of the solid?
- What is the normal vector on the surface of the solid at point $\vec{x}$?
- What is the smallest axis-aligned box that encloses the solid?

These functions (and others) must cover all of the relevant calculations one might want to do with a solid, either directly or through composition with some external calculation.

GEANT4 specializes the solid-based modeling paradigm by requiring the solids in the scene to be structured in a tree. At the top of the tree is the world volume, which can contain zero or more non-overlapping daughter volumes. These daughter volumes in turn can contain their own daughters, and so on. Each daughter volume implicitly subtracts its space from its mother volume, thus the entire tree could be viewed as partitioning the space into non-intersecting regions. Each region is comprised of some material, like water or glass, with well-defined properties.

The tree is a convenient way to specify a geometry where objects naturally contain daughters that displace the parent material, in the way that water or air can be displaced by the photomultiplier tubes in a detector[1]. Perforated solids, such as a tank with holes for piping, can be more awkward to work with, as the interior of the solid is no longer fully contained by the outside. Such situations usually require a partial flattening of the geometry tree, with objects inside the perforated container being elevated to the level of siblings with the container itself.

2.1.2 Track Propagation

In GEANT4, the geometry tree performs a second duty. Track propagation uses the tree structure to limit the number of function calls required to compute the next boundary intersection. At any given step, the track starts inside a volume somewhere

[1]Or, less intuitively, the way that a vacuum can be said to "displace" glass inside a PMT.

in the geometry tree. A ray in the direction of particle travel must be checked for intersection with the containing volume, plus all of the immediate daughter volumes. As a result, the number of sibling volumes at a given node is a strong indicator of the tracking performance. This motivates the GEANT4 user to create deep and narrow geometry trees, which can sometimes be at odds with the structural requirements of volumes fully enclosing their children. For example, the previously mentioned difficulty with perforated containers usually results in a "slow" geometry where volumes inside the container have to be promoted up to a level where they are siblings to the container itself. The resulting geometry tree has a wide layer with many siblings, and therefore more solids to test for intersection when tracks are propagating both inside and outside the container wall.

Recognizing that the geometry tree may not be the most optimal data structure for efficient track propagation, GEANT4 further augments the geometry tree by dividing nodes into *voxels*, which are simply boxes with each face perpendicular to one of the Cartesian axes. Voxels are then tagged with the volumes that intersect them, and a propagating track first checks for voxel intersection, then checks for intersection with the real volumes associated with that voxel. Since box intersection is fast to compute, the voxel optimization substitutes a series of slow intersection calls (with arbitrary solid primitives) for a series of fast intersection calls and then a much smaller list of slow calls. Voxels, however, cannot accelerate the performance of an individual solid, due to the solid representation being hidden from the voxel generator[2].

While effective, the voxel technique in GEANT4 is somewhat limited. Voxels augment the geometry tree specified by the user; they do not replace it. A voxel cannot span levels in the geometry tree, so there is still quite a bit of onus on the GEANT4 user to structure their geometry in a way that allows for fast particle tracking. The creation of voxels in a volume is done by slicing just along one axis, then within each slice by optionally slicing along another axis, and then finally along the third axis if necessary. This iterative approach saves memory, but can be very slow to execute with volumes containing large numbers of daughters.

2.2 Surface-based Modeling

For the purposes of photon tracking, we propose a complementary view of the detector geometry through a representation that focuses on 2D boundaries between volumes rather than the 3D volumes themselves. Fundamentally, these are completely equivalent perspectives from which to describe a 3D scene, although they naturally lead to different implementations and design choices. In this work, we tend to follow the lead of the well-established 3D animation and rendering community.

2.2.1 Detector Definition

Unlike solid modeling, where a great variety of solid primitives (or Boolean combinations thereof) are required to describe most detectors, surfaces are typically represented in a software framework using exactly one simple primitive repeated many times. The

[2]And everything else, for that matter, thanks to the double-edged sword of encapsulation.

most common surface primitive is the triangle, although non-planar primitives are sometimes used, such as Bézier triangles or NURBS[3]. The patches are stitched together to produce a continuous surface. Without loss of generality, we will assume a surface model composed purely of triangles, often called a *triangle mesh*.

The use of a single primitive for all surfaces is the first point of departure between the solid and surface-based approach. GEANT4's infinitely extensible primitives can be used to represent almost any computable solid exactly by implementing the required list of methods for a G4VSolid subclass. However, unless the surface of the solid is piecewise flat everywhere, a finite triangle mesh cannot represent it exactly. Instead, the level of approximation is determined by the number of triangles used to represent the curved surface. Accuracy is traded for memory, and to a lesser extent, speed. An intrinsically curved primitive, like a Bézier triangle, can better approximate a wider range of surfaces with fewer patches, at the expense of slightly more complex data and code required to implement the primitive. Nevertheless, Bézier triangles are also fundamentally an approximation to surfaces in general, just like standard planar triangles.

Accepting this kind of approximation in the geometry description may be unsatisfying, but is not unique to surface modeling. Models of photomultiplier tubes often treat the curved front face as a section of a sphere or ellipsoid for speed reasons, when in fact it is generally a complex curve rotated around a central axis. It is important to be aware of the accuracy tradeoff in surface-based modeling, as the loss of precision tends to be larger than the solid modeling case. When the precision is not acceptable, the solution is to use a finer mesh.

When describing a detector with triangle meshes, there is no tree-like nested volume hierarchy. Each mesh exists in the global coordinate system, and in fact, the detector could be thought of as being composed of exactly one giant list of triangles. To represent the material composition of the detector, both bulk and surface properties, the triangles need to be augmented with some additional properties. First, we need the triangles to be *oriented*. This is typically done by establishing an order for the triangle vertices following a "right-hand rule" to determine the direction of the surface normal, as shown in Figure 1. The normal defines the side of the triangle considered "outside." Once the triangle has an inside and an outside, we can define an *inside material*, an *outside material*, and a *surface material*. The inside and outside materials identify the bulk properties of the two media the boundary separates. These are the same properties the equivalent solids would have, such as index of refraction and absorption lengths. The surface material describes the optical properties of the surface, such as diffuse and specular reflectivity.

One advantage to representing the surface as small triangular patches is the ability to easily specify different surface properties for different regions of the same object. The surface material is a property of the triangle, and triangles with different surface properties can be connected in the same closed mesh. In contrast, applying a reflector to one end of a transparent light guide in GEANT4 usually requires construction of a dummy volume touching the light guide to define a surface patch.

[3]Non-uniform rational basis spline.

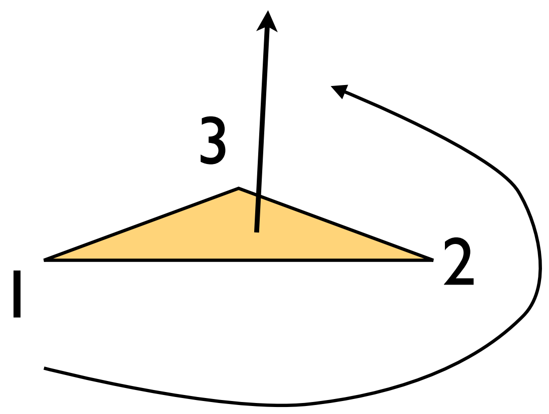

Figure 1: Right-hand rule for determining the direction of the outward normal from the order of the vertices in an oriented triangle.

2.2.2 Track Propagation

Much like the tree-based GEANT4 representation of a detector has a consistency condition to ensure every point in space has a well-defined material, so does a surface-based representation. In GEANT4, volumes are required to be wholly contained within their mother volume, and not overlap with their siblings. In a surface-based model, any path between exactly two triangles must connect sides with the same bulk material. Figure 2 shows an example of this condition. The easiest way to achieve this consistency condition is to create a geometry that is a union of non-intersecting closed meshes, although that is not a requirement. Open meshes have a path from the inside surface to the outside surface, so if they are present in a geometry, the inside and outside material must be the same. Intersecting meshes can be acceptable in practice if particles cannot enter a region where there are paths that violates the consistency condition[4].

Efficient track propagation in a surface-based model requires a tree structure, just as it does for a solid-based GEANT4 model. A well-designed navigation tree can reduce the number of intersection tests required from $O(n)$ to $O(\log n)$. However, unlike the solid-based case, the flattened definition of a surface-based detector provides no inherent tree structure to exploit. This actually proves to be an advantage, as the software is now free to construct a navigation tree independent of how the user described the geometry initially. Additionally, the choice of a single surface primitive allows for very aggressive optimization of the construction of the tree since there is only one case to implement. We will discuss one possible navigation data structure, called a *Bounding Volume Hierarchy*, in Section 5.1, although others are possible.

[4]Intersecting meshes can be made consistent for all paths by adding appropriate priority values to triangles and modifying the path consistency condition to take this into account. This is beyond the scope of this current document, but may be explored in the future.

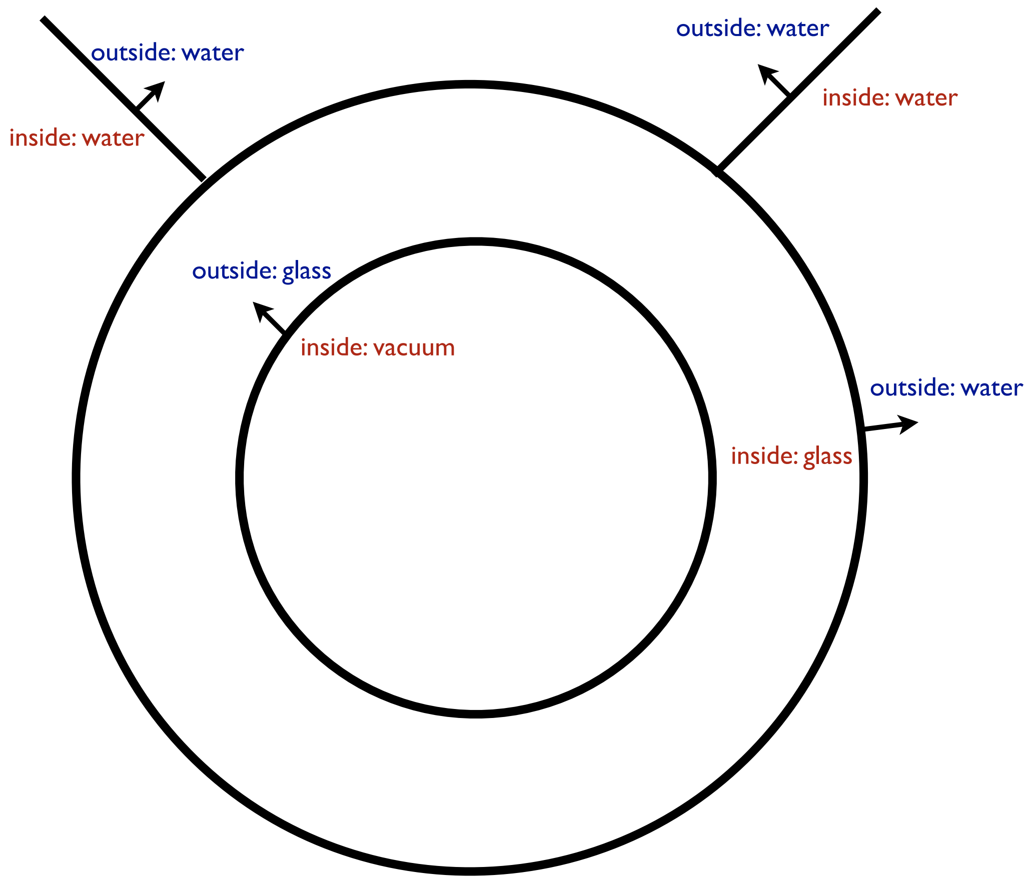

Figure 2: A 2D view of a spherical PMT and light reflector geometry showing the material consistency conditions. The arrows indicate the direction of the surface normal for each segment. Note that the reflector segments are not closed, and therefore the inside and outside material must be identical.

2.3 Representational Complementarity

A surface-based description of a detector geometry can be viewed as a dual representation to the equivalent solid-based description. There is a mapping of concepts between the two perspectives, as summarized in Table 1.

Concept	Solid (GEANT4)	Surface (Chroma)
Fundamental primitive	Many different solid types	Triangle
Primitive unit description	Parameters and a C++ class	Ordered list of triangle vertices
Geometry description	Tree of nested, non-overlapping volumes	List of triangles
Bulk material description	Each volume composed of one material	Each triangle has inside and outside material
Surface material description	A boundary between mother and daughter, or between siblings has a surface material	Each triangle has a surface material
Tracking data structure	Geometry tree plus voxels	Bounding Volume Hierarchy

Table 1: Concept mapping between solid-based models in GEANT4 and surface-based models in Chroma.

3 Requirements of an Optical Simulation

The motivation for *Chroma* comes from the need to propagate a large number of optical photons very quickly. Nearly all of the Monte Carlo time required to simulate an event in a water Cherenkov detector is taken up by photon propagation. Even a low energy (for LBNE) electron with 10 MeV of kinetic energy produces 5000 photons. Compared to other particles, optical photons are relatively simple to simulate, requiring only a handful of well-understood physics processes. This makes photons an ideal candidate for propagation using a dedicated fast algorithm designed for parallel execution.

Although one might imagine such a fast implementation would be useful for simulating events instead of the GEANT4 optics simulation, we believe that a fast optical simulation could be even more beneficial to event reconstruction. Reconstruction tasks often require the estimation of probability distributions for detector observables given some fundamental parameters, like event position and energy. Typically, one has to integrate over possible photon histories in order to create such distributions. Although some cases, like direct propagation from an event vertex to a PMT, can be evaluated analytically quite easily, secondary processes like scattering and reflection quickly make the problem very complex. Given a very fast optical Monte Carlo simulation, one could integrate over these photon histories directly just by creating the appropriate initial photon distributions for the interaction of interest and propagating them through a simulated detector and data acquisition system. Changes to the optics of the detector, such as the addition of reflectors or modifications of the cavity geometry, can be incorporated by simply updating the detector model. However, to imagine using a simulation live during event reconstruction, it must be very, very fast.

Software speed is always a mixture of efficient algorithms and capable hardware. By accepting the approximation inherent in a triangulated surface, we have created an opportunity to use very efficient tracking algorithms. At the same time, the computer industry is now producing hardware with a unique set of features to complement this approach. Manufacturers of graphics processing units (GPUs) have moved toward a more general computing model, compared to the fixed function 3D pipelines used in previous GPUs. A modern GPU is now a fully programmable, data-parallel floating point coprocessor optimized for high throughput. General purpose GPU (GPGPU) computing is still in its early years, so there are multiple, incompatible software interfaces. In 2006, NVIDIA released a special compiler and library interface, called CUDA, designed to allow their new GPU hardware architecture to be used for general purpose computing. Several hardware and software iterations later, CUDA is still the most mature and flexible interface for GPU computing, and so Chroma has been designed to use it[5].

Because CUDA requires the GPU-accelerated sections of the program to be rewritten in a new C++-like language, we have deliberately limited our particle simulation to optical photons. Only the following processes need to be implemented on the GPU for bulk materials:

- Propagation in bulk and refraction/reflection at media boundaries (index of refraction)
- Absorption
- Re-emission from wavelength-shifting absorber
- Rayleigh scattering
- Mie scattering

and for surface materials:

- Diffuse reflection
- Specular reflection
- Surface absorption

Most of these processes amount to a lookup in a table indexed by wavelength, a random number draw, and a simple transformation of the photon trajectory. These processes need to be applied in parallel to thousands of photons (or more) at once.

[5]Several vendors, including NVIDIA, are working on a cross-platform library, called OpenCL, that will allow data parallel calculations to be run on CPUs and GPUs of different vendors. OpenCL is still rather rough around the edges, but is similar enough to CUDA that we can port Chroma to the platform when it becomes advantageous in the future.

4 Detector Description

Given the discussion in the previous sections, we can propose a formal definition of a surface-based detector model. A detector consists of:

- A list $\vec{V}_i$ of three-vectors for all vertices in the model, where i indicates the vertex number.
- A list of integer 3-tuples $\overline{T}_j$, where j indicates the triangle number. The integer components of $\overline{T}_j$ indicate the vertex numbers that compose triangle j. The order of the vertices in $\overline{T}_j$ indicate the direction of the "outward" surface normal using the right-handed convention. Triangles are encouraged to share vertices with their neighbors to save on memory, although this is not required.
- A list of bulk materials, each identified by a unique integer ("material code") and consisting of the different wavelength-dependent functions used to describe index of refraction, absorption length, scattering lengths, etc.
- A list I_j of material codes that identify the material "inside" triangle j.
- A list O_j of material codes that identify the material "outside" triangle j.
- A list of surface materials, each identified by a unique integer ("surface code") and consisting of the different wavelength-dependent functions used to describe reflection and absorption probabilities.
- A list S_j of surface codes that identify the surface material on triangle j.
- A list D_j of *solid identifiers*. These are integers that are assigned to the triangles to allow the simulation to associate hit triangles with particular solid entities. For example, one could use these codes to identify the PMT ID number to which a triangle belongs. Strictly speaking, solid identifiers are not required for a surface-based optical simulation, but they facilitate the assignment of detected photons to the appropriate electronics channels.

In practice, the above lists are not specified by the user directly in Chroma. Instead, the detector is described programmatically as the union of several separate triangle meshes, either loaded from STL[6] or generated on the fly using functions that create meshes in various shapes based on input parameters. Meshes loaded from disk can also be rotated and displaced, allowing a single PMT mesh model to be placed many times in different locations.

5 Implementation

One could implement the optical simulation directly with the data structure described in the previous section. However, a large PMT-based detector like LBNE typically requires 50 million triangles, which would be terribly slow to work with. For each photon step, a ray would need to be tested for intersection with each of the triangles, resulting in a simulation far too slow to use.

[6]STL is a standard triangle mesh format that can be produced by nearly all 3D modeling programs.

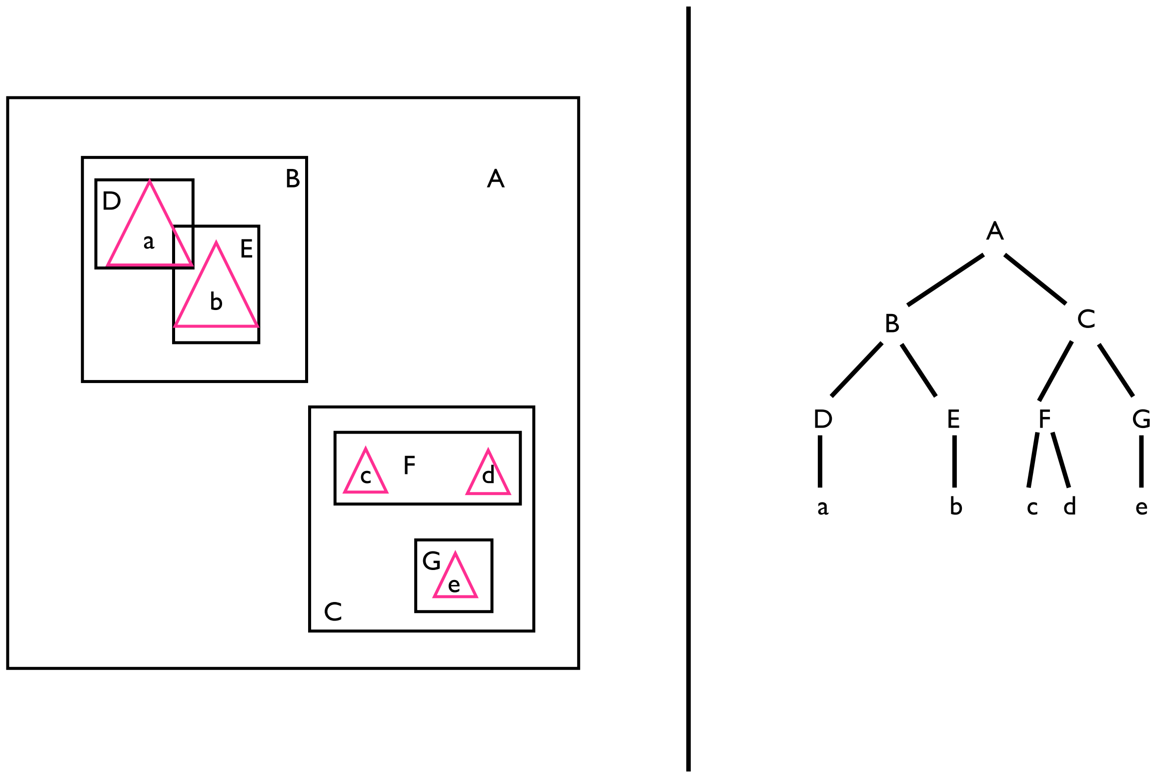

Figure 3: An example 2D bounding volume hierarchy. The left side of the figure shows the spatial relationships between the triangles and bounding volumes, and the right side shows the intermediate nodes and triangles in a tree. Note that bounding volumes D and E overlap, as is permitted.

In order to efficiently propagate photons, we need to generate from the detector description a data structure that can dramatically reduce the number of intersection tests required to find the closest triangle intersection point. A standard technique is to create a *bounding volume hierarchy* from the triangle list.

5.1 Bounding Volume Hierarchies

A bounding volume hierarchy (BVH) is a tree structure consisting of bounding volumes, usually axis-aligned boxes, that are nested within each other. Any given node in the hierarchy may have zero or more children, and the bounding volume for each child must be contained completely within the bounding volume of the parent. Children do not need to fully partition the volume of the parent, and sibling nodes are permitted to overlap. Figure 3 shows a bounding volume hierarchy in 2D space along with the associated tree structure.

The leaf nodes in the BVH contain subsets of the full triangle list. Although triangles may appear in multiple leaf nodes (since leaf nodes with the same parent may overlap), our bottom-up construction method, described in the next section, ensures that each triangle is assigned to exactly one leaf node. The containment condition ensures that in order for a ray to intersect a particular triangle, it must intersect all of the bounding volumes above it in the BVH, all the way back to the root node.

To test for intersections with a BVH, one starts with the root node and tests for intersection with the box associated with that node. If an intersection is found, then intersection must be tested for each of the children. Since siblings may overlap, it is not possible to take a shortcut and skip the rest of the children after the first hit. For each child that intersects, the process is repeated recursively until leaf nodes (if any) are reached. All the triangles associated with each hit leaf node must then be tested, and the closest intersection point identified. The number of intersection tests depends on the depth of the tree, and the average number of children per node, d. Assuming minimal overlap of siblings and optimal partitioning of the triangles, the number of box intersections required in the case where a triangle is hit is proportional to $d \log_d n$, where n is the number of triangles. For a model with 50 million triangles and a BVH with 2 children per node[7], this corresponds to approximately 51 intersection tests per ray, a million-fold improvement.

5.2 BVH Construction with a Z-Curve

The challenge in BVH construction is to achieve a very fine partitioning of the triangle list into leaf nodes, while also completing the tree construction in a reasonable amount of time. The approach that Chroma takes is to sort the triangles in such a way that adjacent triangles are near each other in space, and then to recursively group them in nodes working from the bottom of the tree to the top.

The first step in the BVH construction process used in Chroma is to compute for every triangle a *Morton code*. First, the 3D space enclosing the world volume is quantized into integer x, y, and z coordinates with some number of bits per coordinate. For most complex models in Chroma, we find that 8 bits is a minimum to achieve optimal performance. More bits per coordinate increases the size of the final Morton code, and also increases the amount of time needed to complete the tree. At the moment, Chroma stores the Morton code in a 32-bit integer, so 10 bit quantization on each axis is currently the maximum allowed. If needed, the quantization could be further extended by switching to a 64-bit Morton code.

For each triangle, the coordinates for the centroid of the three vertex points is computed and quantized to an integer (x, y, z) location. Then the *Morton code* is computed by interleaving the bits of x, y, and z to produce a single integer. If the bit representation of $x = x_3x_2x_1x_0$, $y = y_3y_2y_1y_0$, and $z = z_3z_2z_1z_0$, then the bit representation of the Morton code is $m = z_3y_3x_3z_2y_2x_2z_1y_1x_1z_0y_0x_0$. Table 2 shows Morton code generation process for several triangle examples. In effect, a Morton code flattens a higher dimensional space into a 1D space while attempting to preserve spatial locality.

[7]For this idealized case, the optimal number of children to have per node, on average, is the transcendental number e.

Triangle Centroid	Integer Coordinates	Morton Code
$(1.0, 2.5, 3.4)$	$(0, 1, 1)$	6
$(8.3, 5.1, 12.0)$	$(4, 2, 6)$	368
$(10.3, 5.1, 12.0)$	$(5, 2, 6)$	369
$(8.3, 7.1, 12.0)$	$(5, 3, 6)$	370
$(8.3, 5.1, 14.1)$	$(5, 2, 7)$	372

Table 2: Translation of the centroid of a triangle into a Morton code, assuming 3-bit coding for x, y and z coordinates ranging from 0 to 16. All numbers shown are base 10. Notice how the last 4 points have similar Morton codes indicating their spatial proximity.

If one draws a line in the N-dimensional space of cells, connecting them based on their Morton order, then a space filling curve called a *Z-curve* is created. Figure 4 shows a picture of the 2–4 bit curves in two dimensions, and Figure 5 shows the same curve in 3D with color coding to indicate the Morton ordering. These pictures highlight the fractal nature of the curve, which is useful for BVH construction. Right-shifting the Morton codes by 1 bit gives another Morton code in a more coarsely quantized space.

To build the BVH, all triangles with the same Morton code are placed in the same leaf node. The bounding volume for the leaf node is then computed from the min and max of all the triangle vertex coordinates (not the centroids). To group leaf nodes, the Morton codes for each leaf node are right-shifted by one bit, and nodes with identical codes become siblings under a new parent node. The bounding volume of the parent node is defined by the min and max box coordinates for the children. This continues recursively up the tree, until reaching the root node when all Morton code bits have been shifted away. The result is an incomplete binary tree, with nodes missing if those cells in the quantized space contained no triangles. Figure 6 shows a 3D view of a sample model and the boxes corresponding the nodes at different levels in the BVH.

Other techniques for BVH construction are possible, and some future directions for exploration are mentioned in Section 7.

5.3 GPU Processing

Although the CUDA documentation uses terminology that resembles CPU parallel processing with threads, the underlying GPU hardware behaves very differently than a multicore processor. In order to devote more of the chip area to arithmetic units, the GPU architects have stripped away most of the on-chip cache and simplified the instruction decoder and associated support circuitry, while increasing the width of the memory bus. Additionally, all high-end GPUs come with on-board memory (called "device memory" in the CUDA documentation) that is designed for much higher throughput than standard system memory. Table 3 lists the specifications of the GPU and CPU we used for testing.

Once the BVH has been computed for a model, the basic detector representation discussed in Section 4 is loaded into the device memory, where it remains unchanged

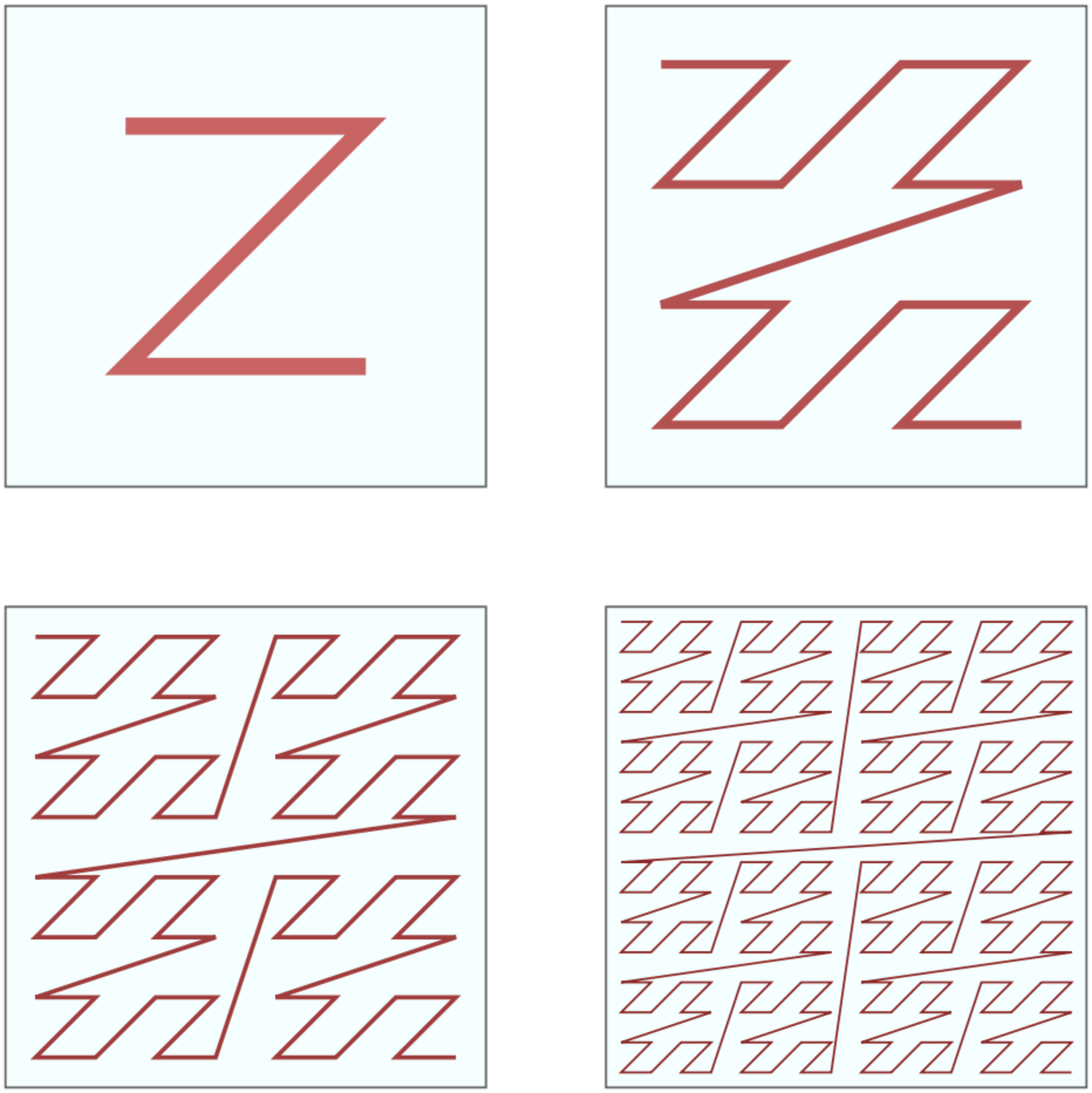

Figure 4: A drawing of a Z-curve for a 2D Morton code with 1, 2, 3, and 4 bits per dimension. The line connects points in the 2D space that will have consecutive Morton codes, indicating the order they will have in a 1D array. Figure from [1], used in accordance with the GFDL license.

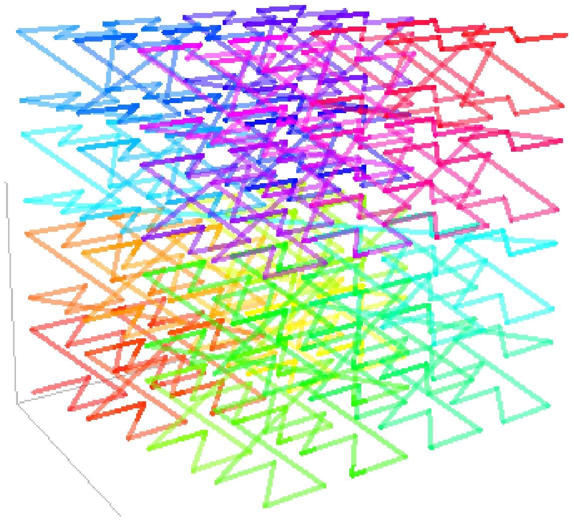

Figure 5: A drawing of a Z-curve for a 3D Morton code with 3 bits per dimension. The line connects points in the 3D space that will have consecutive Morton codes. The color ramp also indicates the 1D ordering. Figure from [1], used in accordance with the GFDL license.

	CPU	GPU
Model #	Intel Core i7-920	NVIDIA GeForce GTX 580
Transistors	0.731 billion	3 billion
Clock rate	2.66 GHz	1.544 GHz
Peak FLOPS	85 GFLOPS	1544 GFLOPS
Memory size	up to 24 GB	3 GB (special order)
Memory bus	192 bits	384 bits
Peak Memory Bandwidth	25.4 GB/sec	192.4 GB/sec
Peak Power Usage	130W	244W
Price	$290	$590

Table 3: Comparison of specifications between the high-end CPU and GPU used for Chroma testing. The GTX 580 has 512 arithmetic units divided into 16 groups, while the Core i7 has 4 general purpose cores. Note that FLOPS = single precision floating point operations per second.

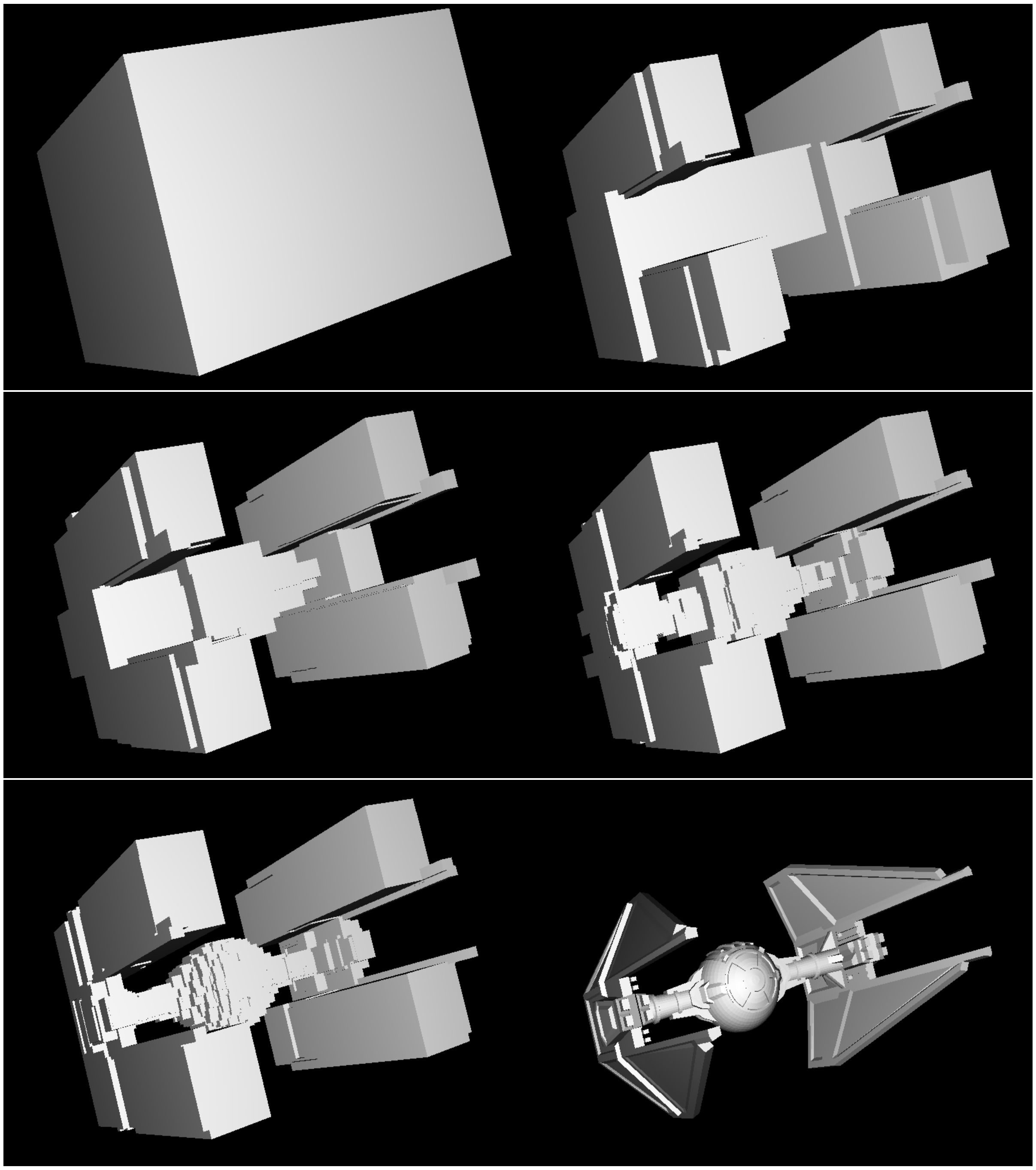

Figure 6: A rendering of the bounding volume boxes at different levels in the tree, starting with the root and moving down 3 layers in each image. The final image is the actual rendered triangle mesh.

for the entire duration of the program run. This includes the vertex list, the triangle list, the material lists, and the solid ID number for each triangle. Additionally, the BVH is loaded into the GPU device memory as a series of flattened arrays. Each node's bounding volume is defined by the coordinates of opposite corners, and the child nodes are organized such that they have consecutive indices in the node arrays. Instead of pointers, each node identifies its children using two integers: index of first child in the node array and the number of children. For leaf nodes, these indices identify regions of the triangle array that are children of the leaf node. Again, since the triangles are in Morton order, the children of each leaf are adjacent to each other.

With the data structure in place, Chroma offers two different kinds of calculation that can be performed with the detector model.

5.3.1 Ray-tracing

The ray-tracing[8] GPU function is used to render an image of the detector in a rectangular window. The ray-tracer is an important debugging tool since it uses exactly the same mesh intersection code called by the full optical simulation. It is also useful for visualizing the detector from different vantage points.

In ray-tracing mode, a pinhole camera is imagined in the detector model, with the film divided into pixels. From each pixel, a ray is sent out, through the pinhole, into the detector geometry. If it hits a triangle, the material codes of the triangle are interpreted as a color and the color intensity is scaled by the cosine of the angle between the normal vector and the incident ray. The pixel from which the ray originated is given this color. The effect is to render the scene as if a light source is present at the camera, and all surfaces are perfectly Lambertian diffuse reflectors. Figure 7 shows some sample renderings of various models made with Chroma.

5.3.2 Photon Propagation

The propagation of photons in the detector uses the same basic mesh intersection core. A list of photons is loaded into the GPU, including their initial start times, positions, directions, wavelengths, and polarization vectors. For maximum throughput, it is best to batch process as many photons as is practical, generally several thousand or more.

The propagation code loops a fixed number of times propagating each photon one step for each pass through the loop. We assign a thread to each photon, allowing them to each perform this simulation simultaneously. At each step, the photon ray is traced to the nearest triangle. The bulk material the photon passed through on the way to the surface can then be obtained from the inside or outside material code (depending on orientation of the triangle relative to the photon incidence angle) of the triangle. From the bulk material code, the appropriate indices of refraction, attenuation and scattering lengths for the current photon wavelength are obtained. Random numbers are drawn to decide if any of the bulk processes[9] stop the photon before reaching the

[8]Technically, the process described in this section should be called *ray-casting*, but the term *ray-tracing* is often used informally instead.

[9]Wavelength-shifting and Mie scattering have not been implemented yet.

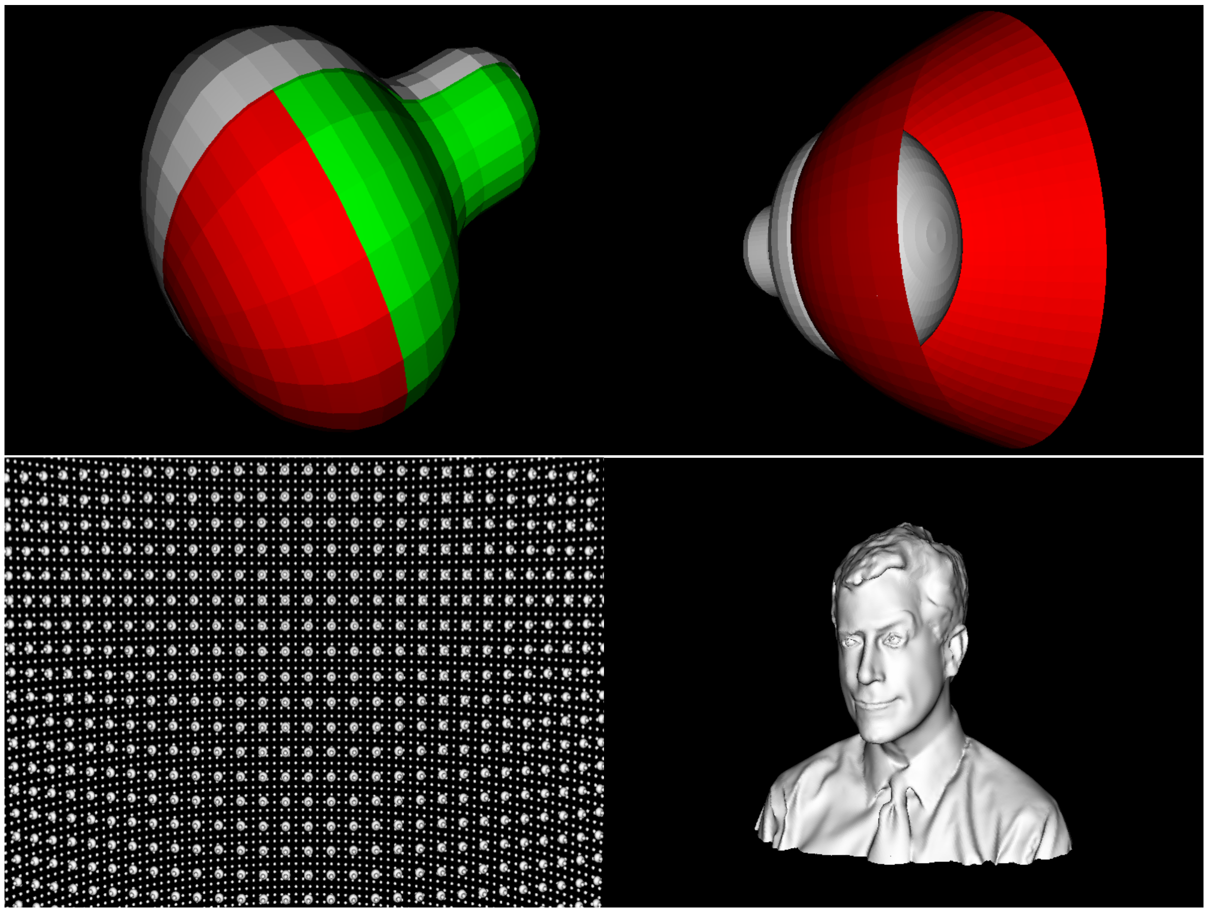

Figure 7: Sample Chroma renderings of a cutaway PMT model showing inner photocathode and reflector surfaces, a PMT model with a Winston cone reflector, the LBNE Water Cherenkov detector with the black liner sheet removed, and a famous guy.

nearest surface. If any of them do stop the photon, the new photon state is computed, and the thread moves to the next iteration of the loop.

If the photon reaches a triangle, one of two processes occurs depending on whether the triangle is associated with a surface material. The surface material, if present, acts as a convenient way to model two special cases of a boundary between two media: a film (whose microscopic physical parameters are unknown such as a photocathode material or a thin film in which it would be needless to create a separate closed mesh such as a mirrored surface) and the diffuse reflection of photons by an opaque solid. If the triangle does not have an associated surface material, the photon is either reflected or refracted according to the Fresnel coefficients determined by the refractive indices of the first and second bulk materials. If a surface material does exist, the photon is absorbed, specularly reflected, or diffusely reflected according to a probability for each determined by the surface material properties.

At the end of the iteration, each photon is marked with a state code indicating which process acted at the end of the step, and also the ID number of the last triangle that was hit, if a surface was reached. The propagation loop continues up to the number of iterations specified in the function call. For maximum speed, several iterations in a row should be performed in one GPU function call, but if step-by-step information is required, the GPU can be told to only propagate one step before returning back to the CPU caller.

5.4 DAQ Simulation

For reconstruction purposes, we want to be able to convert the list of final photon states computed by the propagation function into hit times on the PMT channels. We can do this operation directly on the GPU to avoid the overhead of copying the photon information back to the CPU over a relatively slow PCI-Express bus. The very simple DAQ simulation currently implemented in Chroma applies a random time jitter to the photon absorption time at the photocathode and uses the mapping from triangle ID to solid ID number to determine in which channel the photoelectron would be detected. The earliest time is recorded as the hit for the channel. Although crude, this simple DAQ is sufficient for testing of position reconstruction using Chroma to generate hit time PDFs for all the PMTs.

6 Example Usage

Although the basic BVH construction and mesh intersection code in Chroma is complete, the infrastructure around it is still evolving. In this section, we will highlight some different examples of Chroma usage for both detector visualization and photon simulation.

6.1 Photon Tracks in a Lens

A very simple test of Chroma is to send parallel photons through a spherical lens to observe the effects of the index of refraction and attenuation length. Figure 8 shows

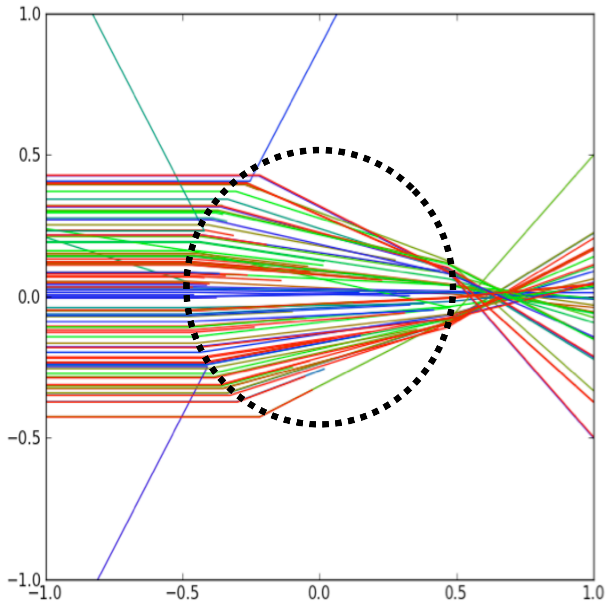

Figure 8: Photon tracks from parallel rays hitting a 1m diameter sphere of glass, with a flat wavelength distribution. The approximate location of the surface of the sphere has been marked with a dashed line. The track colors indicate the photon wavelength.

the photon tracks for a collection of rays passing through such a lens. The focal point of the lens is not a point due to a combination of the triangular facets on the sphere and chromatic aberration of the different wavelengths.

6.2 Display of Position-Dependent PMT Properties

One of the early uses of Chroma was to display the results of 2D scans of the 12" PMT face with a Cherenkov light source. After 4 scans across the diameter of the tube, rotated 45 degrees each time, we had sampled the relative efficiency, the relative mean transit time, and the transit time spread over the PMT at 200 locations. Figure 9 shows a rendered image of this data set displayed on a 3D model of the 12" PMT. The sample points were interpolated in polar coordinates to determine the value of the response function at the centroid of each triangle. Rotating the model in the live Chroma viewer gives the user a good intuition for how the local variation in the PMT response over the face can result in a shift in the average response when the PMT is viewed from different angles.

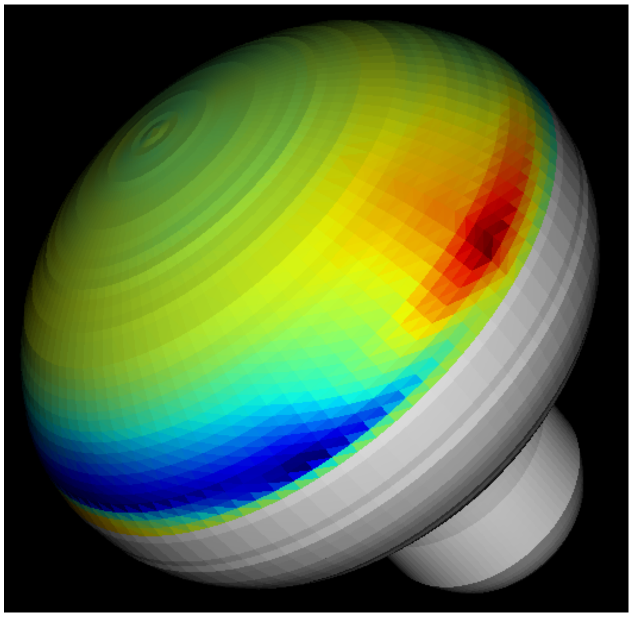

Figure 9: Chroma rendering with triangles colored with a 2D interpolated data set showing the mean transit time for single photoelectrons relative to the center. Photoelectrons from blue regions arrive 3 ns early, and photoelectrons from red regions arrive 3 ns late.

6.3 MiniLBNE Water Cherenkov Detector

For the purposes of developing a function minimizer that can operate on stochastic, Monte Carlo-derived likelihood functions[10], we have created a 1/10 linear scale LBNE water Cherenkov detector, called “MiniLBNE”, shown in Figure 10. This detector is much faster to load and work with than the full size LBNE, allowing for easier experimentation with the minimization algorithm. A rendering of MiniLBNE is shown in Figure 10.

For reconstruction purposes, we do not want to use the ray-tracer, nor do we need full photon tracks. Instead, we want to take the photons that hit PMTs and generate a timing PDF across many events that can be used in the likelihood function evaluation. To reduce overhead, we do this by leaving the final photon states on the GPU and running an additional GPU function to identify the detected photons, apply the PMT timing response[11] and then determine the earliest hit time (if any) for each PMT ID in the event. Figure 11 shows the hit time distribution for a particular PMT when the event is in the center versus directly in front of the PMT. In addition to the reduction in the propagation time from the light source, the timing distribution also narrows as expected from multiple photons hitting the same PMT. The timing distribution for the entire detector with isotropic light at the center is shown in Figure 12, which shows the contribution of both direct and scattered light in the simulation.

[10]This project will also be the subject of an upcoming memo.
[11]For now, a simple Gaussian prompt peak.

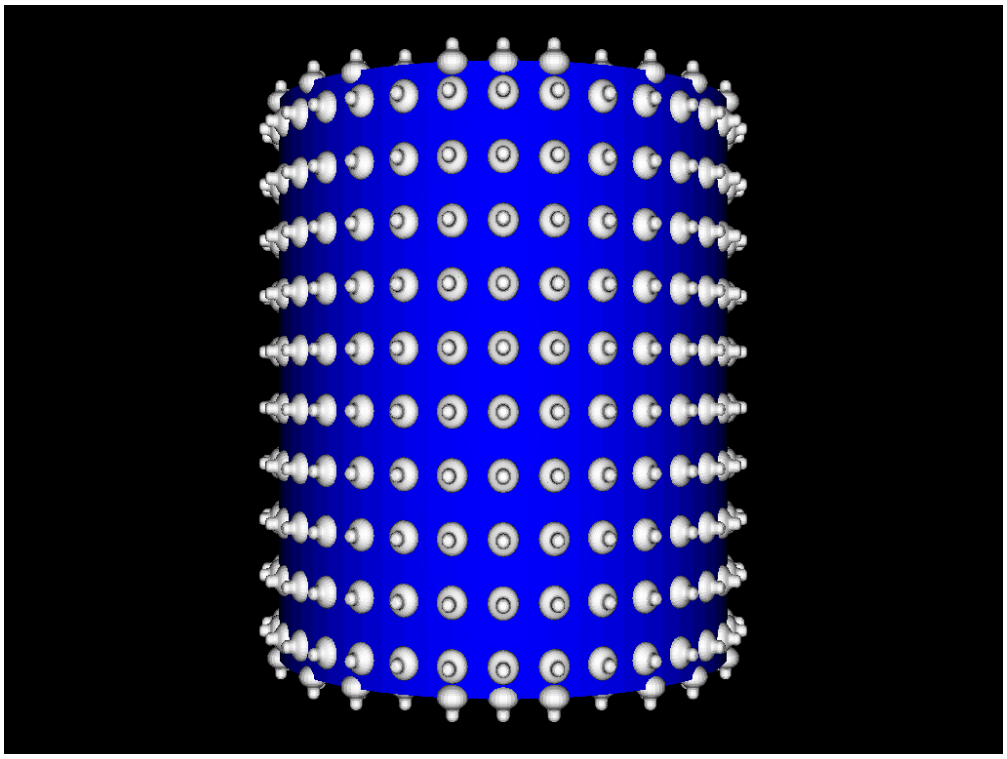

Figure 10: Chroma rendering of external view of MiniLBNE with 12" PMTs.

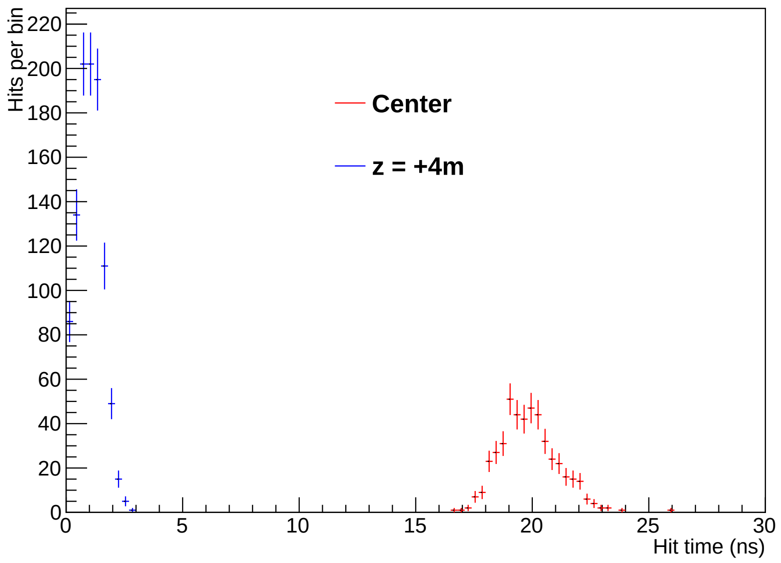

Figure 11: The hit time recorded by PMT channel #224 for an isotropic event at the center vs. nearly in front of a PMT in the MiniLBNE detector. (For illustrative purposes only! This simulated data does not use a realistic PMT efficiency curve or wavelength distribution.)

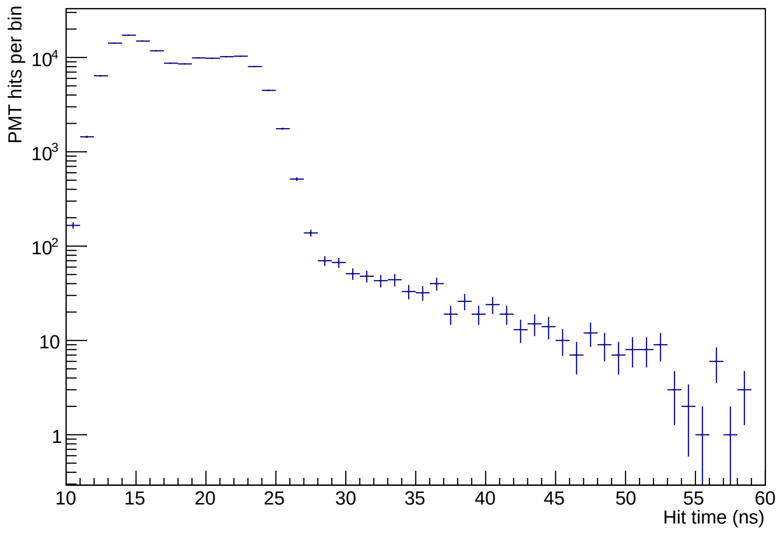

Figure 12: The hit times recorded by all PMT channels for an isotropic event at the center of the MiniLBNE detector. (For illustrative purposes only! This simulated data does not use a realistic PMT efficiency curve or wavelength distribution.)

	Count	Memory Usage
Vertices	20.2M	241.9 MB
Triangles	40.1M	641.3 MB
BVH nodes	5.2M	165.8 MB
Total		1049.0 MB

Table 4: Memory usage of full size water Cherenkov detector model using a truncated PMT mesh without light concentrators.

6.4 Full Size LBNE Water Cherenkov Detector

The ultimate goal, of course, is to simulate photon propagation in the full size LBNE water Cherenkov detector. For a 200 kiloton cavity, as described in the recently produced case study, with 29000 PMTs[12], the detector model requires a very large amount of GPU memory. Currently, we simplify the PMT model by truncating it a short distance past the equatorial region. This cuts the memory usage in half, allowing the detector to fit on a broader range of GPUs. Table 4 lists the memory required for different parts of the detector model.

With this configuration, the simulation can propagate 10 million track steps per second with no physics simulation. Including the full physics simulation, more than 1 million photons can be propagated to completion per second. The detector model in WCSim can propagate roughly 30,000 photons per second with a highly-tuned geometry on a Core i7 2.6 GHz Processor. Future work to more efficiently pack replicated triangle meshes will cut the memory usage significantly, allow the full PMT model to be used again, and possibly improve performance further.

Although we are not yet using the full size water Cherenkov detector for development of reconstruction, we can in the near term verify the optical model of Chroma against the well-tested optical models of WCSim and the SNO+ Monte Carlo. Figure 14 shows the timing distribution for isotropic light emitted in the full size water Cherenkov detector model with no light concentrators. This distribution is just for illustrative purposes and has not been compared to any standard Monte Carlo results.

7 Conclusions

7.1 Summary

In this document, we have outlined a very different approach to fast particle simulation that shifts the focus from solid volumes to the boundaries between volumes. Surface-based modeling is a completely consistent and equivalent approach to solid-based modeling that has been well-studied in the computer graphics community for 3D rendering purposes. We are interested in this surface-based approach for its ability to

[12]The 29,000 PMT estimate assumes the addition of some kind of light enhancement technology to the PMT, such as Winston cones or wavelength-shifting plates.

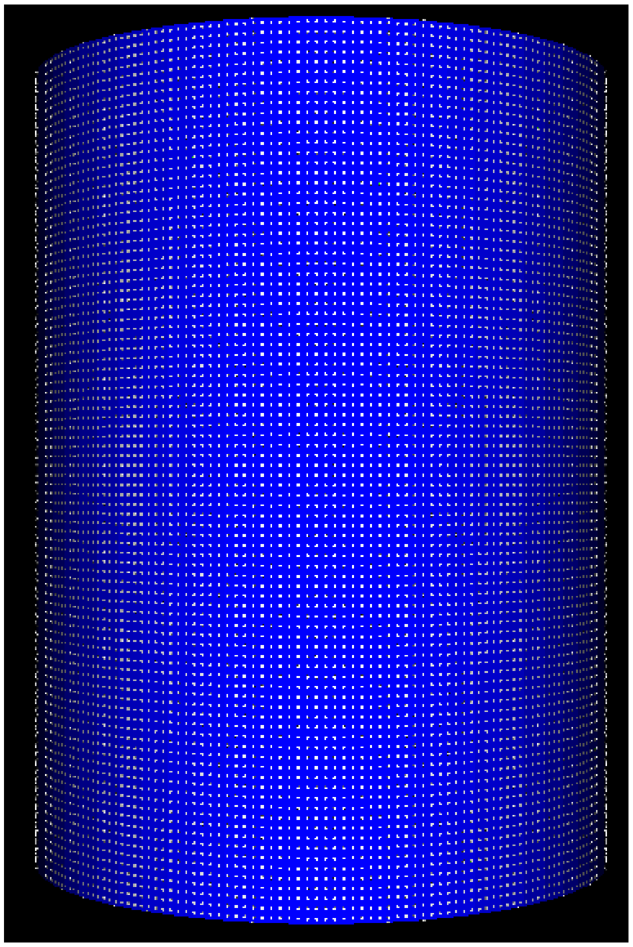

Figure 13: A Chroma rendering of the outside of the LBNE water Cherenkov detector. Blue indicates the opaque cavity liner, and the white dots are the back sides of the truncated PMTs poking through the sheet.

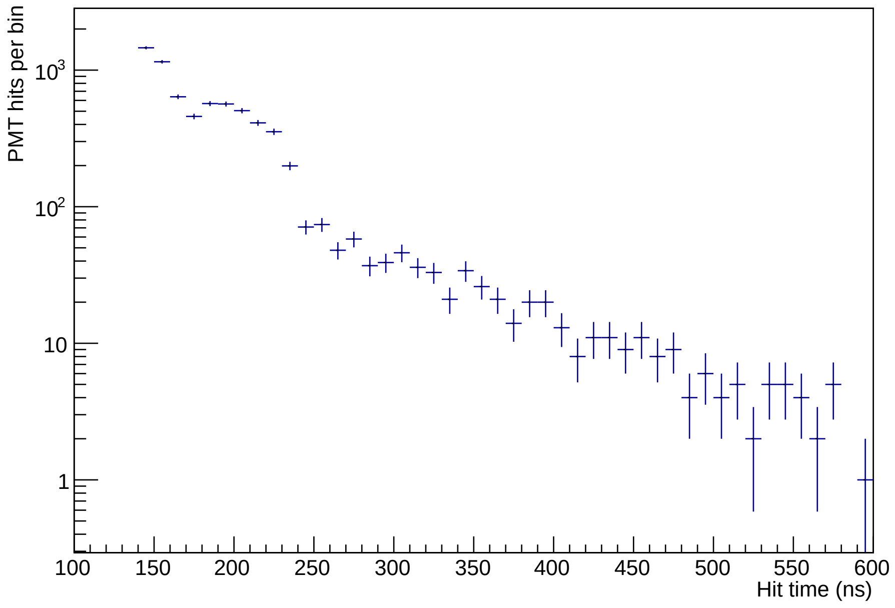

Figure 14: The hit times recorded by all PMT channels for an isotropic event at the center of the full size water Cherenkov detector with no light concentrators. For illustrative purposes only! This simulated data does not use a realistic PMT efficiency curve or wavelength distribution.

facilitate the propagation of photons through a detector model nearly 100 times faster than GEANT4. This speed increase is due to both the algorithmic improvements possible, and the ease with which the calculation can be offloaded to the massively parallel hardware present in today's GPUs.

The surface-based detector model consists entirely of repeated triangular patches, tiling the surface of every boundary between volumes. The triangles are annotated with material codes and ID numbers to allow them to be associated back to particular objects of interest (like specific PMTs). Surface-based detector models sacrifice some accuracy in the representation for curved surfaces in exchange for the opportunity to heavily optimize track propagation. The level of approximation can be controlled, however, by reducing the size of the triangular patches used to map the surface of the object. Thanks to the bounding volume hierarchy, the resource usage of adding more triangles may be linear in memory, but is sub-linear in processing time due to the assistance the tree structure provides in track propagation.

We also have described a technique for fast construction of bounding volume hierarchies using a Morton ordering of triangles. This technique is relatively fast and quite effective at speeding up intersection tests. Chroma uses the same BVH and intersection implementation for both ray-tracing the detector model to display it, and for photon propagation through the detector.

Chroma can now simulate approximately 1 million photons per second in a full size LBNE Water Cherenkov detector, including physics processes like absorption, Rayleigh scattering, Fresnel reflections, and refraction. We are also using Chroma to assist in the development of a Monte Carlo-based maximum likelihood reconstruction algorithm using a reduced detector for now, moving up to the full size detector in the future.

7.2 Future Directions

In the medium-term, we need to verify that Chroma can reproduce the optical properties of currently verified detector simulations, like WCSim and the SNO+ Monte Carlo. However, in the short term we would like to investigate some refinements to the detector description in order to reduce the GPU memory required to simulate full-size LBNE. This will allow Chroma to run on more standard GPUs with less than 1 GB of device memory.

First, we would like to extend the detector definition to more efficiently represent replicated triangle meshes. A water Cherenkov detector has the same PMT surface repeated many times, differing only by a vector displacement followed by a 3D rotation. If we could capture the repetition more directly in the detector model, the number of triangles could be reduced by a factor of 30,000. Moreover, this information could also accelerate BVH construction while also enhancing the granularity. The PMT could be split into a BVH in local coordinates, and then linked to a larger BVH with the inclusion of coordinate transformations at some nodes. This is somewhat similar to how GEANT4 deals with repeated volumes. Some care will be required to include repeated and non-repeated meshes in a general way.

Second, we may also want to consider surfaces with a non-planar primitive. Curved surface patches would allow Chroma to represent objects like PMTs with far fewer elements, further saving memory. There are various standard specifications for curved triangles that are created by attaching a normal vector to each vertex and interpolating the normal vector between the vertices. The key requirement for our implementation would be that the additional data storage and computation does not adversely affect the runtime.

Finally, we need to further explore the issues with using multiple GPUs in parallel to simulate many events for likelihood calculation purposes. Our primary development computer contains 5 CUDA devices, and preliminary tests show that Chroma is not scaling past 2 devices. We are not sure what the bottleneck is yet, but we do believe this can be fixed.

Chroma is currently written in a mixture of Python[13] and CUDA C. This has been tremendously useful for rapid development, but will probably create difficulties integrating Chroma with more standard C++ tools. Once development has stabilized, we will investigate creating a C++ wrapper around the GPU code that will allow ray-tracing calls to be made by other tools as part of hybrid analytic-Monte Carlo reconstruction techniques. One could even imagine experimenting with using Chroma as a replacement for the G4Transportation process to use GEANT4's particle simulation code with the Chroma geometry[14].

We are not entirely sure how Chroma will evolve, but we hope that it will add to the particle physics simulation toolbox in ways we don't yet foresee.

References

[1] Z-order curve. http://en.wikipedia.org/wiki/Z-order_curve. Accessed on July 8, 2011.

[13] Note that we have made extensive use of Numpy in order to achieve the speeds we describe in this document. Python doesn't have to be slow!

[14] One would still want to capture all the initial photon vertices from GEANT4 and propagate them as a batch at the end of the event.