Physically-Based Laser Simulation

Greg Reshko
Carnegie Mellon University
reshko@cs.cmu.edu

Dave Mowatt
Carnegie Mellon University
dmowatt@andrew.cmu.edu

Abstract

In this paper, we describe our work on physically-based simulation of lasers. Currently there are a great variety of computer games featuring lasers as weapons; however, we are not aware of a single game that actually models the laser according to a strict physical model, as opposed to some simplistic ad-hoc approach, such as blowing up the car to pieces whenever the laser hits it.
For this reason, we decided to implement an actual physical model of a laser, as well as to write a very simple game to act as a test-bed for the project.

Overview

A laser is a rather complex phenomenon, which requires a number of approaches both from graphics as well as physics. We will first outline the principles behind how we decided to model the laser and why. We then will continue onto several interesting issues that arise while dealing with this problem and solutions we worked on.

Laser Principle

In order to create a physically valid model of our laser, we need to first look at what exactly laser is and how it works. Obviously, we do not need to go into details on how lasers are actually constructed or operated, but we need to understand one simple, but important concept – a laser is very different from knife. By this we mean that laser cuts fundamentally differently from a cutting object such as a knife or a chainsaw, in a sense that it essentially continuously drills the object, as opposed to actually slicing it. What this means for the simulation, is that we cannot apply any algorithms related to cutting, since we cannot compute the actual 'cut'. The only thing we can do is to compute which faces will be 'destroyed' when the laser hits them. Whenever we move the laser, it is the same as moving the drill-bit sideways – it will still be drilling forward, not to the side.
This observation is obviously not extremely advanced, but it does have very important implications in terms of how we can model the laser. Since we cannot model it as a cut, we will have to model it as a 'drill', and thus we will need to utilize mesh-to-ray intersection model. This brings us to the next topic in our simulation – how do we detect the intersection between the mesh and the laser.

Laser Detection

As we already mentioned, our model of a laser will be represented by a ray. In order, not to destroy every object in the world that lies on the path of our laser, the laser ray will most likely have a minimum and maximum bound on its length.

So now we have a mesh that we trying to cut with a laser, and a laser represented by this ray. We need to determine whether the mesh is being hit by the laser, and if so – which triangles are being cut.
Traversing the entire mesh face by face is not acceptable, since that results in linear running time and that is rather slow for interactive simulations. We chose to use Axis Aligned Bounding Box trees as a speedup for our simulation.
AABB trees are constructed by recursively subdividing the mesh into bounding boxes, until every box contains only one triangle. It is rather expensive to recompute the entire mesh at run-time, so instead the AABB tree for the mesh is computed before the simulation starts and is used throughout the simulation. Below we illustrate how to compute AABB tree for the car mesh, and then how the tree is used to perform the collision detection.

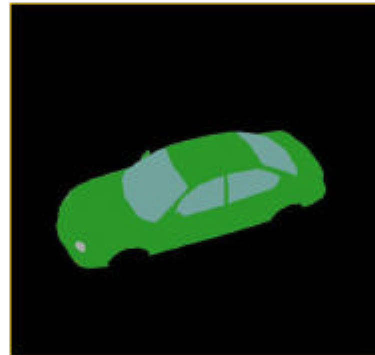

Figure 1. Original Mesh

After we load the mesh, we construct AABB tree for it. The tree is then used to perform fast collision detection between the laser beam and faces of the mesh. The screenshot below has red bounding boxes if they are or were hit by the laser and green boxes if they are untouched by the laser.

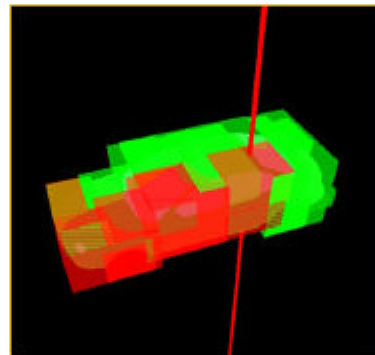

Figure 2. AABB Tree showing collision with the laser beam.

The list of faces that need to be destroyed is easily obtained from the AABB tree and the corresponding faces are removed. For simplicity and due to DirectX implementation of this project, we

are just removing index buffer entries corresponding to these vertices. The result is a mesh with cut-out faces that were 'cut' by the laser.

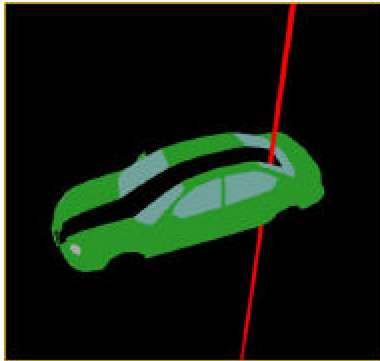

Figure 3. Mesh cut by the laser.

In order to detect collision between the mesh and the laser, we traverse our AABB tree – this is both fast and efficient. The traversal will give us a list of faces that are being hit by a laser in case there is more than one (i.e. the mesh can be as complex as the creator wants it to be). An obvious and simple approach that we tried is that whenever we cut a face, we simply remove it. However – there is an obvious problem with that approached and it is rather well visible in the screenshot above – usually the faces for the meshes are very non-uniform and usually are rather big. That explains why in the above screenshot, the cutout is almost 10 times bigger than the thickness of the actual laser beam.
One solution to this problem will be to have a very fine resolution mesh, however this is virtually not feasible, since the laser is very thin and thus the mesh size will be impossible to render in real-time.
Multi-resolution meshing seems like an appropriate approach to this problem. We note that AABB trees are a very good model for this simulation, since they naturally allow additional 'splits' of the faces in the mesh into several faces, which is required by a multi-resolution approach.

Laser Multi-Resolution

As stated above, multi-resolution approach seems to be required in this model of the laser, due to laser's infinite precision in cutting.
In basic terms, multi-resolution meshing means that whenever we hit a triangle with the laser – we look at the size of the triangle. If the triangle is very small – we just remove it and if it is big enough, we split into several smaller triangles. Here is the conceptual representation:

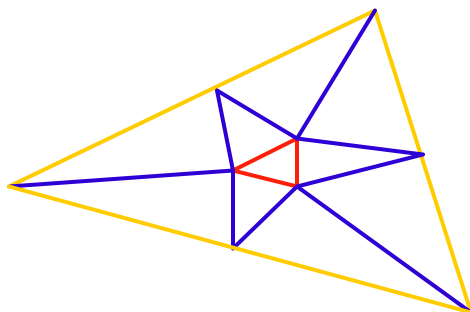

Figure 4. Multi-Resolution Approach.

So in terms of the mesh objects, this means that we need to take the triangle out of the mesh, compute several smaller triangles, insert them back into the mesh, and recomputed the collision detection.
We implemented a multi-resolution approach for our meshes, using 1-to-4 triangle split. This means that whenever a large-size triangle is being cut, it will be split into 4 smaller triangles and the cut will be performed on these sub-triangles. An interesting fact about multi-resolution approach is that we can easily control how 'fine' the laser cut is – i.e. the cut can be as coarse or fine as we want. This can save us computational time in some instances, while yielding great performance in others. Two screenshots below demonstrate a very coarse cut followed by a very fine cut, produced the same simulation, just with different parameters.

Figure 5. Multi-Resolution mesh with a coarse cut.

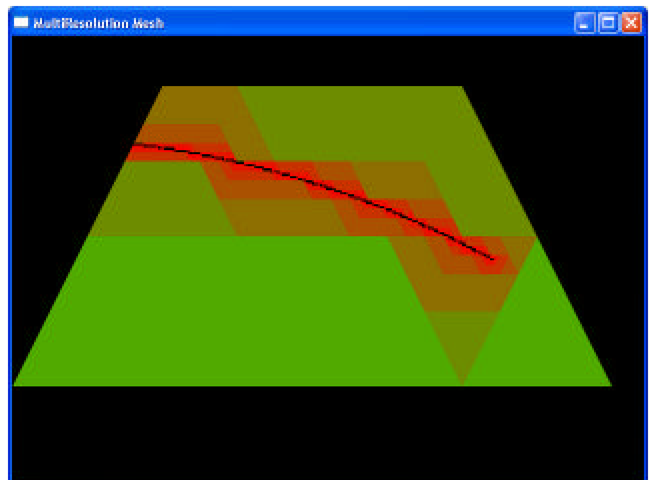

Figure 6. Multi-Resolution mesh with a fine cut.

We clearly see how the multi-resolution approach allows us to fine-tune the cut. We also note that to yield more realistic results and since a laser beam does have a thickness, we fire four rays at our mesh instead of one – this gets rid of artifacts as well as gives thickness to the cut.
Another rather interesting thing to notice about this approach is that is very natural for a laser cut. The pictures above demonstrate the same temperature gradient as observed in actual laser cutting. Smaller triangles, those triangles that are closer to the cut, are colored with brighter color while those that are far away have nearly the same color as before the start of the simulation. This

naturally corresponds to the actual cutting process in real life and in some sense means that this approach is physically valid.

Laser Mesh Separation

Multi-resolution approach with AABB collision detection seems to work rather well for our purposes of cutting the mesh with our virtual laser. However, one problem still remains – when we cut the mesh into halves – how and when do we determine the mesh has actually been cut in half? A mesh that has two visible parts is still one mesh, just with quite a few vertices missing. Below is an illustration of what we need to do:

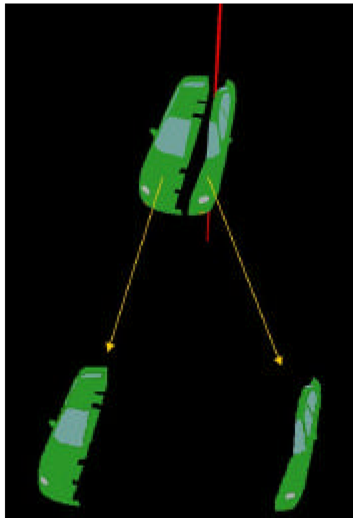

Figure 7. Mesh Separation.

So now we will address the problem of determining how to split the mesh. We note that this problem greatly depends on how we represent the mesh – it depends on whether we use index and vertex buffers, adjacency buffers, or simply a buffer of vertices with 3 vertices per triangle with none of the triangles sharing any vertices. We will first look at this last case, since it seems to be the most straightforward.
The triangles are represented as an array of vertices. This representation is not extremely useful, so we need to convert into a graph structure first. We construct a graph with nodes representing triangles. Nodes have edges between them only if the corresponding triangles share an edge between them. Note that this graph can be constructed very easily if the mesh has an internal adjacency list representation.
This traversal of the vertex array gives us a graph. Nodes are connected by some path if it is possible to reach one node from another by going through some edges in between. In terms of the mesh, this means that two triangles are in the same mesh if it is possible to reach them by going through neighborhood faces. If the graph turns out to have two 'clusters' of nodes – this means the mesh has a distinct cut through it and it can be split.
The actual algorithm works as following. We start at some node in the graph after constructing it from the vertex array. We run BFS or any other traversal algorithm suitable for this purpose to determine which nodes can be reached from the current node. We mark every node that we traverse. As soon as there are no more nodes to traverse, we count the number of nodes traversed. If the count is less than the total number of the triangles in the mesh – we split the mesh. We put nodes that were marked in one mesh and those that remained unmarked into another. This provides a clear and robust method of determining whether the mesh has been cut. The robustness comes from the fact that this algorithm does not take into a consideration the 'shape' of the cut itself – thus the cut can be straight or it can be in some really complex shape – the algorithm will compute the two subsets correctly in either case. Conceptually the separation algorithm is illustrated below.
First we number all faces that are valid in our mesh:

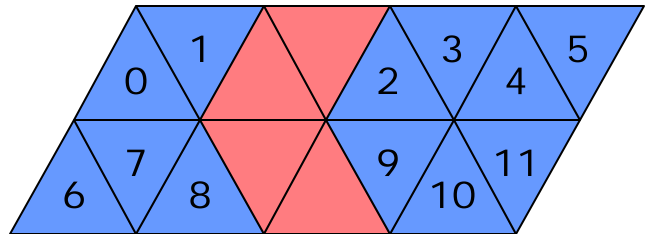

Figure 8. Mesh with a cut. Valid faces are blue and numbered, cut out faces are red.

Now we construct a graph based on the numbered mesh above. We represent triangles as nodes and shared edges between triangles as edges in the graph. We now run a traversal algorithm on this graph and determine if it can be split.

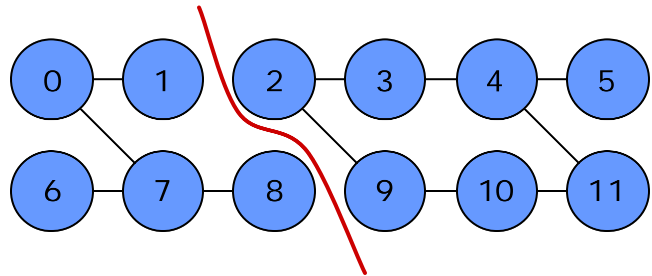

Figure 9. Graph representing the mesh. Red cut in the middle represents the graph being split into two clusters.

Obviously this graph consists of two distinct clusters of faces – thus the mesh can be split into two parts according to the above representation.
This more or less concludes the laser portion of our project and of this paper. We will now talk about the test-bed we created and several relevant topics.

The Game

All of the above concepts have been tested individually to assure that our math and implementation are correct. In order to demonstrate the above concepts altogether, we decided to create a simple test-bed. The screenshot is below:

Figure 10. Game screenshot.

We chose a simple first-person shooter game as our testing environment. The game has a car with a laser beam on top of it. The user can fire the laser at enemy vehicles and essentially cut them to pieces. We are not creating a commercial application and thus the graphics and scenario have a lot of room for improvement, but we rather concentrate on the physical validity of the game. In addition to the laser, we also looked at another interesting thing – shadows.

Shadow Volumes

A basic stencil buffer algorithm is used for applying shadows. A shadow volume is constructed for each mesh. This volume is rendered to the stencil buffer, leaving a mask of positive values where the shadows should be drawn. A transparent black rectangle is rendered across the entire screen using the stencil buffer to only draw where a shadow occurs.

The shadow volume is built using the edges of polygons visible to the light source. Any duplicate edges are removed, leaving only the edges of the silhouette. This is shown in Figure 1, where the silhouette is highlighted in blue. Each of these edges

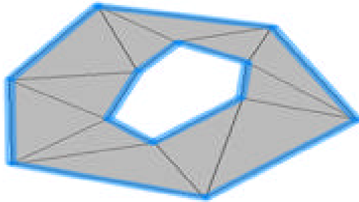

Figure 11. Mesh Silhouette.

are extruded away from the light source for some distance. This distance is set to be something reasonably high for the scene; it represents the maximum distance away the object will be able to cast its shadow. The result is an uncapped volume which encloses the space where the light should not reach.

All front faces of the shadow volume are rendered to the stencil buffer using a stencil operation that increments the values everywhere the shadow is drawn. The rendering also uses depth testing against the full scene which has already been rendered without shadows. Next all the back faces of the shadow volume are rendered using a stencil operation which decrements all the values where it is drawn. The back side can be drawn by switching the culling mode. The result of this operation is the stencil buffer has a positive value anywhere on the screen where

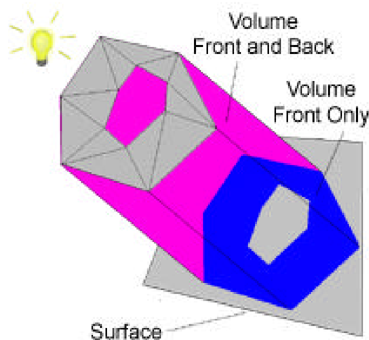

Figure 12. Shadow Volume.

the front of the shadow volume is visible but the back is obscured by some surface. This situation occurs whenever an object intersects the shadow volume, obscuring all the back faces of the volume behind the surface, as shown in Figure 2. The area in the diagram which has only the front of the volume is where the stencil buffer has a positive value.

Finally, a transparent black is rendered to the screen to actually draw the shadow on any objects where the stencil buffer has a positive value. This is done by rendering a transparent black quad covering the entire screen using the stencil buffer as a mask. The results are realistic shadows created by the object blocking light from a single light source.

Laser Compatible Shadow Volumes

When using a laser with hollow meshes, culling must be turned off. However, shadow volumes use culling to draw the front and back sides of the shadow volume. The method for this situation is to construct two shadow volumes, one for the polygons facing the light (as is normally done), and an additional one for polygons facing away from the light. The second shadow volume is for the polygons that are normally dropped by culling. The motivation for using two shadow volumes is that constructing a single shadow volume with all the polygons together would eliminate the outer edge of the silhouette. Both volumes could then be rendered to the stencil buffer and shadows will show up for all polygons that may be exposed to the light, regardless of orientation. Figure 3 shows an example of a shadow cast by un-culled polygons which have been exposed to the light after a laser deformation.

Figure 13. Un-culled Shadow.

Summary

In our opinion, this project was both very educational and interesting. We implemented a number of things that are very non-trivial and produced some interesting results.

We ran into several difficulties in regards to the implementation – in particular DirectX 9 mesh representation seems rather complicated. For this reason, we tested all algorithms on a simpler vertex-buffer model before moving to the mesh model.

The performance of our implementations seems to work rather well; however, the game itself can definitely be improved. That includes both the graphics and the scenario.

We envision adding robust mesh-to-mesh collision detection, such that the vehicles can collide with each other, as well as some other features that will make the game more interesting.

In regards to the goal of this project, i.e. to create a physically-based laser simulation – we believe that we achieved that with satisfactory results as demonstrated in screenshots in this paper as well as attached videos.

References

BARAFF D., WITKIN A. 2002. PHYSICALLY BASED MODELING. IN PROCEEDINGS OF SIGGRAPH 2002, ACM PRESS / ACM SIGGRAPH, COMPUTER GRAPHICS PROCEEDINGS, ANNUAL CONFERENCE SERIES, ACM.

BERGEN, G. EFFICIENT COLLISION DETECTION OF COMPLEX DEFORMABLE MODELS USING AABB TREES. JOURNAL OF GRAPHICS TOOLS, 4(2):1--13, 1997.

GRINSPUN E., KRYSL P., SCHRÖDER P., CHARMS: A SIMPLE FRAMEWORK FOR ADAPTIVE SIMULATION, ACM TRANSACTIONS ON GRAPHICS. 21(3), PP. 281-290, 2002.