Fire simulation and rendering in Beowulf

Magnus Wrenninge †
Sony Imageworks

Vincent Serritella †
Sony Imageworks

Theo Vandernoot ‡
Sony Imageworks

Henrik Falt §
Sony Imageworks

Patrick Witting ¶
Sony Imageworks

1 Introduction

In a world without electricity, fire proved to be an integral part of Beowulf's 900+ all-CG shots. Given the range of fire effects required – from simple torches to a fire-breating dragon – we needed a pipeline that could support both prop-based fire as well as complex and heavily art-directed shots. Before the show was over, we had taken effects artists out of the equation for a majority of the shots and replaced the entire rendering solution in the process.

2 Simulation

During Ghost Rider, Sony's proprietary fire pipeline (spFire) was developed to handle the many hundred fire simulations needed. It let effects TDs set up and control simulations in Houdini while leveraging off of Maya's fluid solver for the actual simulation. This foundation worked very well, letting us use all our existing effects animation tools in Houdini at the same time as Maya's fast CFD solver, built-in forces and user interface.

An extensive amount of the fire in Beowulf consisted of background elements, so a library was set up with pre-simulated torches, stockade cauldrons etc. The artist lighting a particular shot could then easily browse this library and populate the scene as needed. This way, effects artists could spend their time on shots requiring more complex and art-directed fire setups.

The most demanding fire shots on the show involved the dragon breaking out of its cave, setting fire to trees and bushes, then to a bridge and to Beowulf's castle. These shots had to be broken down into many separate buffers which were simulated separately but could be rendered together. In some cases, over 30 individual simulations were needed to complete a single shot.

Proprietary cache files from the fire simulations were also used to drive secondary animation. Particles were advected through the velocity fields for particle based smoke and embers, and geometry was scorched by fire interaction by exporting point clouds to RenderMan which would then affect surface shaders.

3 Rendering

On Ghost Rider, the spFire pipeline employed custom PRMan DSOs to handle rendering, but the process had limitations and complex scenes required extensive Z slicing to get around the memory implications of using millions of semi-transparent RiPoints. To get around these limitations, we added support for fire rendering to Sony's in-house volume rendering package Svea. Svea is integrated into Houdini (for volumetric modeling) and Katana (our in-house lighting and compositing package).

With hundreds of fire shots in the movie, rendering speed turned out to be a especially important. Fire rendering is especially hard on raymarchers as the sharp flame interface introduces much higher frequencies than smoke or dust would. To help with this, and also to speed up rendering in general, an adaptive raymarching algorithm was developed. A simple adaptive raymarcher is trivial to write, but once it had to account for holdouts, internal motion blur, camera motion blur and other factors important to production rendering, the code turned out to be much more involved. We found that using the contribution to the final pixel value as an error metric automatically incorporated all these factors, and this sped up rendering considerably.

Levelsets were generated for each simulation buffer as a pre-pass and gave the raymarcher a tight-fitting bound on what parts of each buffer actually contained relevant information. The actual shader values were evaluated at each voxel and any non-zero values were marked as being inside the interface. The levelset was then extended by the velocity vectors to account for internal motion blur. This simple optimization turned out to be the biggest time saver, cutting 30 to 90 percent off rendering times.

For camera motion blur, we found that jittering the sample time at each raymarch step drastically reduced noise compared to shooting multiple rays for each pixel given equivalent render times. In effect, this bent the path taken by the ray marcher, so extra care had to be taken to ensure it worked with the adaptive march algorithm while still remaining strictly deterministic.