Guidelines for Project in EDAN35 High Performance Computer Graphics

Erik Nossborn
Lund University
Sweden

December 21, 2014

Abstract

An implementation for rendering metaballs using ray casting was created, with the end goal of animating a water flow.

1 Introduction

Metaballs can describe a smooth, liquid-looking volume in a simple way, while also being easy to manipulate by moving the metaballs, density limit, or changing the density function. Ray casting was chosen for rendering the surface, as it was deemed more likely to be able to produce smooth surfaces, compared to an algorithm like marching cubes.

The density function chosen was $D = (1-r^2)^2$ for $r < 1$, 0 for $r >= 1$, where r is the distance to the metaball center. The global density function is then $D_T = \sum D$ over all metaballs, and the surface is described by all points where $D_T = L$ for some chosen L. Points where the density is greater than L are inside the volume. Both D and its derivative (including the gradient for the multidimensional case) are continuous, which means the global density function will be too. In effect, this means there will never be any discontinuities in either surface or normal.[1] Both function and derivative also have finite support, and because of this, not all balls need to be taken into account at every point, allowing for balls to be removed in single instances, or discounted altogether for the remainder of the pixel calculation.

2 Algorithm

First, the world space transforms for the camera space base vectors are extracted from the inverse view matrix,[2] and sent to the shader.

[1]Since the gradient of D_T is an inwards facing normal to the volume described by $D_T = L$

[2]In reality from the view model matrix, since the program sees all rotations as model rotations.

A list of metaball locations are collected from a particle system generator, after which they are sorted in distance to camera order. The axis aligned bounding box is also computed. These values are sent to the shader.

A screen covering quad is created and drawn, after which the shader executes. The vertex shader does nothing interesting.

In the fragment shader, the ray vector is calculated by using the pixel position, window dimensions, and camera space base vectors (as described in world space).

Starting at the distance where the closest ball might have an influence for some pixels, steps are taken in the direction of the ray until a surface is hit, or it is deemed the ray will never hit anything.

For every iteration, the density function for the current point is calculated. Several culling methods are applied to ease the effort.

- Checks are made to the bounding box of all metaballs

  - Checks are made to see if the ray will ever intersect the bounding box.[3] If not, no surfaces will ever be intersected

- Check to see if the next ball to be evaluated is too far away from the camera to affect the result. If so, none of the later balls will either, since they are sorted.

- Check to ball's local bounding box.

  - Check to the perpendicular plane of the current ray and position. If the whole ball is behind, it will never be relevant again for this pixel, and the loop will skip it in subsequent iterations.[4]

When a surface is found, the normal is calculated from the gradient. The color is then calculated. Any coloring algorithm that takes position and normal could be used. In this case, a water-like shader is used with the help of a skybox texture.

Any pixel that does not find an intersection with a surface gets a color from the skybox.

3 Results

The performance is quite bad, and scales badly with both window size and number of balls. It is also heavily dependant on the slowest pixel to calculate, since the time each pixel takes to compute varies greatly.

Most artefacts only become apparent when several balls are stacked very close, or when zoomed in. See Figure 1. They are created from the discrete steps the algorithm takes.

Because of the limited number of metaballs, the individual balls can also be seen, even when they partially combine. This can be seen in Figure 2.

[3] If it is not moving away from the plane that was checked against.

[4] However, because of the program structure, this can only be done if the ball is first in the list of balls to evaluate.

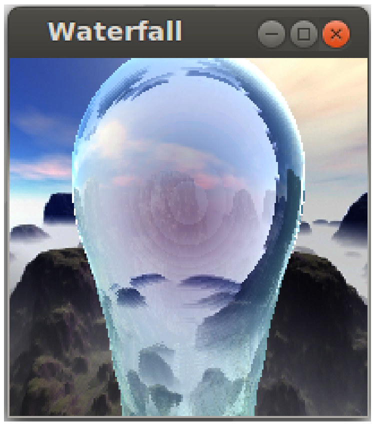

Figure 1: Visible artefacts in the form of rings. When several metaballs combine in the same place, the gradient gets steeper, and as such, the difference between discrete samples gets greater as well, causing imprecision.

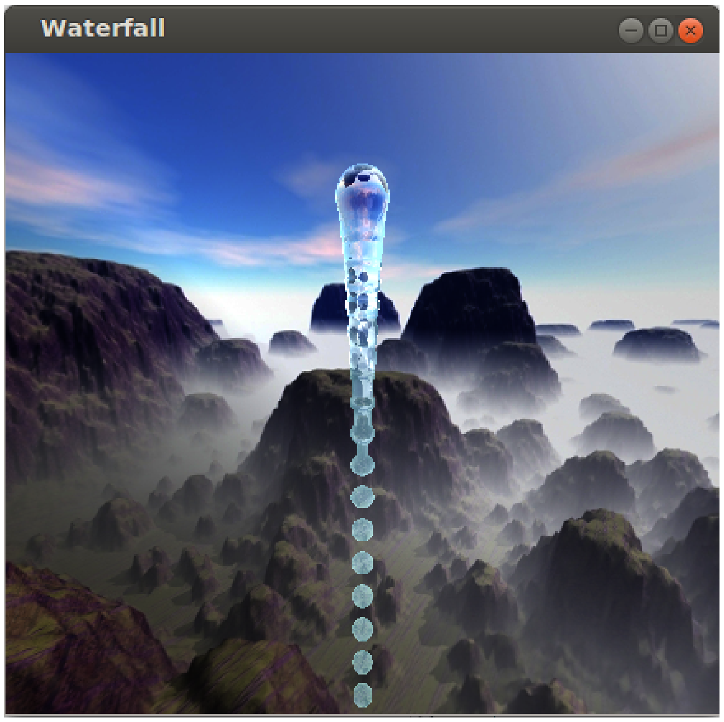

Figure 2: The simulated water flow. Acceleration in the particle system causes individual balls to split away. Individual balls can be seen even before then, however.

4 Discussion

Right now, the sideways culling is quite inefficient. An attempt was made to create a list of relevant balls for each pixel, before the stepping iteration. Relevant balls would be the ones where the pixel's ray penetrated the sphere of influence.[5] This, however, caused artefacts when balls where erroneously missed. Whether this was a bug in the implementation or caused by something else is unknown.

Having such a list, however, would be very useful, as the average number of balls evaluated would drop significantly.

Some culling could be performed even before sending the metaballs to the fragment shader, throwing away metaballs that will have no impact on any of the pixels.

Rendering the scene with polygon spheres covering the influence radius of the metaballs, and saving the result to a texture, could not only allow the pixel shader to quickly know if a pixel misses all metaballs, but would also give a tighter bound on the start value of the ray iteration, by reading the depth value of this intermediate texture.

Other improvements could include variable step length based on current density and distance to camera (perhaps combined with a concluding binary homing to lessen the error bound), since unnecessarily small steps are often taken.

Another possibility is to abandon the discrete steps and instead solve an approximated equation for the intersection, as done by Kanamori et. al.[6]

[5] Where the density function is non-zero.

[6] Yoshihiro Kanamori, Zoltan Szego and Tomoyuki Nishita, 2008: "GPU-based Fast Ray Casting for a Large Number of Metaballs" (`http://kanamori.cs.tsukuba.ac.jp/projects/metaball/eg08_metaballs.pdf`)