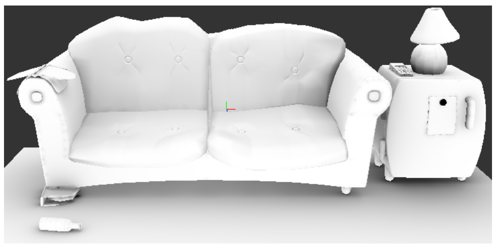
POINT-BASED AMBIENT OCCLUSION

TNCG14 ADVANCED COMPUTER GRAPHICS PROGRAMMING

Dan Englesson

Spring, 2011



Abstract

A two pass method for calculating point-based ambient occlusion was implemented on the GPU using GLSL and OpenGL, in the course TNCG14 - Advanced Computer Graphics at Linköping University. The program can load in wavefront-object files and render them with point-based ambient occlusion. The result is a good approximation of a ray-traced ambient occlusion render but it is rendered with a greater speed than with a ray-traced method. A point-based rendering visualization method called "Affinely projected point-sprites" was implemented and compared with the normal way of visualizing objects with triangles.

1 Introduction

This project was done in the course , TNCG14 - Advanced Computer Graphics at Linköping University during spring 2011. The topic of the project was free to choose so the chosen topic for this project became Point-based Ambient Occlusion because it is being more and more used in the movie industry to simulate global illumination and ambient occlusion in movies with a greater speed than with normal ray-tracing methods which due to the long rendering times is rarely used in movie-productions for global illumination.

2 Background and Related Works

Point-based global illumination and ambient occlusion is relatively new in the movie industry. In 2005 a prototype implementation of point-based ambient occlusion was implemented by Rene Limberger at Sony Imageworks for the movie Surf's Up and also a full point-based global illumination method was shortly thereafter implemented at ILM by Christophe Hery and used in the production of Pirates of the Caribbean: Dead Man's Chest [1].

In the book GPU Gems 2 there is a paper on Dynamic Ambient Occlusion and Indirect Lighting by Bunnell [2] on the GPU with CG shaders. It is an iterative approach and uses hierarchical trees for speeding up the calculations. The algorithm used in this project was heavily inspired by this paper but was instead implemented in OpenGL and GLSL shaders. Work of this sort has also been done on the CPU at Pixar [3] which approximates clusters of distant surfels with spherical harmonics and for medium-distant surfels a similar disk approximation method to Bunnell is used, and for the closest surfels ray-tracing is used.

Some work has also been put on visualizing the surfels and in this project Affinely projected point sprites [4] was implemented. It is a simple method and has some flaws when viewed at a extreme angles but yields good results. A related approach is EWA (Elliptical Weighted Average) surface splatting[5] which is by far the best method to use but it is a whole project on its own and was therefore not included in this project.

3 Implementation

The program was written in OpenGL for loading the Wavefront objects and storing the vertices as point-clouds in textures, and then sent to GLSL for calculations on the GPU. The method used in this project for simulating point-based ambient occlusion on the GPU is a two path method where in the first pass a point-cloud is generated and in the second pass the ambient occlusion is calculated. Two methods for visualizing the point-data was also compared, it was visualized with triangles or with affinely projected point-sprites. The comparison between these two methods for visualization was not the main focus of this project and will therefor be briefly discussed.

3.1 Pass 1: Generating the point-cloud

There are several ways of converting a mesh into a point-cloud, for example Turks point repulsion algorithm [6] which can be used to generate an even point-cloud on the surfaces, or to simply use the vertices' positions which is the method used in this project. The vertices were chosen to be the point-cloud since they were already there and their normals and area could easily be calculated, see equation (1) and equation (2).

$$\boldsymbol{N}_v = ||\sum_i^k \boldsymbol{N}_f(i)|| \tag{1}$$

$$\boldsymbol{A}_s = \frac{1}{3}\sum_i^p \frac{1}{2}|\boldsymbol{e1}_i \times \boldsymbol{e2}_i| \tag{2}$$

N_v is the vertex normal which is the normalized sum of the neighboring face normals $N_f(i)$ and A_s is the area of the surfel which is one third of the sum of the neighboring faces' areas which is calculated as half the magnitude of the unnormalized normal. Here $e1_i$ and $e2_i$ are two edges spanning up the i:th face.

Every vertex, or for now on surfel, will store the following:

Listing 1: Surfel structure

```
struct Surfel{
  vec3 position;
  vec3 normal;
  vec3 area;
  (vec3 color;//for GI only.)
};
```

All surfel positions are stored in a Texture Buffer, and their corresponding normal and area are stored into two separate Texture Buffers and sent to the fragment shader for ambient occlusion calculation.

3.2 Pass 2: Ambient occlusion calculation

Ambient occlusion gives darker areas at parts of the object that is partly visible to the surroundings. It calculates the percentage of the hemisphere at the point that is not occluded by nearby surfaces, see figure 1.

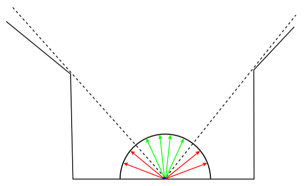


Figure 1: Ambient occlusion visualization. The vectors occluded are seen as red arrows, and the vectors not occluded are seen as green.

Instead of spawning rays at a given point p over the hemisphere as seen in figure 1, the point-based method projects all visible surfels $E_(i)$ that lies above the hemisphere on to the surfel R . Equation (3) describes the ambient occlusion calculation over almost the entire hemisphere except for surfels that lie near the horizon of the same plane which the receiver,R in figure 2, lies in.

$$o_s = \sum_i^k (1 - \frac{1}{\sqrt{\frac{A_{E,i}}{\pi d^2} + 1}})(\boldsymbol{N}_{E,i} \cdot \boldsymbol{r}_i)(4 * (\boldsymbol{N}_R \cdot \boldsymbol{r}_i)) \tag{3}$$

The calculated occlusion value o_s for a given surfel s is the sum of the all surfels that lies within the hemispheres' view weighted according to their distance squared d^2, the emitters area $A_{E,i}$, and the angles between the normal of the receiver and the direction r and the angle between the emitter and direction r, see figure 2 for visual comparison.

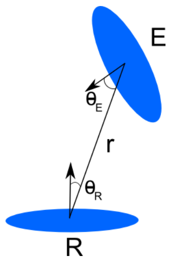


Figure 2: Ambient occlusion calculation between two surfels, R is the receiver and E the emitter that cast shadow onto R.

During this process the bent normal can easily be calculated by subtracting the directions of the vectors that are occluded by a nearby surface, so the bent normal will become the direction to where it is least occluded, see figure 3. The bent normal is very useful to simulating global illumination from surfels or environment maps, which is not included in this project but it is fairly easy to go from point-based ambient occlusion to point-based global illumination. To be able to simulate global illumination with point-clouds, the color of each point/surfel needs to be stored as well.

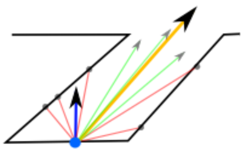


Figure 3: Bent normal, the least occluded direction. The blue arrow is the normal, and orange is the bent normal.

3.3 Over-occlusion compensation

The resulting ambient occlusion from equation (3) is sometimes too dark in some areas, because when calculating the ambient occlusion all surfels are taken into account even those that are occluded by other surfels much closer. Therefore surfels receives more surfels than they should and the result is over-darken areas, see figure 5(a) in section 4. There are several methods to reduce this problem, one method, which is an iterative approach, mentioned in [2], is to do the same calculations as before in equation (3) in a second pass but for each surfel multiply it with the last ambient occlusion result calculated for that specific surfel. After some iterations the result will look similar to a ray-traced ambient occlusion result. The more passes the better approximation. Another way is to scale the ambient occlusion value by a scale factor to make the ambient occlusion brighter. However by scaling the ambient occlusion, cracks and tight corners will also become much brighter when scaled as well, which is not desirable. Two methods that takes into account tight corners and cracks which was mentioned in [4] was to divide the hemisphere into n patches and each patch can at most occlude $1/n$. The second method, which was implemented, attenuates the occlusion from those surfels that are distant , in other words, a maximum distance value is set so distant surfels does not contribute. A result of this method can be seen in section 4, figure 6.

3.4 Visualization of the point-cloud

Since the point-cloud was generated by the vertices of an object, one method was to simply visualize the model with triangles and interpolate the ambient occlusion values stored at each vertex across the triangle-surfaces. This was the most straight forward way of visualizing it and therefore became the standard visualization in this project. An other method implemented for comparison was affinely projected point sprites [4] which is a splatting technique used to project the surfels onto the screen according to its normal and area. This was done by using OpenGL's point-sprites which gives view-spaced aligned squares with their centers at its vertex position p_i. A depth offset dz from the center p_i to a pixel (x,y) on the point-sprite, that is $r \cdot r$ in size, where r is the radius, can be calculated with the linear equation in equation (4), where $n = (n_x, n_y, n_z)^t$ is the vertex normal transformed to camera-space.

$$dz = -\frac{n_x}{n_z}x - \frac{n_y}{n_z}y \tag{4}$$

To be able to render the point-splats as ellipsoids one had to check if the pixel (x,y) lay within the given radius r by calculating the 3D distance to the center,$||(x, y, dz)|| \leq r$. If the pixel lay outside the radius it was killed with the comand *discard* in GLSL. The result was surfels that looked like they were following the curvature of the object according to the vertex normal, see figure 8 and figure 9 in section 4 for a visual comparison of the point-sprites and affinely projected point-sprites.

Since the size of each point-sprite in OpenGL is defined according to the pixel size on the screen, the surfels looked like they became larger when zooming out from the object. By taking the position of the camera into account the size of the surfels could stay true to the object instead of growing when zooming out. Equation (5) shows how to calculate the size of the surfel that does not vary when viewed from different distances.

$$s_{size} = R\frac{h_w}{||c - p||} \tag{5}$$

Where R is the radius of the surfel, h_w is the height of the window, c is the camera position and p is the surfel position. Basically the denominator is the z-depth value of the surfel.

4 Results and Performance

Here follow some results of the point-cloud renderer and performance benchmarks. Different results of over-occlusion are presented as well as a visual comparison between the point-sprites used in OpenGL, the affinely projected point-sprites rendering technique and the straightforward way of rendering triangles.

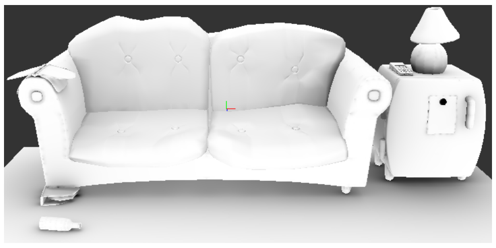


Figure 4: Rendered result of the point-based ambient occlusion method, rendered with triangles.

Figure 4 show a typical rendered frame from the program, rendered with triangles.

4.1 Over-occlusion compensation

In figure 5(a), on the next page, the ambient occlusion is somewhat over-occluded. By reducing the amount of occlusion by scaling it with a constant value smaller than 1, the ambient occlusion render gets brighter. It gives quick and good results at 40-60 percent of the original ambient occlusion value, see figure 5(c),(d). However it reduces the amount of occlusion everywhere not taking into account corners and cracks that should still stay dark. Compare especially the small pebble at the right foot in figure 5(c)(d) and figure 6 which is rendered by not taking into account distant surfels. One can see that the pebble in figure 6 has darker shadows beneath it than in figure 5. The shadows at the feet are more distinct in figure 6 than in figure 5 where they are more soft.

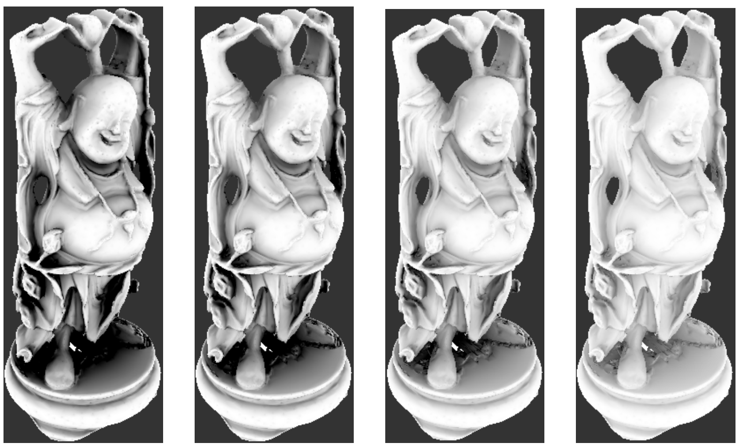

(a) (b) (c) (d)

Figure 5: Different amount of ambient occlusion applied to the surfels. In (a) the amount of ambient occlusion is the raw ambient occlusion value that is obtained from equation (3), (b) is 80 percent of (a), (c) is 60 percent of (a) and (d) is 40 percent of (a).

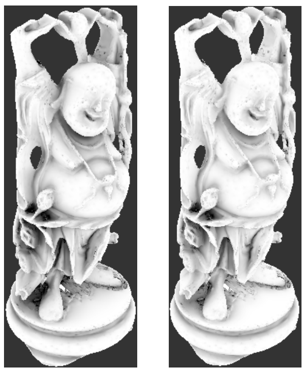

(a) (b)

Figure 6: over-occlusion reduction applied by having a distance threshold. (a) distance threshold(th), th = 1.99 and (b) th = 0.99

A combination of both can be suitable as well to lighten up dark areas such as cracks and corners that are too dark. Figure 11 shows the hybrid method in (a), and in (b) the distance attenuation method was only used. The distance was set to 2.0 in both images where in (a) the ambient occlusion value was scaled by 0.5 and in (b) the ambient occlusion value was not scaled. Note the dark areas in (b) is brighter in (a).

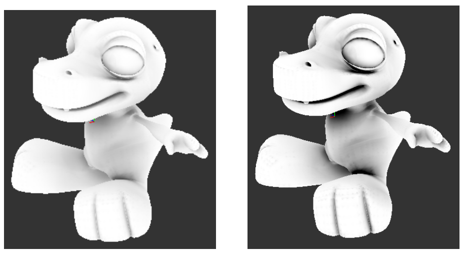

(a) (b)

Figure 7: (a) A hybrid of scaled occlusion and distance attenuation. In (b) only distance attenuation with the same distance as in (a).

4.2 Affinely projected point-sprites

By comparing the OpenGL point-sprites in figure 9 with the Affinely projected point-sprites in figure 8 one can clearly see the difference in these two images. The Affinely projected point-sprites visualizes the spheres much better than the ordinary point-sprite method in OpenGL, especially at the contours. However the images are taken from the same angle, but the floor is missing in figure 8. This is due to the fact that equation (4) is a parallel projection, which does not take into account the angle between the splat normal and the view

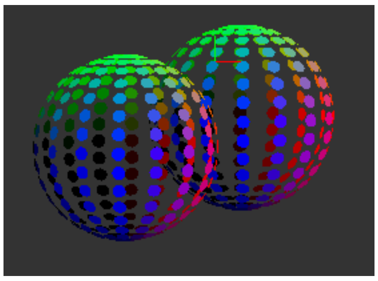


Figure 8: Affinely projected point-sprites, note the roundness of the point-sprites and how they follow the spheres curvature.

ray and therefore the splats becomes too thin and will not be rendered when viewed at extreme angles.

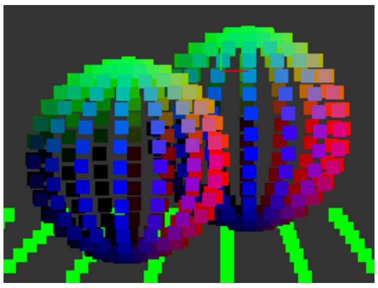


Figure 9: OpenGL point-sprites, visualizing two spheres and a ground-plane

In figure 10 the Rex model is visualized with the affinly projected point-sprites method. With some additionally Phong- or Gouraud-shading the model would look quite good, because now the sprites does not blend with each other so one can see distinct surfels.

In figure 11(a) the rex model is rendered with the triangle method, and in figure 11(b) the model is rendered with affinely projected point-sprites. However in figure 11(b) the size of the point-sprites had to be reduced in order to be able to render. It is due to the fact that the size of point-sprites can not be too large because the a larger point-sprite the more filling has to be done in the fragmentshader. A more complex model can be rendered with point-sprites if the radius of each point-sprite is reduced. A more detailed investigation of

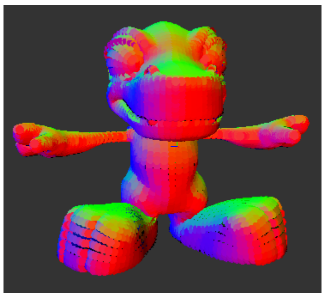


Figure 10: Affinely projected point-sprites, visualizing the REX model.

these two methods is performed in section 4.3, were benchmarks is made according to the radius, and number of vertices.

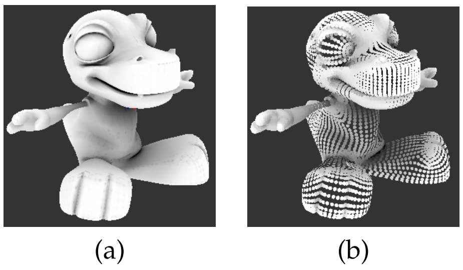


Figure 11: (a) Rendered with triangles (b) Rendered with affinely projected point-sprites

4.3 Performance

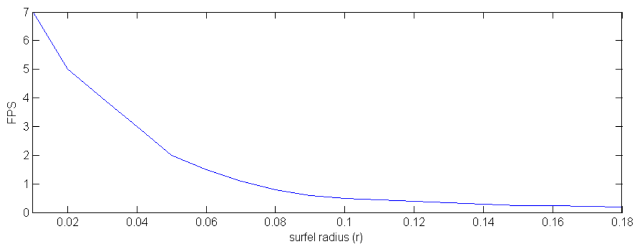


Figure 12: A graph over how the frames per second is affected only by the point-sprite radius

In figure 12 one can see how the radius affects the frames per second as the radius of the splats are increased. It shows that the frame rate drops dramatically just by increasing the radius by a small number. This is a huge bottleneck, since the splats needs to be quite big to fill the entire object. To reduce this problem one can have more points but a smaller radius. However by increasing the amount of vertices the frames per second drops by itself a lot, see Table 1. In Table 1 the radius stays the same but the amount of vertices are increased as it is rendered with ambient occlusion on.

Table 1: A comparison between the amount of vertices rendered with triangles and with affinely projected point-sprites

Vertices	Triangles (fps)	point-sprites (fps)
81	200	200
289	142	200
1089	36	26
4225	6	2
16641	0.48	0.1

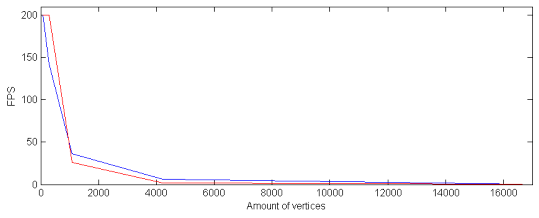


Figure 13: A graph over how the frames per second is affected by the amount of vertices in the scenen.

Figure 13 is a plot of Table 1 and with Table 1 combined it shows that the fps decreases drastically when increasing the amount of vertices when ambient occlusion is turned on. It also show that rendering it with triangles gives a little more fps, but they do follow each other closely. However when rendering with the point-sprites the radius needs to be bigger to cover all the area of the object. This will reduce the fps even more when point-sprites are used. A comparison between rendering with triangles and rendering with affinely projected point-sprites that has a radius that cover the entire surface is shown in Table 2 for different amount of vertices.

Table 2: A comparison between the surfel radius and the amount of vertices rendered with triangles and with affinely projected point-sprites

Vertices	Triangles (fps)	point-sprites (fps)	radius
81	200	30	0.46
289	142	7.3	0.24
1089	36	1.7	0.12
4225	6	0.37	0.06
16641	0.48	0.1	0.03

Table 2 clearly states that the affinely projected point-sprites can not be compared to the triangle method since the difference in frames per second has a mean difference value of about 69 frames per second, which is quite a lot.

4.4 Comparison between my renderer Iris, and Blender 3D's renderer

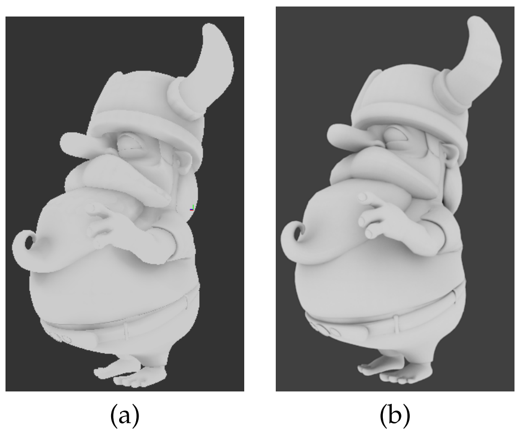


Figure 14: (a) Rendered with Iris, 11s (b) Blender internal with ray-traced ambient occlusion, 20s

The open-source program Blender 3D has also implemented point-based ambient occlusion and has as well ray-traced ambient occlusion. The test was performed on a ASUS G53S with a four core Intel i5 processor and Geforce GTX 460M as graphic-card. In figure 14(a) the viking model was rendered with Iris, and (b) was rendered with Blenders' ray-traced internal renderer with 16 samples. The viking in (a) took 11 seconds to render and (b) took 20 seconds to render without any anti-aliasing for only ambient-occlusion comparison. Blenders internal renderer works only on CPU, but it is threaded. Figure 14(a) approximates the ray-tracing method quite well, but can be tweaked even further to resemble (b).

Table 3: A comparison between the my render Iris, and blender's internal renderer that both has ray-traced and point-based ambient occlusion.

Iris point-based	Blender ray-traced	Blender point-based 1pass/2pass
11s	20s	10s/40s

Table 3 shows the time it took to render the viking with ambient occlusion in Iris and Blender internal with ray-tracing and with point-based ambient occlusion. One sees that the blender renderer with one pass of point-based ambient occlusion is faster than the one rendered with Iris and is still computed on the CPU and not with the GPU. Probably or most certainly an octree data structure is implemented in Blender to reduce the calculations from n^2 to $nlog(n)$ which gives a huge difference in computational time, see figure 15.

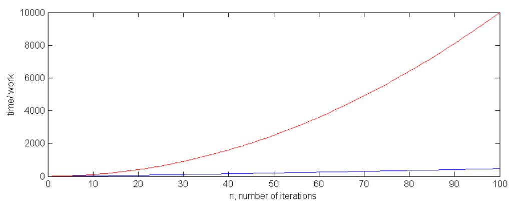


Figure 15: A graph over the computational time it takes for a n^2, seen in red, and a $nlog(n)$ system, seen in blue.

5 Conclusion and Future work

The point-based ambient occlusion approach approximates a ray-traced method quite well and at a greater speed than a ray-traced method, see figure 14 and Table 3. Although the rendering time is compared with a threaded ray-traced renderer on the CPU with this rendering method on the GPU, conclusions can still be drawn that the this method still renders faster than the threaded ray-traced method on CPU with an octree data structure, which means that it would outperform the ray-tracing method if an octree data structure was implemented. Creating an octree data structure for all surfels would be the most significant speedup which will reduce the calculations from n^2 to $nlog(n)$ and therefore the most important part to implement next in the future. An other performance boost would be to implement Vertex-buffer objects (VBO) so it is directly uploaded to the VRAM. Since the size of the surfels when rendering with affinely projected point-sprites is a big bottleneck when it comes to performance, see figure 12, the best idea would be to increase the amount of surfels an reduce the radius to cover the entire object, the ratio in fps between the triangle visualization and the point-based is reduced when the radius gets smaller and the amount of vertices increases, see Table 2.

Two more over-occlusion reduction methods would also be good to implement, the iterative shadow reduction algorithm and the patch method to complete the list of over-occlusion reduction methods. They seems to work a bit better than the ones implemented, but takes longer time to calculate.

Lastly one would implement global illumination but one then have to bake/store a point-cloud with direct illuminations, and therefore a color component needs to be added to the surfel structure. The direct illumination method needs to be written as well. It is quite a little step to go from point-based ambient occlusion to point-based global illumination but one has to stop somewhere and therefore it was decided to not implement it for now.

References

[1] Per H. Christensen. Point-based global illumination for movie production. Siggraph 2010 course: Global illumination across industries, Pixar Animation Studios.

[2] Michael Bunnell. Dynamic ambient occlusion and indirect lighting. *GPU Gems 2, Chapter 14*, 2005.

[3] Per H. Christensen. Point-based approximate color bleeding. Pixar technical memo, Pixar Animation Studios.

[4] Markus Gross. *Point-based Graphics*. Morgan Kauffmann publishers, 2007.

[5] M.Paulin G. Guennebaud. Efficient screen space approach for hardware accelerated surfel rendering. Technical report, CNRS-IRIT, Universit Paul Sabatier, Toulous, France.

[6] Greg Turk. Generating textures on arbritrary surfaces using reaction-diffusion. Technical report, University of North Carolina at Chapel Hill.