AN OPEN SOURCE FRAMEWORK FOR BROWNIAN MOTION SIMULATION IN A NEUROMUSCULAR JUNCTION (131 pp)

Director of Thesis: Arden Ruttan
# TABLE OF CONTENTS

PREFACE...............................................................................................................................ix

Chapter

1 Motivation and Background Biology.............................................................................1
2 Results.............................................................................................................................5
3 Theory and Methods
   3.1 Diffusion modeling.................................................................................................10
   3.2 Brownian Motion....................................................................................................11
   3.3 Monte Carlo..........................................................................................................12
   3.4 Monte Carlo algorithm for Brownian motion......................................................13
   3.5 Ray tracing.............................................................................................................16
4 Related work
   4.1 First-Passage Monte Carlo Algorithm: Diffusion without All the Hops...............19
5 Implementation
   5.1 Framework.............................................................................................................21
   5.2 Data File Format....................................................................................................22
   5.3 Generation of surface molecules...........................................................................27
   5.4 Precalculated Brownian Motion.........................................................................27
   5.6 Runtime Brownian Motion....................................................................................29
   5.7 Visualization of Results.......................................................................................30
6 The Neuromuscular Junction in Detail
   6.1 Tutorial Data File..................................................................................................32
   6.2 Reactions...............................................................................................................33
   6.3 Geometry and structures.......................................................................................35
7 Future Work..................................................................................................................42

BIBLIOGRAPHY..................................................................................................................44
Appendix 1: Source code used not by the thesis author.................................................46
Appendix 2: The author's source code............................................................................63
Acknowledgments

First and foremost, I would like to acknowledgment my advisor, Dr. Ruttan, for all his help and support throughout this thesis process as well as for what I have learned from him during his classes. I would also like to acknowledge my committee members, Dr. Volkert and Dr. Zhao, for their interest in this subject matter, the relevant knowledge they taught me, and their participation in my oral defense. I would like to acknowledge Dr. Stiles, Director of the Supercomputing Center of Pittsburgh and Dr. Burdzy, Professor of Mathematics at the University of Washington, for their generous willingness to help assisting a student outside their organizations. I would also like to acknowledge my parents, girlfriend Marianne Pender, my boss Todd Stoltz, and my employer John Clunk, for being supportive and understanding of my priorities during this thesis.
Chapter 1

Motivation and Background Biology

Every time a voluntary muscle contracts, motor neurons carry signals from the central nervous system telling it to do so. Acetylcholine, a neurotransmitter, is released from the neuron and crosses the space between the nerve cell and the muscle cell that is known as the neuromuscular junction, motor endplate or synaptic cleft [16]. On the other side of the cleft, acetylcholine binds to effectors[13]. Each effector has two sites that bind acetylcholine, and once bound, opens a channel releasing calcium ions into the muscle [5,13] (See section 6.2). This creates electrical current proportional to the number of channels open [13].

While any number of muscle fibers in a muscle can contract, individual muscle fibers work on the all-or-none principle, and either contract as strongly as they possibly can, or do not contract at all[16]. This effect happens from positive feedback: calcium ions are positively charged, and their release furthers the current imbalance, releasing more calcium ions. Any current, no matter how weak or strong, sufficient to release calcium ions starts the positive feedback loop. Below a certain threshold, endplate current is not strong enough to start the loop. Such an event is called “miniature endplate current”.

Acetylcholine molecules are released in groups, called quanta or quanta packets [17]. One quanta is not enough to contract the muscle and only results in a “miniature endplate
current”. Miniature endplate current, and release of a single quantal pack, since having fewer particles and release events, provides a better opportunity for a statistical analysis than a full release with an unknown number and timing of quantal packets [17].

The molecular mechanism for this process was not understood until recently, and is still an active area of research [5, 6, 17]. The neuromuscular junction is too small for observation under laboratory equipment [5, 17] and measurements of the speed the reaction takes are subject to bias based on the method of measurement [5].

Analytic methods were used in [15] to provide evidence that passive diffusion of acetylcholine is not fast enough to cause the measured miniature current endplate rise times, and an unknown active mechanism must be responsible for transporting acetylcholine across the cleft. [6] presents new methods of measuring miniature endplate current rise times, and a three dimensional computer simulation called m-cell that uses passive diffusion to show it is theoretically fast enough to produce the newly measured rise times. While the researchers state their simulation does not prove the actual mechanism used is the same as the one in the simulation, they claim it shows passive diffusion could be solely responsible for the reaction.

The methods and algorithms of simulation [6] were described in detail [5, 6, 13] and binary code is publicly available, but neither the source code nor the original input data was provided [personal communication with Dr. Stiles]. No open source implementations of the algorithms described are known to the author. The goal of this project is to reproduce the algorithms described in [6, 13], to provide a foundation for
further research, and to provide validation for Dr. Stiles's claims about how the neuromuscular junction works.

Reproducing a simulation already done by someone else has several advantages. First, it provides additional scientific validation. A bug in simulation software that from a computer scientist's perspective is trivial could give results that to a biologist are dramatically different. Dr Stiles makes the claim that passive diffusion of acetylcholine could be responsible for generating electric current in the neuromuscular junction within the time observed by his measurements. How can a good scientist accept this as proven without recreating this simulation, or at least reviewing the source code?

Another advantage of reproducing a simulation is the ability to compare run time and visualization of results directly against the simulation running in m-cell[5]. Writing a new simulation and loading into m-cell as well as the new framework would have accomplished the same comparison.

Reproducing the m-cell simulation requires simulation geometry. Simulation geometry from two different sources was used in the new framework. Geometry and reaction data for a similar, more detailed and up to date simulation of the neuromuscular junction that uses m-cell is available in a tutorial from “Computational Neuroscience: Realistic Modeling for Experimentalists” [5]. This version is far more complex, consisting of over fifty five thousand triangles for the geometry and over three hundred release events and uses a rat's muscle instead of a lizard's. This geometry was used as the basis for a data file for the new simulation framework.
A new simulation, based on the original details of [6] and [13] was also created to better reproduce the simulation given in [6] for use with the new framework. The goal of this new simulation was to include detailed, realistic geometry, and at the same time reproduce the measurements given in [6,13] for the simulation [6]. At thirteen hundred triangles, this is much smaller than the m-cell tutorial of a rat neuromuscular junction. The smaller size results in faster performance, more manageable data files, and easier debugging. A 2D outline was drawn around a 2D electron microscope image from [13] and scaled to a primary cleft height of fifty nanometers, similar to the geometry used in [13].
Chapter 2

Results

Laboratory results by [6] measure the time from the start of the experiment until twenty percent to eighty percent of the miniature endplate current peak. Simulation results in [6] are reported in the same way, to match the laboratory results.

Since endplate current generation was the primary output measured in the original m-cell simulation[6], endplate current was chosen as a means to compare various neuromuscular junction simulations. Running the publicly available neuromuscular junction simulation[5] in both the m-cell framework and the new framework should produce similar results if both frameworks are valid. Results from the author's newly created simulation are also measured in the new framework. Results from running either simulation in the new framework that matches the original m-cell results [6] provide support for Dr. Stiles's conclusions about how a neuromuscular junction works.

Running the neuromuscular junction simulation provided by m-cell in the actual m-cell framework shows no current until fourteen microseconds, when the first channel opens. Peak measurement occurs at two hundred seventy five microseconds, with seven hundred fifty one channels open. Twenty percent of the channels, or one hundred fifty one channels were open at seventy microseconds. Eight percent of the peak value, or six hundred and one channels, occurs at one hundred fifty three microseconds. This puts this simulation within the range of the measurements reported in [6], and is sufficient to
support the claim that passive diffusion is capable of generating between twenty and eighty percent of a rise at eighty microseconds. Because this is a Monte Carlo Simulation (see section 3.3) results vary each run time.

Detailed statistics for how far a particle can travel in a time step where given in [13]. Calculations from the open source framework probability density function matched these values (see section 5.4), validating the rate of diffusion in the new framework is correct.

Collision detection was validated by a simulation that released particles inside a cube for several thousand time steps, varying diffusion constants and time steps. Particles never left the box, validating collision detection is working.
Image taken from running the simulation provided by m-cell in the new open source framework. Acetylcholine (green) barely reaches the surface with the effector sites after eighty microseconds. This is enough to cause more than twenty percent of the peak current, consistent with Dr. Stiles's results.

Comparing the timing of the current generation of the publicly available m-cell simulation of a rat[5] neuromuscular junction with the original m-cell simulation [6] of a lizard neuromuscular junction is only meaningful if the geometry of both neuromuscular junctions is similar enough to expect similar rise times. Both simulations vary in shape and detail of geometry presented. Neuromuscular junctions also vary in shape and size between species and different muscles and nerves[13]. The rat simulation contained a primary cleft height that varies, but is approximately fifty nanometers [5]. Measurements
for the primary cleft height in [6] were not given, but the lizard endplate was modeled similar to [13], which states it used a primary cleft height of fifty nanometers. Since the simulation is measuring the time acetylcholine takes to cross the primary cleft, different primary cleft heights would affect the results.

Diffusion coefficients, the standard way to measure the speed a chemical diffuses, vary widely between simulations. The original m-cell simulation [6] uses $6 \times 10^{-6}$ centimeters squared per second, and the publicly available simulation provided by m-cell [5] uses $2.1 \times 10^{-6}$ centimeters squared per second, and [15] uses $4 \times 10^{-6}$. $6 \times 10^{-6}$ is based on measured diffusion of acetylcholine in a laboratory moving freely through water or saline [6]. $2.1 \times 10^{-6}$ represents the reduced speed due to a slow down crossing the basal membrane (see section 5.3) [personal communication with Dr. Stiles]. Running simulation[5] with the original diffusion coefficient of $6 \times 10^{-6}$ in the m-cell framework creates a faster rise in current, but is still within the twenty to eighty percent reported by Dr. Stiles.

Unfortunately, time constraints prevented completion of reaction mechanisms within the new open source framework to the point endplate current could be measured. However the open source framework simulation shows acetylcholine diffusing at a rate similar to the m-cell framework when running the rat neuromuscular junction simulation[5] in both frameworks. Since the primary goal of this project was to recreate the algorithms for Brownian motion presented in [6, 13] and validating the conclusion presented about the function of the neuromuscular junction in [6] was a secondary goal,
this project was a success. The author plans to distribute the source code used freely on the Internet. See chapters 5 and 6 for details of the code and simulation.

Writing the simulations and the open source framework described in this thesis has greatly added to the author's understanding of the chemical reactions, physics and biological structures involved. Running and visualizing a neuromuscular junction provides more understanding of neurotransmission between muscle and nerves and in more detail than what can be learned from a textbook, which at best contains two-dimensional images, a few numbers and qualitative notes. Having an open source implementation of this algorithm also provides many opportunities for future work by the author. Eventually, a library of many different biological structures and processes along with widely available open source software could allow any student or researcher to view formal logic, quantitative results and three-dimensional geometry and interact with it.
3.1 Diffusion modeling

There are two commonly used ways of modeling diffusion [7, 13]. Various points, usually on an evenly spaced grid, can be chosen to sample the concentration of the particles of interest. The diffusion equation, given by Adolf Fick in 1855 [19], can be solved to calculate the concentration at later times based on a starting concentration at each point and a diffusion constant for each substance of interest. If boundaries that prevent particles from diffusing are included in the simulation, they are limited to the resolution of the sample points chosen.

The other way to model diffusion uses a Monte Carlo algorithm (see section 3.3) to track each individual particle and move it around by Brownian Motion (see section 3.2). In this approach, particles have arbitrary coordinates, and are not bound to a grid. Any representation of a surface where collision detection can be calculated can be used, and accuracy is limited by floating point error, the length of the time step and the number of times the simulation is run.

For many problems, the former approach is just as accurate and much faster and easier to compute. However if the geometry contains structures that are small relative to the density of particles in an area, or if the particles are sparse, this approach works
poorly [7, 13] – too many grid points are needed, and most of them are empty or have only one particle, which severely limits accuracy and efficiency. The Brownian motion simulation is not the best solution to all diffusion problem either. Performance scales linearly with the number of molecules in a Brownian motion simulation but is independent of the number of molecules in a grid based approach [7]. With a large number of particles in a relatively open space, or a large area in general, modeling each particle is computationally wasteful. Both approaches are relevant to different biological simulations on different scales. Using both approaches in a simulation that covers processes that happen on different scales has also been done [2].

3.2 Brownian Motion

In 1827 a botanist named Robert Brown noticed small particles place in a fluid moved about seemingly at random. He documented and published this observation, but never was able to find the cause of this movement. Albert Einstein [9] later explained this movement to be the effect of kinetic energy of the molecules that make up the fluid on particles or molecules within the fluid. Prior to Einstein's work, it was assumed that Brownian motion and diffusion were two separate processes. Today the term “Brownian motion” is not only used to describe the physical movement of small particles, but also applied to the mathematics of such motion, and even to the motion of something that is not physical, such as the price of a stock.
While it is possible to model each individual molecule within a fluid, including its inertia, velocity, direction and effect as it collides with other molecules, it is rarely practical. With a mole of water weighing eighteen grams, at a density of 1 gram per milliliter, there are $3.34 \times 10^{14}$ water molecules in a cubic micron. Using three 64-bit floating point numbers to track each would require over eighty petabytes of memory. Unless we are interested in individual water molecules, there is no benefit from this level of complexity. Biological researchers are usually only interested in the behavior of particles that are many times larger and far less numerous than the molecules of the surrounding fluid. The molecules that make up the fluid are only relevant for their effect on the particles of interest. The net effect of an unknown number of molecules applying a varying level of force in different directions at different times on a larger particle can be approximated by Brownian motion by moving the particle a varying distance each time step in an arbitrary direction, independent of its previous velocity. Some approaches [8] do use inertia and velocity to model motion within a single time step, in the case, for instances, when the liquid under consideration is moving at a constant velocity. In such cases, Brownian motion is simulated by arbitrary, small changes in the constant velocity of the fluid.

3.3 Monte Carlo

The first mention of the term Monte Carlo as it is used here was in the paper “The Monte Carlo Method” [10]. The authors of this paper write “Problems involving only a
few particles were studied in classical mechanics, through the study of systems of ordinary differential equations. For the description of systems with very many particles, and entirely different technique was used, namely the method of statistical mechanics.”

The problem was in dealing with a number of items too large for the first approach to be practical but too small for the second approach to be accurate, so the paper introduced a new approach for this situation.

Instead of statistically evaluating actual events, the situation is evaluated theoretically multiple times, each time randomly choosing a course of action according to its probability distribution. With enough samples you can statistically evaluate the result accurately. The name “Monte Carlo” refers to a famous gambling location, and is used as an analogy to the process of gambling. Although published in 1949, this paper [10] claims “computing machines are extremely well suited” to this task.

3.4 Monte Carlo algorithm for Brownian motion

To move a particle a certain distance in a Monte Carlo simulation, we need a function to calculate the probability distribution for where the particle will end up in a certain time. The probability of the distance a particle moves by Brownian motion can be approximated by a Gaussian distribution [13]. The Gaussian curve is highest at zero so the probability of moving a smaller distance is higher. The standard deviation of the Gaussian distribution is the diffusion constant [9], which is given in area² * time. Movement in all directions is equally likely.
Usually, a one to one mapping is used; for every particle being simulating, one particle is tracked in the simulation. One could approximate a solution to a large number of diffusing particles using a smaller number of particles at a loss of accuracy, but a better solution for a larger number of particles is available (see section 3.1). M-Cell and related simulations all use a one to one mapping [Personal communication with Dr. Stiles]

It is not necessary to run the entire simulation multiple times for this to be considered a Monte Carlo simulation. Each diffusion particle makes many hops over the time of the simulation, and with enough hops, the time for the particles to diffuse usually will approach a similar value. However multiple runs can provide a more accurate assessment of possible outcomes.

Modeling mathematically correct Brownian motion is challenging in an environment without changes in density or surfaces and just moving a particle around in empty space. When elements are introduced into the simulation that the particle can interact with, such as other particles or boundaries, the simulation gets more complicated. Of course, at the atomic level, there are no discrete surfaces, but it is far more efficient to represent such surfaces as a continuous boundary than to represent each molecule making up the boundary. If a simulation calculates that a particle would travel a certain distance in a certain time and there is no boundary between the two points, it can move the particle assuming no surface interaction.
If a boundary is present, several different methods exist to calculate the direction of reflection. One typically models the force components of each particle separately for each axis, and move the particle independently along each axis, reflecting it along an axis if it hits a boundary. The open source framework uses the surface normal as the direction of reflection. This model of Brownian motion is called “normally reflected Brownian motion” corresponds to Neumann boundary conditions in the continuous model of diffusion. While other models exist, this model of Brownian motion is the most popular and also a reasonable approximation of how particles move when observed under a microscope. [8, Personal Communication with Dr. Burdzy].

A particle observed through one time step in the simulation takes a straight path, possibly with sharp corners from bouncing off surfaces. This is not how particles travel by Brownian motion when observed through a microscope. The probability of a particle moving in a perfectly straight line between two points is very low. Observing the same particle with the same start and end points over the same interval using several small time steps would show movement more consistent with observation.

Standard models of Brownian motion do not take into account the previous position, direction or momentum of a particle. The simulation only needs the probability distribution of where the particle ends up, the path the particle takes between positions is not important.
3.5 Ray tracing

An important feature of a Brownian motion simulations is a mechanism to detect the collision of a molecule with a surface. A graphical technique know as ray tracing implements collision detection similar to what is needed. Ray tracing is a drawing algorithm that calculates a series of rays starting from the eye or camera, and projects them into the scene. Each ray is collision tested against objects in the scene, and draws the object hit on the first collision. If a ray hits a reflective or refractive surface, one or more new rays can be cast from that point in other directions to realistically handle these effects. New rays can also be cast towards lights in the scene when hitting any surface, to test for shadows. A ray tracer can handle nearly any representation of a geometric primitive, as long as collisions can be calculated. A ray tracer which only track the initial rays from the viewer is called a ray caster. [18] The ray tracer used for collision detection can also be used to provide visualization of the simulation results.

The author's ray tracer employs several techniques to accelerate drawing and other collision detection. Objects composed of triangles have an axis aligned bounding box created around them. First, rays are checked against the bounding box, and if they hit, they are checked against each triangle in the set. Rays that do not hit the bounding box do not need further calculations, and are discarded, saving computation time. A common operation in a ray tracer is finding the nearest surface a ray hits. If another surface hit is known, the distance to that surface can be used in the bounding box calculation.
Sets of triangles can also be automatically subdivided. This accelerates collision detection, since each smaller set contains fewer triangles and smaller bounding boxes. The largest dimension of the axis aligned bounding box is picked as the direction to subdivide, and each triangle in the set is placed in two new subsets, based on the center of the triangle's location compared to the center of the axis aligned bounding box. This process can be done manually by specifying the subsets in the data file, but loading a large set of triangles and specifying the number of times to subdivide is easier for the user.

The ray tracer is multithreaded to take advantage of parallel processing. The image is divided vertically into groups of scan lines, and each group is assigned its own thread. Each pixel in the finished image consists of four rays for antialiasing. All rays and pixels processed on a single scanline are always processed within the same thread, eliminating most issues with sharing memory between threads. The height and width of the image produced by the raytracer is specified in the input file in both pixels to determine the resolution and degrees to determine the perspective.

Lighting uses the Blinn shading module [18]. Lighting is made up of three components, ambient, diffuse, and specular, which are calculated for each light in the scene, and all lighting values are added together to obtain the color of the surface the ray hits. Ambient light is evaluated as $C_a C_m$. Diffuse light is evaluated as $C_m C_p (N \cdot L)$. Specular light is evaluated as $C_p S (N \cdot H)^a$. Where $C_a$ is the color of the ambient light, $C_m$ is the color of the material, $C_p$ is the color of a point light, $N$ is the surface normal, $L$ is
the direction of the light, $S$ is a parameter that determines the specular strength of the material, $H$ is the halfway vector between the viewer and the light source, $a$ is a parameter to determine the shape of the specular highlights. A colors are specified by separate red, green and blue components.
Chapter 4
Related work

4.1 First-Passage Monte Carlo Algorithm: Diffusion without All the Hops

Paper [4] introduced another algorithm for the simulation of Brownian motion. Simulation space is partitioned into non-overlapping “protective domains” each containing one particle. Any geometric shape could be used, but spheres are picked for being computational simplicity. Instead of calculating the location of all particles at the next time step, one can calculate the exact time each particle would leave its protective domain. A heap is used to track the order the particles leave their protective domains. The first particle to leave its protective domain is updated both in space and time and a new protective domain is created around its new position so it does not overlap any of the other protective domains. The particle's time to leave the new protective domain is calculated and added to the heap.

If a particle is updated to a position where it can only get a very small protective domain, it will be more likely to leave the domain first and only travel a short distance. It will then only get a small protective domain again, and repeat the process. This is computationally expensive, and the simulation must detect when this happens. When this situation is detected, the program can update the neighboring particles based on how far they moved since they were last updated, and redraw their protective domains.
In some simulations, this algorithm results in several orders of magnitude speed up in Monte Carlo based Brownian motion simulation. However it does not apply well to every situation. It was designed for situations were sparsely distributed particles are likely to react with each other. In a biological simulation, we are typically modeling a substance that reacts with a receptor, not with itself. Protecting each particle from interfering with every other particle seems like a needless constraint. If thousands of particles in close proximity to each other are being simulated or if particles interact with complex geometry, protective domains could be small enough to make this algorithm computationally more expensive than modeling Brownian motion similar to m-cell. This algorithm may also work badly if particles interact with non moving elements of the simulation, since if a particle gets stuck with a small protective domain, the simulation cannot simply update the position if the particle's neighbors if the neighbors are not moving. Finally, visualizing the results would be difficult and awkward because the simulation only calculates the position of one particle at each time step, and the time steps are nonuniform between particles. For these reasons, this algorithm probably would not have worked well with the simulation chosen for this paper, but it was time constraints that prevented implementing this algorithm more than any other reason.
Chapter 5
Implementation

5.1 Framework

The framework to run the simulation is based on a ray tracer previously written by the author for another project and was described in Section 3.5. The collision detection to see what a ray hits and the collision detection on a particle moving by Brownian motion are the same. An m-cell like approach, where two programs were written, one to calculate the results, and another to visualize them would have been superior, but this was omitted due to lack of time. Visualizing results while running also assists in debugging. The program to calculate needs the ray tracer code base to do collision detection, so using it to do the display was trivial.

The framework is written in C#. Although C++ or C would have likely resulted in a faster program, C# is easy to read, less verbose, and easily portable between various Linux and Windows machines. M-cell runs noticeably faster than the open source framework, and the goal is to provide easily readable code to explain the algorithm, rather than to optimize run time. Open GL was deliberately not used for drawing, to increase portability of the code.

The ray tracer is multithreaded to take advantage of parallel architecture in modern machines, and makes use of simple bounding box checking around objects to speed up
computational time, as well as subdividing complex objects. Reflections, refractions and shadows were disabled for this project, making the drawing algorithm technically a ray caster. Anti aliasing was left on. The current model is made up entirely of triangles, but this is a limitation of the model and not the framework. The ray tracer includes code to handle spheres, and can be trivially extended to support any primitive as long as collision detection and surface normal calculations are implemented.

5.2 Data File Format

MDL is documented and human readable, so working directly with the format would have been possible. This was not chosen since several versions of MDL exist which are not entirely backwards compatible, documentation of mdl is minimal, mdl is not XML based, and aside from Blender (a 3d modeling program), MDL does not interface easily with widely used software.

The simulation file format in the author's simulation framework is an extension of the ray tracer program's file form, which was originally an XML format similar to VRML. Tags were added to file format to support needed concepts, such as effector density and particle release events. The ray tracer program did not use VRML directly, since VRML is a very large standard to support, VRML does not contain all needed representations to do a cell simulation and VRML is also regarded as obsolete.

The root element, *camera*, must be included once and only once in the data file. It determines the view of the raytracer. The attributes *cx* and *cy* determine the image
resolution in pixels. The attributes, width and height determine the angle of perspective projection in degrees. The element, origin, determines where the camera is located, and has three attributes, x, y and z. The element, direction, also has three attributes, x, y and z and determines the direction the center of the camera is facing. These parameters allow any perspective projection to be used, so long as the y axis is up. Right handed coordinates are used, with the z axis pointing towards the camera.

The root element, light, is optional, but at least one light must be included for practical purposes or the entire image will be black. The attribute type can specify either point or ambient. A point light has an element center which species the center of the light with x, y and z attributes. Shadows, included in the original ray tracer, are no longer supported since all simulations used by the author are closed geometry, and surfaces of interest would always be in shadow. A ambient light affects all surfaces equally. All lights must have an element intensity which has attributes r, g, and b to specify the color and brightness of the light. Zero intensity represents no light, and one represents a light capable of producing the brightest output values. Intensity values larger than one are supported, and pixels brighter than can be displayed are clipped.

The root element, material, specifies the appearance of a surface. The attribute, name, specifies the name of the material for use by surfaces. The attribute, opacity, defines how opaque a surface is. Values one or greater are completely opaque, and values zero or less specify a surface that is not drawn. The attribute, indexOfRefraction is used to specify the index of refraction for values between zero and one, but is of limited
use visualizing biological simulations. Similarly, the attribute *reflection*, specifies how much of a reflection the surface displays, is of limited use in biological simulations. The element, *color*, includes attributes *r*, *g*, and *b*, which range from zero to one and determine the brightness of diffuse lighting on the surface, specified as $C_m$ in section 3.5.

The element, *specular*, determines specular highlights. The attribute *power* ranges from zero to one to determine how much specular lighting is included, and corresponds to $S$ in the specular calculations in section 3.5. The attribute *shine* determines how tight the specular highlight is, and corresponds to the parameter $a$ in section 3.5.

Simulation geometry is specified with the root element *indexedFaceSet*. Each *indexedFaceSet* must have one and only one *coordinate* element. The *coordinate* element has one attribute, *point*. The *point* attribute is list of points used to make all the faces in the set. Each point is specified as three space delimited numbers representing $x$, $y$ and $z$ coordinates. Commas are used to delimit points. Each individual face can be represented as a *face* element, having the attribute *coordIndex* to specify coordinate indexes of the face as space delimited indexes into the *coordinate* element. Both an *indexedFaceSet* and a *face* can have a *material* attribute to specify the name of a material for the face. If both the *indexedFaceSet* and *face* element specify a *material* attribute the *face* element's attribute is used. Currently only triangles are supported, having a *coordIndex* attribute with three numbers. The element *triangles* can also be used to specify a large number of faces, having the attribute *coordIndex* to specify any number of triangles, delimiting each index into the *coordinate* element with spaces, and delimiting
each triangle with commas. The attribute *subdivisions* can be set to a number to automatically subdivided the *indexedFaceSet* for performance (see section 3.5).

The element *transform* of an *indexedFaceSet* can be used to move, scale or rotate the geometry. A *transform* element has two attributes, *matrix* and *order*. The attribute *matrix* specifies a space delimited set of numbers to defines a four by four transformation matrix. The attribute *order* specifies the order transformations are applied, if multiple transformations are specified.

The element *surface_molecules* of an *indexedFaceSet* specifies the surface contains non-moving surface molecules for reaction dynamics, and has three attributes, *boundingBox*, *species*, and *concentration*. The attribute *boundingBox* includes six numbers delimited with spaces, which specify an axis aligned bounding box by the min and max x axis values, min and max y axis values, and min and max z axis values. Surface molecules are applied before *transform* elements, so the bounding box is effectively transformed. The attribute *species* is the name of the species of molecule used. The attribute *concentration* is the concentration of surface molecules, in molecules per square micron. Any number of *surface_molecules* elements can be specified per surface.

The attribute *reflectiveToMolecules* of an *indexedFaceSet* specifies whether the surface reflects molecules or they can pass freely. A value for *reflectiveToMolecules* of zero allows molecules to pass freely, any other value, or omitting this attribute, makes the surface reflect molecules.
The root element *release* determines where and when molecules are related into the scene, and has four attributes, *delay*, *location*, *count* and *species*. The *delay* attribute is the time, in seconds, from the start of the simulation that molecules are released. The *location* attribute is three space delimited numbers specifying the coordinate for the release event. The *count* attribute determines how many molecules are released. The *species* attribute is the name of the species to release. The *release* element can include an element called *repeat* which has two attributes, *delay* and *count*. The *delay* attribute determines how long after the previous release more molecules are released. The *count* attribute determines how many times the event is repeated.

The root element *species* specifies the types of molecules in the simulation. Each *species* has three attributes, *diffusionCoefficient*, *name* and *radius*. The *name* attribute species the molecule name for later reference. The *diffusionCoefficient* specifies a diffusion coefficient in centimeters squared per second. A *diffusionCoefficient* attribute is ignored for surface molecules, but otherwise required. The *radius* specifies how far apart in microns a molecule must be from another molecule to react. When two molecules that react are near, both *radius* attributes are added to determine if they are within range.

The root element *reaction* determines reactions that can occur. It includes the attributes *species*, *rate* and *reversible*. The *species* element includes a comma delimited list of species' names needed for the reaction, usually only two. The *rate* is the reaction rate. If the *reversible* attribute is present and set to any value other than zero, it indicates
the reaction is reversible with the same reaction rate. If the reaction is reversible but at a different rate, the reverse reaction must be specified as a separate reaction element.

5.3 Generation of surface molecules

Effector sites used in the pathways are specified as being on a geometric object at a certain density. The area of each triangle in objects with surface molecules is calculated, showing how many surface molecules we are needed for the triangle. Fractional particles cannot be added, so fractional particles are treated as the probability of a whole particle, and this probability is compared to a random number. Once the number of particles on the triangle is known, each particle is placed randomly on a containing rectangle and rejection sampling is used to replace those that do not land on our triangle. This gives a uniformly random distribution of receptor sites.

5.4 Precalculated Brownian Motion

Once the model is loaded in, and the scene is first rendered without particles as a standard ray traced scene. The rendered scene stored for later use, with particles being drawn in later.

A set of $2^{16}$ distances are precalculated, using a Gaussian distribution to give an equal probability of each distance according to the formulas: $L_j = L_d \prod^{1/2} \text{erf}^{-1}(j/n - 0.5/n)$ and $L_d^2 = 4D(\Delta t/\prod)$ [13]. Where D is the diffusion constant, L is distance, n is the number of steps, j is the step number, and t is time.
Once all the distances are calculated, each is equally likely and we can simply index into the array at random to find how far a particle moves. This is the same approach used by m-cell, but more bins were chosen to improve accuracy. The original m-cell simulation [6] as well as the simulation in [13] only used one hundred bins, which in the author's opinion did not affect accuracy enough to alter the results, but adding more bins increases accuracy at very little memory and computational cost. In this simulation framework, each bin is 8 bytes using 512k of memory. $2^{32}$ bins would have been 32GB of memory with little gain in accuracy.

The source code to generate the bins can be validated by checking statistics given in [13]. Specifically, the median bin value is approximately 0.84 times the mean value, regardless of number of bins, diffusion constant or time step. For one hundred bins, the largest value is approximately 3.6 times the mean value, regardless of the time step or diffusion constant. A larger number of bins will result in a larger maximum value. Matching theses statistics for a wide variety of parameters gives reasonable certainty the code to generate the bins is correct, and therefore reasonable certainty that particles are diffusing at the correct rate, for any diffusion constant, and for any time step.

$2^{16}$ random directions where also calculated and binned to avoid calculating them when needed. A random number is picked for each axis, and the direction vector is then normalized.
<table>
<thead>
<tr>
<th>Rat neuro-muscular junction tutorial[5]</th>
<th>Rat tutorial with $\Delta t = \text{Dr. Stiles measured } 80%$ rise time</th>
<th>Origin al m-cell[6]</th>
<th>Original m-cell with more bins</th>
<th>Original m-cell with rat diffusion constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diffusion constant in cm²/second</td>
<td>2.1x10⁻⁶</td>
<td>2.1x10⁻⁶</td>
<td>6x10⁻⁶</td>
<td>6x10⁻⁶</td>
</tr>
<tr>
<td>$\Delta t$ in microseconds</td>
<td>0.1</td>
<td>100</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of bins</td>
<td>32767</td>
<td>32767</td>
<td>100</td>
<td>32767</td>
</tr>
<tr>
<td>Mean in microns</td>
<td>0.00517</td>
<td>0.164</td>
<td>0.0276</td>
<td>0.0276</td>
</tr>
<tr>
<td>Median in microns</td>
<td>0.00437</td>
<td>0.138</td>
<td>0.0236</td>
<td>0.0234</td>
</tr>
<tr>
<td>Expected median</td>
<td>0.00434</td>
<td>0.137</td>
<td>0.0232</td>
<td>0.0232</td>
</tr>
<tr>
<td>Maximum in microns</td>
<td>0.0280</td>
<td>0.886</td>
<td>0.0972</td>
<td>0.150</td>
</tr>
<tr>
<td>Expected maximum</td>
<td>-</td>
<td>-</td>
<td>0.0994</td>
<td>-</td>
</tr>
</tbody>
</table>

*Binned distance statistics from various simulation parameters validate the distance of each bin is being calculated correctly by the author's framework*

5.6 Runtime Brownian Motion

Several uniform time steps are taken. At each new time step, before applying Brownian motion, the simulation checks the release events loaded from the data file to see if new acetylcholine particles should be released. The Brownian motion algorithm is mutlitheaded, N threads are created, and each thread is allocated total number of particles / N to move on each time step.

Each particle in the scene is then given a randomly picked direction and distance, similar to m-cell. The ray tracer then checks to see if there is a surface between the old and the new locations. The new location is used if no surface is hit. If a surface is hit,
and no surface molecules are within range to interact with, the normal at the intersecting point is taken, and the remaining distance is projected from this point, again checking for and testing collision with other surfaces.

If a surface molecule is within a specified range, the probability of a reaction occurring is based on the contact supplied representing the reaction rate. A random number is checked against the probability to decide if a reaction occurs. If reactions occur, they are assumed to occur instantly. If a reaction does not occur, reflection happens as if no surface molecule were present. This approach models surface molecules as circles which are easy to calculate for intersections. Surface molecules were modeled as squares in [13] but no explanation for why was given.

5.7 Visualization of Results

After a certain number of frames (currently set to ten frames), the original ray traced buffer without particles is drawn, and receptors and diffusing particles are drawn over top. The simulation does not redraw every frame, since the cost of drawing is higher than the cost of calculating a time step. Even if the drawing code were optimal, the drawing operation would be significant. The simulation was not meant to run real time, but runs at about four frames per second on an Intel Q6600 providing an acceptable way of visualizing results. Frames can be saved to create a video file for later viewing at 30fps. Drawing particles for visualization could have used a depth buffer generated from the ray traced scene to hid particles that would have been behind surfaces. This was avoided, to
better see all particles in the simulation. To provide a sense of depth, particles are drawn brighter if they are closer to the camera, and dimmer if they are farther away. Use of hardware based rendering could have allowed for real time rotation and movement of the camera.
Chapter 6

The Neuromuscular Junction in Detail

6.1 Tutorial Data File

The geometric and chemical reaction data for the neuromuscular junction is taken from a tutorial on m-cell modeling [5]. The chemical reactions are the same for [5], [6], [13] and the author's own simulation. This tutorial [5] consists of nine MDL files, the format used by m-cell. To work with the model, the geometry was manually converted to VRML, imported into 3D Studio Max, edited, and exported back to VRML. Such editing did not change the geometry, but only changed how triangles were grouped into objects, since the ray tracer used in this framework uses bounding boxes around objects for speed improvements (see section 5.1), and Dr. Stiles grouping of geometry was not well suited to this. 3D Studio Max also provided an environment to view and analyze the geometry. A program was written to convert from the 3D Studio Max exported VRML to the XML based ray tracer format.

3D Studio Max was also used to create a model for the author's own simulation. The electron microscope image from [13] of a lizard's neuromuscular junction similar to that used in [6] and [13] was outlined, and the outline was extruded to create geometry. Measurements from [6] and [13] were used to scale the geometry to match the simulation in [6] in as much detail as was presented.
6.2 Reactions

Three sets of reactions are present in the updated neuromuscular model presented in [5]. The first is binding of acetylcholine to effectors. Each effector has two separate binding sites and can bind one acetylcholine molecule at each site. This reaction is reversible, and either or both sites can have acetylcholine unbound. If both sites are bound, the effector can undergo a conformational change to another state. The bound acetylcholine cannot unbind in this state, but the conformational change is reversible, making the entire process fully reversible. This reaction is modeled in [13, 14] as two separate sites, with no modeling of the time for the conformational change. Each effector, or pair of effectors, undergoing conformational change opens a channel. Each channel has a conductance of twenty five picosiemens[14], constant throughout the simulation.

The second set of reactions deals with acetylcholinesterase, the enzyme responsible for ending activity with the neuromuscular junction. The effector sites sit on the basal lamina, and once acetylcholine is bound, three different things can happen. First, the binding process is reversible, so acetylcholine can simply unbind, and leave everything in the starting state. Acetylcholine can also be hydrolyzed into choline and acetic acid. Since this is not a reversible process, and acetic acid is not used by the simulation except as a product of this reaction, this molecule is discarded from the simulation. Choline is released by this reaction and is modeled in the simulation, since choline participates in the third pathway. This reaction leaves the effector in the starting state. The third thing
that can happen is a second acetylcholine molecule can bind to the effector. In this state, either acetylcholine molecule can unbind or hydrolyze similar to when only one acetylcholine is bound.

The third set of reactions is choline reuptake. The effectors sit on the nerve membrane and can bind one choline molecule. The choline can unbind, or the effector can change to another state. The state change is not reversible. Once the state has changed, the choline eventually unbinds on the other side of the nerve membrane leaving the effector back in the starting state. The m-cell tutorial [5] appears to track the choline on the other side of the membrane, but since it cannot reenter the simulation, it should be discarded.
6.3 Geometry and structures

Figure 6.2.1: The neuromuscular junction from the publicly available m-cell tutorial loaded into 3D Studio Max

The geometry of the simulation is made up of three biological structures: the nerve membrane, the muscle membrane and the basal lamina. The nerve membrane [Figure 6.2.2] is a nearly flat sheet that is treated as one object by both simulations. It holds the choline reuptake receptors, and provides a boundary to prevent anything from diffusing out of the simulation in a upwards direction.
Figure 6.2.2: Nerve Membrane

The muscle membrane [Figure 6.2.3] is folded in a wave like shape that holds receptors that bind the Acetylcholine and trigger current within the muscle. This membrane is also a boundary, and prevents diffusion out of the bottom of our simulation and nearly prevents diffusion on two of the four sides. The receptors that bind acetylcholine are not uniformly distributed; they are more dense closer to the nerve membrane, and sparser farther away from it [6, 13]. A linear interpolation from the bottom of the muscle membrane to top would have been a good approximation of receptor density. The m-cell program does not provide a way to do this, so the model was broken into three objects; top, middle and bottom, each with a different uniform distribution of receptors. Since the muscle membrane is folded, it creates objects with
groups of triangles that are separated by empty space. This works inefficiently with a ray tracer optimized to use bounding box collision detection, so 3D Studio Max was used to break the objects up into groups of contiguous triangles.

Figure 6.2.3: Muscle Membrane, purple areas have the highest receptor count, dark green has the lowest.

The basal lamina [Figure 6.2.4] only needs to be present in the simulation since it holds the acetylcholinesterase effectors, it is completely transparent to diffusing molecules. It contains most of triangles in the model, and was treated as one object by m-cell.
The m-cell simulation also includes an object called 'cleft edges' [Figure 6.2.5]. This structure is not present in real biology [personal communication with Dr. Stiles], and is simply a way to prevent particles from diffusing out of the simulation. If conditions on the other side of this boundary are similar to conditions in the simulation, one could assume that for every molecule that would diffuse out of the simulation, another would likely diffuse into it. However there is typically no activity at all on the other side of the boundary [personal communication with Dr. Stiles], and particles should be allowed to diffuse across this boundary. Otherwise, the concentration of particles in the simulation is too high. This could be accomplished by modeling the muscle and nerve membranes.
with less detail over a larger area than the area with activity, since only a few particles are likely to diffuse this far away and are not likely to do anything relevant.

For this simulation, a modified version of the geometry was used to provide a rough approximation of the shape of an entire neuromuscular junction, based on pictures from [7].

In a neuromuscular junction, pores opens in the nerve membrane which allow stored acetylcholine to diffuse into the neuromuscular junction. This was simplified for the simulation presented in [5] as well as the author's simulation. A point near the center of the model was chosen, and acetylcholine is scripted to appear at certain times.

Figure 6.2.5: Original Cleft Edges
Figure 6.2.6: Sites of Release Events, only the center site is used
Figure 6.2.7: The author's simulation geometry of a neuromuscular junction. Reaction mechanisms work the same way as the rat tutorial.
Chapter 7

Future Work

7.1

Having an source code available for this algorithm and simulation opens the door to many new directions for research, both computational and biological.

The author originally wanted to test if such a simulation could be run by modeling geometry using metaballs instead of triangle primitives, and if it would run significantly faster. Metaballs have some properties that make them well suited to models such as this. We can model a complicated scene with far fewer primitives. We can test if a point is inside or outside our surface much faster. We can compute a more accurate normal than a triangulated surface at any point.

Metaballs also have some drawbacks. Calculating the exact edge of a surface is difficult. Calculating the area of a surface is next to impossible. Modeling with metaballs is not well enough supported by most modeling programs to produce geometry of this complexity, so writing a custom modeling program or 3D Studio Max plug in would likely have been required. Due to time constraints, a Metaball model of this simulation was not constructed.

Perhaps the biggest reason not to work on more efficient ways to run this simulation is that the computational time of this simulation on a single CPU machine is already acceptable. This came as a surprise to the author, as it would be reasonable to expect
computation complexity as a limiting factor, given the current literature and the devotion of supercomputing resources by Dr. Stiles. There are still many simulations that could be created and be useful that would be computationally faster to run. CPU speed is simply not the most significant limiting factor in what can be accomplish with Biological simulations like this one. As the author continues to work in this area, the author wants to spend time finding ways to make software that is useful to Biologists and work on ways to make technology do things it doesn't already do well. Speeding up the run time of such simulations is worthwhile, but seems less worthwhile by comparison.
Bibliography:


Appendix 1: Source code used not by the thesis author

/*************************************************************************
Cephes Math Library Release 2.8: June, 2000

Contributors:
  * Sergey Bochkanov (ALGLIB project). Translation from C to pseudocode.

See subroutines comments for additional copyrights.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.

- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
/*************************************************************************/
using System;

class normaldistr
{
    /***************************************************************************/
    // Error function
    // The integral is
    //
    // \[
    //   \frac{2}{\sqrt{\pi}} \int_{-\infty}^{x} e^{-t^2} dt
    // \]
    // For 0 <= |x| < 1, \( \text{erf}(x) = x * \frac{P4(x^2)}{Q5(x^2)} \); otherwise
    // \( \text{erf}(x) = 1 - \text{erfc}(x) \).
    /***************************************************************************/
    public static double erf(double x)
    {
        double result = 0;
        double xsq = 0;
        double s = 0;
        double p = 0;
        double q = 0;
        s = Math.Sign(x);
        x = Math.Abs(x);
        if (x<0.5 )
        {
            xsq = x*x;
            p = 0.007547728033418631287834;
            p = 0.288805137207594084924010+xsq*p;
            p = 14.3383842191748205576712+xsq*p;
            p = 38.0140318123903008244444+xsq*p;
            p = 3017.8278536507577809226+xsq*p;
            p = 7404.07142710151470082064+xsq*p;
            p = 80437.3630960840172832162+xsq*p;
        }
    }
}

ACCURACY:

<table>
<thead>
<tr>
<th>arithmetic</th>
<th>domain</th>
<th># trials</th>
<th>peak</th>
<th>rms</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE</td>
<td>0,1</td>
<td>30000</td>
<td>3.7e-16</td>
<td>1.0e-16</td>
</tr>
</tbody>
</table>

Cephes Math Library Release 2.8: June, 2000

***************************************************************************/
q = 0.0;
q = 1.00000000000000000000000+x*q;
q = 38.0190713951939403753468+x*q;
q = 658.070155459240506326937+x*q;
q = 6379.60017324428279487120+x*q;
q = 34216.5257924628539769006+x*q;
q = 80437.3630960840172826266+x*q;
result = s*1.1283791670955125738961589031*x*p/q;
    return result;
}
if ( x>=10 )
{
    result = s;
    return result;
}
result = s*(1-erfc(x));
    return result;
}

**************************************************************************
Complementary error function

1 - erf(x) =

\[
\int_{-\infty}^{x} \frac{2}{\sqrt{\pi}} e^{-t^2} dt
\]

For small x, erfc(x) = 1 - erf(x); otherwise rational approximations are computed.

ACCURACY:

Relative error:

+-----------------+------------+------------+-----------+--------+
| arithmetic      | domain     | # trials   | peak      | rms    |
| IEEE            | 0.26.6417  | 30000      | 5.7e-14   | 1.5e-14|

Cephes Math Library Release 2.8: June, 2000
**************************************************************************
double p = 0;
double q = 0;

if( x<0 )
{
    result = 2-erfc(-x);
    return result;
}
if( x<0.5 )
{
    result = 1.0-erf(x);
    return result;
}
if( x>=10 )
{
    result = 0;
    return result;
}
p = 0.0;
p = 0.5641877825507397413087057563+x*p;
p = 9.675807882987265400604202961+x*p;
p = 77.08161730368428609781633646+x*p;
p = 368.5196154710010637133875746+x*p;
p = 1143.262070703886173606073338+x*p;
p = 2320.439590251635247384768711+x*p;
p = 2898.029329216765611275846+x*p;
p = 1826.3348842295112592168999+x*p;
q = 1.0;
q = 17.14980943627607849376131193+x*q;
q = 137.1255960500622202878443578+x*q;
q = 661.361207107653469211984771+x*q;
q = 2094.38436778953953790281779+x*q;
q = 4429.612803883683682726711528526+x*q;
q = 6089.5424232724435504633068+x*q;
q = 4958.82756472114071495438422+x*q;
q = 1826.3348842295112595576438+x*q;
result = Math.Exp(-AP.Math.Sqr(x))*p/q;
return result;

*************************************************************************/

Normal distribution function

Returns the area under the Gaussian probability density function, integrated from minus infinity to x:

\[
\int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \, dx
\]
\[
\text{ndtr}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{t^2}{2}\right) \, dt
\]

\[
= \frac{1 + \text{erf}(z)}{2}
\]

\[
= \frac{\text{erfc}(z)}{2}
\]

where \( z = \frac{x}{\sqrt{2}} \). Computation is via the functions \( \text{erf} \) and \( \text{erfc} \).

**ACCURACY:**

**Relative error:**

<table>
<thead>
<tr>
<th>arithmetic</th>
<th>domain</th>
<th># trials</th>
<th>peak</th>
<th>rms</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE</td>
<td>-13,0</td>
<td>30000</td>
<td>3.4e-14</td>
<td>6.7e-15</td>
</tr>
</tbody>
</table>

Cephes Math Library Release 2.8: June, 2000
*************************************************************************/

```java
public static double normaldistribution(double x)
{
    double result = 0;

    result = 0.5*(erf(x/1.41421356237309504880)+1);
    return result;
}
```

/*************************************************************************/
Inverse of the error function

Cephes Math Library Release 2.8: June, 2000
*************************************************************************/

```java
public static double inverf(double e)
{
    double result = 0;

    result = invnormaldistribution(0.5*(e+1))/Math.Sqrt(2);
    return result;
}
```

/*************************************************************************/
Inverse of Normal distribution function

Returns the argument, \( x \), for which the area under the
Gaussian probability density function (integrated from
minus infinity to x) is equal to y.

For small arguments 0 < y < exp(-2), the program computes
\[ z = \sqrt{ -2.0 \times \log(y) } \]
then the approximation is
\[ x = z - \log(z)/z - (1/z) \frac{P(1/z)}{Q(1/z)}. \]
There are two rational functions P/Q, one for 0 < y < exp(-32)
and the other for y up to exp(-2). For larger arguments,
w = y - 0.5, and \[ x/\sqrt{2\pi} = w + w^3 \frac{R(w^2)}{S(w^2)}. \]

ACCURACY:

Relative error:

<table>
<thead>
<tr>
<th>arithmetic</th>
<th>domain</th>
<th># trials</th>
<th>peak</th>
<th>rms</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE</td>
<td>0.125, 1</td>
<td>20000</td>
<td>7.2e-16</td>
<td>1.3e-16</td>
</tr>
<tr>
<td>IEEE</td>
<td>3e-308, 0.135</td>
<td>50000</td>
<td>4.6e-16</td>
<td>9.8e-17</td>
</tr>
</tbody>
</table>

Cephes Math Library Release 2.8: June, 2000
*************************************************************************/

public static double invnormaldistribution(double y0)
{
    double result = 0;
    double expm2 = 0;
    double s2pi = 0;
    double x = 0;
    double y = 0;
    double z = 0;
    double y2 = 0;
    double x0 = 0;
    double x1 = 0;
    int code = 0;
    double p0 = 0;
    double q0 = 0;
    double p1 = 0;
    double q1 = 0;
    double p2 = 0;
    double q2 = 0;

    expm2 = 0.13533528323661269189;
s2pi = 2.50662827463100050242;
    if ( y0<=0 )
    {
        result = AP.Math.MaxRealNumber;
        return result;
    }
    if ( y0>=1 )
    {
        result = AP.Math.MaxRealNumber;
        return result;
    }
```c
}  
code = 1;  
y = y0;  
if (y>1.0-expm2 )  
{  
y = 1.0-y;  
  code = 0;  
}  
if (y>expm2 )  
{  
y = y-0.5;  
y2 = y*y;  
p0 = -59.9633501014107895267;  
p0 = 98.0010754185999661536+y2*p0;  
p0 = -56.6762857469070293439+y2*p0;  
p0 = 13.9312609387279679503+y2*p0;  
p0 = -1.23916583867381258016+y2*p0;  
q0 = 1;  
q0 = 1.95448858338141759834+y2*q0;  
q0 = 4.67627912898881538453+y2*q0;  
q0 = 86.3602421390890590575+y2*q0;  
q0 = -225.462687854119370527+y2*q0;  
q0 = 200.260212380060660359+y2*q0;  
q0 = -82.0372256168333339912+y2*q0;  
q0 = 15.9056225126211695515+y2*q0;  
q0 = -1.1833162112133003142+y2*q0;  
x = y*y2*p0/q0;  
x = x*s2pi;  
result = x;  
return result;  
}  
  
x = Math.Sqrt(-(2.0*Math.Log(y)));  
x0 = x*Math.Log(x)/x;  
z = 1.0/x;  
if (x<8.0 )  
{  
p1 = 4.05544892305962419923;  
p1 = 31.5251094599893866154+z*p1;  
p1 = 57.1628192246421288162+z*p1;  
p1 = 44.0805073893200834700+z*p1;  
p1 = 14.6849561928858024014+z*p1;  
p1 = 2.1866306850790267539+z*p1;  
p1 = -(1.40256079171354495875*0.1)+z*p1;  
p1 = -(3.5042462682784203418*0.01)+z*p1;  
p1 = -(8.57456785154685413611*0.0001)+z*p1;  
q1 = 1;  
q1 = 15.7799883256466749731+z*q1;  
q1 = 45.3907635128879210584+z*q1;  
q1 = 41.3172038254672030440+z*q1;  
q1 = 15.0425385692907503408+z*q1;  
```

q1 = 2.50464946208309415979 + z*q1;
q1 = -(1.42182922854787788574*0.1)+z*q1;
q1 = -(3.80806407691578277194*0.01)+z*q1;
q1 = -(9.33259480895457427372*0.0001)+z*q1;
x1 = z*p1/q1;
}
else
{
p2 = 3.23774891776946035970;
p2 = 6.91522889068984211695 + z*p2;
p2 = 3.93881025292474443415 + z*p2;
p2 = 1.33303460815807542389 + z*p2;
p2 = 2.0148538549179081538*0.1 + z*p2;
p2 = 1.23716634817820021358*0.01 + z*p2;
p2 = 3.0158153508235416007*0.0001 + z*p2;
p2 = 2.65806974686737550832*0.000001 + z*p2;
p2 = 6.23974539184983293730*0.000000001 + z*p2;
q2 = 1;
q2 = 6.02427039364742014255 + z*q2;
q2 = 3.67983563856160859403 + z*q2;
q2 = 1.37702099489081330721 + z*q2;
q2 = 2.1623699354496635890*0.1 + z*q2;
q2 = 1.34204006088543189037*0.01 + z*q2;
q2 = 3.28014464682127739104*0.0001 + z*q2;
q2 = 2.89247864745380683936*0.000001 + z*q2;
q2 = 6.7901940809981274425*0.000000001 + z*q2;
x1 = z*p2/q2;
}

x = x0-x1;
if ( code!=0 )
{
x = -x;
}
result = x;
return result;
}
}

namespace AP
{
/***************************************************************
Class defining a complex number with double precision.
**************************************************************/
public struct Complex
{
public double x;
public double y;
public Complex(double _x)
{
x = _x;
y = 0;
}
public Complex(double _x, double _y)
{
    x = _x;
y = _y;
}
public static implicit operator Complex(double _x)
{
    return new Complex(_x);
}
public static bool operator == (Complex lhs, Complex rhs)
{
    return (lhs.x == rhs.x) & (lhs.y == rhs.y);
}
public static bool operator != (Complex lhs, Complex rhs)
{
    return (lhs.x != rhs.x) | (lhs.y != rhs.y);
}
public static Complex operator +(Complex lhs)
{
    return lhs;
}
public static Complex operator -(Complex lhs)
{
    return new Complex(-lhs.x, -lhs.y);
}
public static Complex operator +(Complex lhs, Complex rhs)
{
    return new Complex(lhs.x + rhs.x, lhs.y + rhs.y);
}
public static Complex operator -(Complex lhs, Complex rhs)
{
    return new Complex(lhs.x - rhs.x, lhs.y - rhs.y);
}
public static Complex operator *(Complex lhs, Complex rhs)
{
    return new Complex(lhs.x * rhs.x - lhs.y * rhs.y, lhs.x * rhs.y + lhs.y * rhs.x);
}
public static Complex operator / (Complex lhs, Complex rhs)
{
    Complex result;
double e;
double f;
if (System.Math.Abs(rhs.y) < System.Math.Abs(rhs.x))
{
    e = rhs.y / rhs.x;
f = rhs.x + rhs.y * e;
result.x = (lhs.x + lhs.y * e) / f;
}
result.y = (lhs.y - lhs.x * e) / f;
}
else
{
  e = rhs.x / rhs.y;
  f = rhs.y + rhs.x * e;
  result.x = (lhs.y + lhs.x * e) / f;
  result.y = (-lhs.x + lhs.y * e) / f;
}
return result;

/*************************************************************/
AP math namespace
**************************************************************/

public class Math
{
  private static System.Random RndObject = new System.Random();

  public const double MachineEpsilon = 5E-16;
  public const double MaxRealNumber = 1E300;
  public const double MinRealNumber = 1E-300;

  public static double RandomReal()
  {
    return RndObject.NextDouble();
  }
  public static int RandomInteger(int N)
  {
    return RndObject.Next(N);
  }
  public static double Sqr(double X)
  {
    return X * X;
  }
  public static double AbsComplex(Complex z)
  {
    double w;
    double xabs;
    double yabs;
    double v;

    xabs = System.Math.Abs(z.x);
    yabs = System.Math.Abs(z.y);
    w = xabs > yabs ? xabs : yabs;
    v = xabs < yabs ? xabs : yabs;
    if (v == 0)
      return w;
    else
      w = xabs * yabs / (xabs + yabs);
      v = yabs / (xabs + yabs) * w;
      return v;
  }
}
```csharp
using System;

namespace Utilities
{
    public class StatUtil
    {
        private const double P_LOW = 0.02425D;
        private const double P_HIGH = 1.0D - P_LOW;

        // Coefficients in rational approximations.
        private static readonly double[] ICDF_A =
        { -3.969683028665376e+01, 2.209460984245205e+02, -2.759285104469687e+02, 1.383577518672690e+02, -3.066479806614716e+01, 2.506628277459239e+00,];
```

This code snippet includes methods for calculating the conjugate (`Conj`) and complex square root (`CSqr`) of a complex number, along with a class `StatUtil` that contains methods for statistical calculations such as the inverse normal cumulative distribution function and error function.
private static readonly double[] ICDF_B =
{ -5.447609879822406e+01, 1.615858368580409e+01,
  -1.556989798598866e+02, 6.680131188771972e+01,
  -1.328068155288572e+01 };  

private static readonly double[] ICDF_C =
{ -7.784894002430293e-03, -3.223964580411365e-01,
  -2.400758277161838e+00, -2.549732539343734e+00,
  4.37466414146968e+00, 2.938163982698783e+00 }; 

private static readonly double[] ICDF_D =
{ 7.784695709041462e-03, 3.224671290700398e-01,
  2.445134137142996e+00, 3.754408661907416e+00 }; 

/* public static double getInvCDF(double d, bool highPrecision)
{
    // Define break-points.
    // variable for result
    double z = 0;

    if(d == 0) z = Double.NEGATIVE_INFINITY;
    else if(d == 1) z = Double.POSITIVE_INFINITY;
    else if(Double.isNaN(d) || d < 0 || d > 1) z = Double.NaN;
    // Rational approximation for lower region:
    else if( d < P_LOW )
    {
        double q = Math.Sqrt(-2*Math.log(d));
        z = (((((ICDF_C[0]*q+ICDF_C[1])*q+ICDF_C[2])*q+ICDF_C[3])*q+ICDF_C[4])*q+ICDF_C[5]) /
            ((((ICDF_D[0]*q+ICDF_D[1])*q+ICDF_D[2])*q+ICDF_D[3])*q+1));
    }
    // Rational approximation for upper region:
    else if( P_HIGH < d )
    {
        double q = Math.Sqrt(-2*Math.log(1-d));
        z = -(((ICDF_C[0]*q+ICDF_C[1])*q+ICDF_C[2])*q+ICDF_C[3])*q+ICDF_C[4])*q+ICDF_C[5]) /
            ((((ICDF_D[0]*q+ICDF_D[1])*q+ICDF_D[2])*q+ICDF_D[3])*q+1));
    }
    // Rational approximation for central region:
    else
    {
        double q = d - 0.5D;
        double r = q * q;
            ((((ICDF_B[0]*r+ICDF_B[1])*r+ICDF_B[2])*r+ICDF_B[3])*r+ICDF_B[4])*r+1));
    }
    if(highPrecision) z = refine(z, d);
    return z;
private static readonly double[] ERF_A =
{ 3.1612374387056560E00, 1.13864154151050156E02,
3.7745237685302021E02, 3.20937758913846947E03,
1.85777706184603153E-1 }
;

private static readonly double[] ERF_B =
{ 2.36012909523441209E01, 2.440263793444173E02,
1.28261652607737228E03, 2.84423683343917062E03 }
;

private static readonly double[] ERF_C =
{ 5.64188496988670089E-1, 8.88314979438837594E0,
6.6119190637146295E01, 2.98635138197400131E02,
8.8195221241769090E02, 1.71204761263407058E03,
2.05107837782607147E03, 1.23033935479799725E03,
2.15311535474403846E-8 }
;

private static readonly double[] ERF_D =
{ 1.57449261107098347E01,3.60344899949804439E-1,
1.2578172611229246E-1,1.60837851847422766E-2,
6.58749161529837803E-4,1.63153871373020978E-2 }
;

private static readonly double[] ERF_P =
{ 3.05326634961232344E-1,3.60344899949804439E-1,
1.2578172611229246E-1,1.60837851847422766E-2,
6.58749161529837803E-4,1.63153871373020978E-2 }
;

private static readonly double[] ERF_Q =
{ 2.56852019228982242E00,1.87295284992346047E00,
5.27905102951428412E-1,6.05183413124413191E-2,
2.33520497626869185E-3 }
;

private static readonly double PI_SQR = Math.Sqrt(Math.PI);
private const double THRESHOLD = 0.46875D;

/* **************************************
* Hardware dependant constants were calculated
* on Dell "Dimension 4100":
* - Pentium III 800 MHz */
* running Microsoft Windows 2000
* ********************************************************* */
private const double X_MIN = Double.MinValue;
private const double X_INF = Double.MaxValue;
private const double X_NEG = -9.38241396824444;
private const double X_SMALL = 1.110223024625156663E-16;
private const double X_BIG = 9.194E0;
private static readonly double X_HUGE = 1.0D / (2.0D * Math.Sqrt(X_SMALL));
private static readonly double X_MAX = Math.Min(X_INF, (1 / (Math.Sqrt(Math.PI) * X_MIN)));

private static double calerf(double X, int jint)
{
    /* ORIGINAL FORTRAN version can be found at:
    * http://www.netlib.org/specfun/erf
    *********************************************/
    return X;
}
double result = 0;
double Y = Math.Abs(X);
double Y_SQ, X_NUM, X_DEN;

if(Y <= THRESHOLD)
{
    Y_SQ = 0.0D;
    if(Y > X_SMALL) Y_SQ = Y * Y;
    X_DEN = Y_SQ;
    for(int i=0;i<3;i++)
    {
        X_NUM = (X_NUM + ERF_A[i]) * Y_SQ;
        X_DEN = (X_DEN + ERF_B[i]) * Y_SQ;
    }
    result = X * (X_NUM + ERF_A[3]) / (X_DEN + ERF_B[3]);
    if(jint != 0) result = 1 - result;
    if(jint == 2) result = Math.Exp(Y_SQ) * result;
    return result;
}
else if(Y <= 4.0D)
{
    X_NUM = ERF_C[8] * Y;
    X_DEN = Y;
    for(int i=0;i<7;i++)
    {
        X_NUM = (X_NUM + ERF_C[i]) * Y;
        X_DEN = (X_DEN + ERF_D[i]) * Y;
    }
    result = (X_NUM + ERF_C[7]) / (X_DEN + ERF_D[7]);
    if(jint != 2)
    {
        Y_SQ = Math.Round(Y*16.0D)/16.0D;
        double del = (Y - Y_SQ) * (Y + Y_SQ);
    }
}
else
{
    result = 0.0D;
    if(Y >= X_BIG && (jint != 2 || Y >= X_MAX)){}
    else if(Y >= X_BIG && Y >= X_HUGE) result = PI_SQRT / Y;
else
{
    Y_SQ = 1.0D / (Y * Y);
    X_NUM = ERF_P[5] * Y_SQ;
    X_DEN = Y_SQ;
    for (int i=0;i<4; i++)
    {
        X_NUM = (X_NUM + ERF_P[i]) * Y_SQ;
        X_DEN = (X_DEN + ERF_Q[i]) * Y_SQ;
    }
    result = Y_SQ * (X_NUM + ERF_P[4]) / (X_DEN + ERF_Q[4]);
    result = (PI_SQRT - result) / Y;
    if(jint != 2)
    {
        Y_SQ = Math.Round(Y*16.0D)/16.0D;
        double del = (Y - Y_SQ) * (Y + Y_SQ);
    }
}

if(jint == 0)
{
    result = (0.5D - result) + 0.5D;
    if(X < 0) result = -result;
}
else if(jint == 1)
{
    if(X < 0) result = 2.0D - result;
}
else
{
    if(X < 0)
    {
        if(X < X_NEG) result = X_INF;
        else
        {
            Y_SQ = Math.Round(X*16.0D)/16.0D;
            double del = (X - Y_SQ) * (X + Y_SQ);
            Y = Math.Exp(Y_SQ * Y_SQ) * Math.Exp(del);
            result = (Y + Y) - result;
        }
    }
}
return result;

public static double erf(double d){ return calerf(d, 0); }
public static double erfc(double d){ return calerf(d, 1); }
public static double erfcx(double d){ return calerf(d, 2); }
/* ****************************************************
* Refining algorithm is based on Halley rational method
* for finding roots of equations as described at:
* http://www.math.uio.no/~jacklam/notes/invnorm/index.html
* by:
* Peter J. Acklam
* jacklam@math.uio.no
* ************************************************** */

public static double refine(double x, double d)
{
    if (d > 0 && d < 1)
    {
        double e = 0.5D * erfc(-x/Math.Sqrt(2.0D)) - d;
        double u = e * Math.Sqrt(2.0D*Math.PI) * Math.Exp((x*x)/2.0D);
        x = x - u/(1.0D + x*u/2.0D);
    }
    return x;
}
Appendix 2: The author's source code

using System;
using System.Threading;
using RayTracer;
using Utilities;

namespace Simulation
{
    public class Species
    {
        public string name;
    }

    public class Molecule
    {
        public Point3D p;
        public Species species;
        public int state;
    }

    public class Brownian
    {
        public Scene scene;
        public Random rand;

        private double lastTime;
        private double[] distances;
        private Point3D[] directions;

        public static double[] GaussianDistribution(double diffusionConstant, double deltaT, int count, double scale)
        {
            double Ld = scale * Math.Sqrt(4 * diffusionConstant * (deltaT / Math.PI));
            double x;
            double[] ary = new double[count];
for (int i = 0; i < count; i++)
{
    x = (i + 0.5) / count;
    ary[i] = Ld * Math.Sqrt(Math.PI) * normaldistri.inverf(x);
}
return ary;

public static Point3D[] RandomDirections(int count, Random rand)
{
    Point3D[] directions = new Point3D[count];
    Point3D direction;
    Point3D bias = new Point3D();
    for (int i = 0; i < count; i++)
    {
        direction = new Point3D(
            rand.NextDouble() - .5,
            rand.NextDouble() - .5,
            rand.NextDouble() - .5);
        direction.normalize();
        bias += direction;
        directions[i] = direction;
    }
    return directions;
}

public void Initialize(double timestep)
//TODO none of this should be hard coded, but instead, loaded into part of the scene
{
    double diffusion_constant = 2.1e-6;//includes slowing down for basal lamina
    //double diffusion_constant = 6e-6;//saltwater

    distances = GaussianDistribution(diffusion_constant, timestep, Int16.MaxValue, 10000);
    if (rand == null)
    {
        rand = new Random(187);
        directions = RandomDirections(Int16.MaxValue, rand);
        lastTime = 0;
    }
}

private TraceInfo TestCollide(Point3D origin, Point3D direction, double distance)
{
    TraceInfo info = new TraceInfo(),
    min = null;

    foreach (Primitive primitive in scene.Primitives)
    {
        info.primitive = primitive; //TODO find a better way to link these 2 structures
if (primitive.intersectRay(origin, direction, info, distance))
{
    if (min == null || info.distance < min.distance)
    {
        min = info;
        info = new TraceInfo();
    }
    if (min.distance < 0)
    {
        break;
    }
}
return min;

private class SimulateParams
{
    public int startI;
    public int endI;
    public double time;
}

public void SimulateProc(object simulateParams)
{
    SimulateParams s = (SimulateParams)simulateParams;
    int j, k;

    for (int i = s.startI; i < s.endI; i++)
    {
        Particle p = scene.Particles[i];
        lock (rand)
        {
            j = rand.Next(Int16.MaxValue);
            k = rand.Next(Int16.MaxValue);
        }
        Point3D direction = directions[j].Clone();
        double distance = distances[k];
        MoveOneParticle(p, direction, distance);
    }
}

private void MoveOneParticle(Particle p, Point3D direction, double distance)
{
    TraceInfo min;

    min = TestCollide(p.location, direction, distance);
    if (min == null || (min.distance > distance && min.distance > 0))
    {
        direction.scale(distance);
        p.location += direction;
    }
else{
    Point3D offset = direction.Clone();
    offset.normalize(min.distance);
    p.location += offset;
}

TriangleD tri = min.primitive as TriangleD;
if (tri != null & & tri.surfaceMolecules != null)
{
    foreach(Molecule m in tri.surfaceMolecules)
    {
        if (m.species.name == "ache" & & m.state >= 1)
            continue;
        if (m.species.name == "achr" & & m.state >= 2)
            continue;
        Point3D d = p.location.Clone();
        d -= m.p;
        if (d.magnitude() <= 0.05) //TODO random number
        {
            m.state++;
            p.toDelete = true;
        }
    }
}

if (tri != null & & !tri.reflectiveToMolecules)
{
    direction = Point3D.Reflect(min.normal, direction);
    direction.scale(-1); //TODO do we reverse or not?
}
MoveOneParticle(p, direction, distance - min.distance);

public void SimulateMultiThread(double time, int threads)
//TODO, this only works for particles that all have the same diffusion constant
{
    foreach (ReleaseSite r in scene.ReleaseSites)
    r.Simulate(time, scene.Particles);

    for (int i = 0; i < scene.Particles.Count; i++)
    if (scene.Particles[i].toDelete)
        scene.Particles.RemoveAt(i);

    int lastI = 0;
    for (int i = 1; i <= threads; i++)
    {
        SimulateParams s = new SimulateParams();
        s.startI = lastI;
        lastI = i;
    }
}
s.endI = i * scene.Particles.Count / threads;
s.time = time;
ThreadPool.QueueUserWorkItem(SimulateProc, s);
lastI = s.endI;
}
lastTime = time;
}
}

using Utilities;

namespace RayTracer
{
    public class Camera
    {
        public Camera()
        {
            origin = new Point3D(0,0,0);
direction = new Point3D(0,0,-1);
        }
        public int cx;//width in pixels
        public int cy;
        public double width;//width in degrees
        public double height;
        public Point3D origin;
        public Point3D direction;
        public bool WorldToView(ref Point2D viewPoint, Point3D worldPoint)
        //returns false if the point is outside or behind the view plane
        //return true otherwise, and sets view coordinates in viewPoint
        //this only works for the default camera position and direction
        {
            double scale = origin.z - worldPoint.z;
            if (scale <= 0)
            {
                return false; //behind view plane
            }
            viewPoint.x = cx / 2 + cx * (worldPoint.x - origin.x) / scale * 180 / width;
            viewPoint.y = cy / 2 + cy * (worldPoint.y - origin.y) / scale * 180 / height;
            viewPoint.y = cy - viewPoint.y;
            //TODO use view direction, and also optimize. Find out of a matrix is faster (I think it is not)
            return viewPoint.x >= 0 && viewPoint.x < cx && viewPoint.y >= 0 && viewPoint.y < cy;
        }
    }
}
using System;

namespace Utilities
{
    public class ColorD
    {
        public double r, g, b;

        public ColorD(double r, double g, double b)
        {
            this.r = r;
            this.g = g;
            this.b = b;
        }

        public static ColorD operator +(ColorD c1, ColorD c2)
        {
            return new ColorD(c1.r + c2.r, c1.g + c2.g, c1.b + c2.b);
        }

        public static ColorD operator -(ColorD c1, ColorD c2)
        {
            return new ColorD(c1.r - c2.r, c1.g - c2.g, c1.b - c2.b);
        }

        public static ColorD operator *(ColorD c1, ColorD c2)
        {
            return new ColorD(c1.r * c2.r, c1.g * c2.g, c1.b * c2.b);
        }

        public static ColorD operator /(ColorD c1, ColorD c2)
        {
            return new ColorD(c1.r / c2.r, c1.g / c2.g, c1.b / c2.b);
        }

        public static ColorD operator *(ColorD c1, double d)
        {
            return new ColorD(c1.r * d, c1.g * d, c1.b * d);
        }

        public static ColorD operator /(ColorD c1, double d)
        {
            return new ColorD(c1.r / d, c1.g / d, c1.b / d);
        }

        public static ColorD operator *(double d, ColorD c1)
        {
            return new ColorD(c1.r * d, c1.g * d, c1.b * d);
        }
    }
}
public static ColorD operator /(double d, ColorD c1) {
    return new ColorD(c1.r / d, c1.g / d, c1.b / d);
}

//borrowed from Point3D
public double magnitude() {
    return Math.Sqrt(r * r + g * g + b * b);
}

public double Brightness {
    get {
        return Math.Max(Math.Max(r, g), b);
    }
    set {
        if (value > 0) {
            double br = Brightness;
            if (br > 0)
                scale(value / br);
            else r = g = b = value; //arbitrarily set black to grey when brightened
        }
        else r = g = b = 0;
    }
}

public bool normalize(double scale) {
    double m = magnitude();
    if (m > 0.0) {
        r = scale * r / m;
        g = scale * g / m;
        b = scale * b / m;
        return true;
    }
    else return false;
}

public void scale(double scale) {
    r *= scale;
    g *= scale;
    b *= scale;
}
public ColorD Clone()
{
    return new ColorD(r, g, b);
}
}

using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Windows.Forms;
using System.Text;
using Utilities;
using RayTracer;
using Simulation;

namespace RayTracer
{
    public partial class Form1 : Form
    {
        private readonly RayTracer tracer;
        private Bitmap bmp;
        private byte[, ,] buffer;
        private DateTime last;
        private Scene scene;

        public Form1()
        {
            CheckForIllegalCrossThreadCalls = false; // needed for the way we handle painting on scanline complete
            InitializeComponent();

            tracer = new RayTracer();

            tracer.MinWeight = 0.1;
            tracer.MaxDepth = 16;
        }

        static byte colorFromDouble(double color, double min, double max)
        {
            if (max <= 0)
                return 0;
            if (color <= min)
                return 0;
            if (color >= max)
                return 255;
        }
    }
}
return (byte)(256.0 * (color - min) / (max - min));
}

static byte colorFromDouble(double color)
{
    if (color <= 0)
        return 0;
    else if (color >= 1)
        return 255;
    else return (byte)(255.0 * color);
}

void buildBuffer()
{
    int x, y;
    double max = double.MinValue,
            min = double.MaxValue;
    ColorD pixel;

    for (y = 0; y < tracer.scene.Camera.cy; y++)
        for (x = 0; x < tracer.scene.Camera.cx; x++)
        {
            pixel = tracer.Buffer[x, y];
            tracer.Buffer[x, y] = pixel;
            max = Math.Max(max, pixel.r);
            max = Math.Max(max, pixel.g);
            max = Math.Max(max, pixel.b);
            min = Math.Min(min, pixel.r);
            min = Math.Min(min, pixel.g);
            min = Math.Min(min, pixel.b);
        }

    if (min < 0)
        min = 0;

    for (y = 0; y < tracer.scene.Camera.cy; y++)
        for (x = 0; x < tracer.scene.Camera.cx; x++)
        {
            //need to flip vertically
            //in 2D y is down
            //in the raytracer, Y is up
            //I know this can be changed, but it is what I am used to
            pixel = tracer.Buffer[x, tracer.scene.Camera.cy - y - 1];
            buffer[y, x, 0] = colorFromDouble(pixel.r, min, max);
            buffer[y, x, 1] = colorFromDouble(pixel.g, min, max);
            buffer[y, x, 2] = colorFromDouble(pixel.b, min, max);
        }
}

void getBmpFromBuffer()
// updates the buffer as a System.Drawing.Bitmap
// there are faster ways to do this
// for this program it doesn't matter
{
    getBmpFromBuffer(0, tracer.scene.Camera.cy - 1);
}

void getBmpFromBuffer(int y1, int y2)
{
    if (tracer.scene == null)
        return;
    int cx = tracer.scene.Camera.cx,
        cy = tracer.scene.Camera.cy;
    Color color;

    lock (bmp)
    {
        for (int y = y1; y <= y2; y++)
        {
            Rectangle rect = new Rectangle(0, cy - y - 1, cx, 1);
            BitmapData bmpData = bmp.LockBits(
                rect,
                System.Drawing.Imaging.ImageLockMode.WriteOnly,
                bmp.PixelFormat);

            for (int x = 0; x < cx; x++)
            {
                color = Color.FromArgb(buffer[y, x, 0],
                    buffer[y, x, 1],
                    buffer[y, x, 2]);
                // need to flip vertically
                // bmp.SetPixel(x, cy - y - 1, color);
                rgbValues[3 * x] = color.R;
                rgbValues[3 * x + 1] = color.G;
                rgbValues[3 * x + 2] = color.B;
            }

            System.Runtime.InteropServices.Marshal.Copy(rgbValues, 0, bmpData.Scan0, cx * 3);
            bmp.UnlockBits(bmpData);
        }
    }
}

/* Point2D particleScreen = new Point2D(0,0);
    color = Color.FromArgb(0, 255, 0);
    foreach (Particle p in tracer.scene.Particles)
    {
        if (tracer.WorldToView(ref particleScreen, p.location))
            bmp.SetPixel((int)particleScreen.x, cy - (int)particleScreen.y - 1, color);
    }*/
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
    Close();
}

private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
    SaveFileDialog dlg = new SaveFileDialog();
    dlg.Filter = "jpg files(*.jpg;*.jpeg)|*.jpg;*.jpeg";
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        Stream stream = dlg.OpenFile();
        getBmpFromBuffer();
        lock (bmp)
        {
            bmp.Save(stream, ImageFormat.Jpeg);
        }
        stream.Close();
    }
}

private void OnScanlineComplete(int y)
{
    int x, y2 = tracer.scene.Camera.cy - y - 1;
    ColorD pixel;
    for (x = 0; x < tracer.scene.Camera.cx; x++)
    {
        //need to flip vertically
        //in 2D y is down
        //in the raytracer, Y is up
        //I know this can be changed, but it is what I am used to
        pixel = tracer.Buffer[x, y];
        buffer[y2, x, 0] = colorFromDouble(pixel.r);
        buffer[y2, x, 1] = colorFromDouble(pixel.g);
        buffer[y2, x, 2] = colorFromDouble(pixel.b);
    }
    //            if (new Random().Next(16) == 0)
    //update no more than once every second
    getBmpFromBuffer(y2, y2);
    if (DateTime.Now - last > new TimeSpan(0, 0, 2))
    {
        Refresh();
        last = DateTime.Now;
private void OnRenderComplete()
{
    Point2D viewPoint = new Point2D();

    buildBuffer();
    getBmpFromBuffer();
    Invalidate(true);
    Cursor.Current = Cursors.Default;

    int j = 0;
    lock (bmp)
    {
        bmp.Save(j++.ToString(), ImageFormat.Bmp);
    }

    Brownian simulator = new Brownian();
    simulator.scene = scene;
    const double timestep = 0.1e-6;
    simulator.Initialize(timestep);

    //initialization is at time 0
    //first step is at 0 + timestep
    double time = timestep;
    System.Drawing.Color color;
    StringBuilder s = new StringBuilder();

    while (time < 300e-6)
    {
        for (int i = 0; i < 10; i++)
        {
            simulator.SimulateMultiThread(time, 8);
            time += timestep;
        }

        int count = 0;
        foreach (Primitive p in scene.Primitives)
        {
            TriangleSet triSet = p as TriangleSet;
            if (triSet != null)
                foreach (TriangleD tri in triSet.triangles)
                    if (tri.surfaceMolecules != null)
                        foreach (Molecule m in tri.surfaceMolecules)
                            if (m.species.name == "achr" && m.state > 1)
                                count++;
        }

        int nTime = (int)Math.Round(time * 1e6);
    }
s.AppendLine(nTime.ToString() + " " + count.ToString());

//TODO it no longer makes sense to draw individual dots
//go back to using the raytracer to draw a scene of spheres

getBmpFromBuffer();

lock (bmp)
{
    foreach (Particle p in scene.Particles)
    {
        if (scene.Camera.WorldToView(ref viewPoint, p.location))
        {
            color = Color.FromArgb(0, colorFromDouble(.5 + p.location.z / 2.0), 0);
            bmp.SetPixel((int)viewPoint.x, (int)viewPoint.y, color);
        }
    }
    foreach (Primitive p in scene.Primitives)
    {
        if (p is TriangleSet)
        {
            foreach (TriangleD tri in p.triangles)
            {
                if (tri.surfaceMolecules != null)
                {
                    foreach (Molecule m in tri.surfaceMolecules)
                    {
                        if (scene.Camera.WorldToView(ref viewPoint, m.p))
                        {
                            color = Color.FromArgb(colorFromDouble(.5 + m.p.z / 2.0), 0, 0);
                            bmp.SetPixel((int)viewPoint.x, (int)viewPoint.y, color);
                        }
                    }
                }
            }
            bmp.Save(j++.ToString(), ImageFormat.Bmp);
        }
    }
    Refresh();
}

Console.Write(s.ToString());

byte[] rgbValues;

private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();

dlg.Filter = "Brad 3D Xml files(*.xml);*.*xml";
if (dlg.ShowDialog() == DialogResult.OK)
{
    Stream stream = dlg.OpenFile();
    if (stream != null)
```csharp
{ 
    Console.Write("Loading file...");
    Cursor.Current = Cursors.WaitCursor;
    scene = new Scene();
    scene.Camera = new Camera();
    scene.Camera.cx = 30;
    scene.Camera.cy = 15;
    scene.Camera.width = 60.0;
    scene.Camera.height = 30.0;
    scene.Camera.origin.z = 5;
    tracer.setScene(scene);

    rgbValues = new byte[scene.Camera.cx * 3];

    {
        for (int y = 0; y < tracer.scene.Camera.cy; y++)
            for (int x = 0; x < tracer.scene.Camera.cx; x++)
                bmp.SetPixel(x, y, Color.Green);
    }

    scene.ReadFile(stream);
    stream.Close();
    //TODO                    setView(fitToolStripMenuItem.Checked);
    buffer = new byte[tracer.scene.Camera.cy, tracer.scene.Camera.cx, 3];
    tracer.ScanlineComplete += new RayTracer.ScanlineHandler(OnScanlineComplete);
    tracer.RenderComplete += new RayTracer.RenderHandler(OnRenderComplete);
    //TODO find a faster way to initialize
    for (int y = 0; y < tracer.scene.Camera.cy; y++)
        for (int x = 0; x < tracer.scene.Camera.cx; x++)
            for (int c = 0; c < 3; c++)
                buffer[y, x, c] = 128;
    last = DateTime.Now;
    tracer.RenderMultiThread(16);
}
}

private void Form1_Paint(object sender, PaintEventArgs e)
{
    if (bmp != null)
    {
        lock (bmp)
        {
            Rectangle rc = ClientRectangle;
            double aspect1 = bmp.Width / (double)bmp.Height;
            double aspect2 = rc.Width / (double)rc.Height;
            if (aspect1 > aspect2)
                rc.Height = (int)(rc.Height * (aspect2 / aspect1));
            else rc.Width = (int)(rc.Width * (aspect1 / aspect2));
        }
    }
}
private void Form1_Resize(object sender, EventArgs e)
{
    Invalidate();
}

using Utilities;

namespace RayTracer
{
    public class Material
    {
        public Material()
        {
            color = new ColorD(0, 0, 0);
            setDefaults();
        }

        public Material(double r, double g, double b)
        {
            color = new ColorD(r, g, b);
            setDefaults();
        }

        public Material(ColorD c)
        {
            color = c;
            setDefaults();
        }

        void setDefaults()
        {
            specularPower = 1;
            specularShine = 10;
            indexOfRefraction = 1;
            reflection = 0;
            opacity = 1;
            _multiplierForRender = 0;
        }

        public Material Clone()
        {
            Material m = new Material(color);
        }
    }
}
m.specularPower = specularPower;
m.specularShine = specularShine;
m.indexOfRefraction = indexOfRefraction;
m.reflection = reflection;
m.opacity = opacity;
return m;
}

// speed optimization
double _multiplierForRender;
internal double multiplierForRender
{
    get
    {
        if (_multiplierForRender == 0)
        {
            if (texture != null)
            {
                _multiplierForRender = opacity * (1 - reflection) / 255.0 * texture.multiply;
            }
            else _multiplierForRender = 1.0;
        }
        return _multiplierForRender;
    }
}

public string name;
public ColorD color;
public double specularPower;
public double specularShine;
public double indexOfRefraction;
public double reflection;
public double opacity;
public Texture texture;
}

using System.Collections.Generic;
using Utilities;

namespace Simulation
{
    public class Particle
    {
        public Point3D location;
        public bool toDelete;
    }

    public class ReleaseSite
    {
private List<double> events;
private int nextEvent = 0;
private Point3D location;

public ReleaseSite(
    Point3D location,
    double startTime,
    double releaseInterval, //Time between release events in a train
    double trainInterval, //Time between trains
    double trainDuration,
    double numberOfTrains)
{
    this.location = location;

double time = startTime, time2;
events = new List<double>();

    for (int i = 0; i < numberOfTrains; i++)
    {
        time2 = time + trainDuration;
        while (time < time2)
        {
            events.Add(time);
            time += releaseInterval;
        }
        time += trainInterval;
    }
}

public void Simulate(double time, IList<Particle> particles)
{
    while (true)
    {
        if (nextEvent < events.Count && events[nextEvent] <= time)
        {
            for (int i = 0; i < 10; i++)
            {
                Particle p = new Particle();
                p.location = location;
                particles.Add(p);
            }
            nextEvent++;
        }
        else return;
    }
}
using System.Drawing;

namespace splat
{
    /// <summary>
    /// Summary description for Pixel.
    /// </summary>
    public struct Pixel
    {
        public Pixel(float a, byte r, byte g, byte b)
        {
            //The compiler won't let me initialize this in a function call
            //fromArgb(a, r, g, b);
            this.r = r;
            this.g = g;
            this.b = b;
            this.a = a;
        }

        public byte r, g, b;
        public float a;
        public void fromArgb(float a, int r, int g, int b)
        {
            this.r = (byte)r;
            this.g = (byte)g;
            this.b = (byte)b;
            this.a = a;
        }

        public void draw(int x, int y, Bitmap dst)
        {
            if (this.a > 0)
                dst.SetPixel(x, y, Color.FromArgb(255, r, g, b));
        }
    }
}

using System;

namespace Utilities
{
    /// <summary>
    /// 2 dimensional point with double coordinates
    /// </summary>
    public struct Point2D
    {
        public Point2D(double x, double y)
        {
            public double x, y;
        }
    }
}
{  
  this.x = x;  this.y = y;
}

public Point2D Clone()
{
  return new Point2D(x, y);
}

public void setXY(double x, double y)
{
  this.x = x;  this.y = y;
}

public void invert()
{
  x = -x;
  y = -y;
}

public static Point2D operator -(Point2D p1, Point2D p2)
{
  return new Point2D(p1.x - p2.x, p1.y - p2.y);
}

public static Point2D operator +(Point2D p1, Point2D p2)
{
  return new Point2D(p1.x + p2.x, p1.y + p2.y);
}

public double magnitude()
{
  return Math.Sqrt(x * x + y * y);
}

public bool normalize()
{
  double m = magnitude();

  if (m > 0.0)
  {
    x = x / m;
    y = y / m;
    return true;
  }
  else return false;
}

public bool normalize(double scale)
```csharp
double m = magnitude();

if (m > 0.0)
{
    x = scale * x / m;
    y = scale * y / m;
    return true;
}
else return false;

public void scale(double scale)
{
    x *= scale;
    y *= scale;
}

using System;

namespace Utilities
{
    /// <summary>
    /// Utility functions for a point or vector in 3 space
    /// Ported from Volatin code
    ///
    /// Similar to Point3F, but uses doubles
    /// This should really be a generic class, however:
    /// C# 2.0 does not yet support operator contraints for generics
    /// Also, C# does not allow inheritance with structs
    /// Keeping this class performance optimized is important
    /// </summary>
    public struct Point3D
    {
        public double x, y, z;

        public Point3D(double x, double y, double z)
        {
            this.x = x; this.y = y; this.z = z;
        }

        public Point3D Clone()
        {
            return new Point3D(x, y, z);
        }

        public void setXYZ(double x, double y, double z)
        {
```
```java
this.x = x; this.y = y; this.z = z;

public void rotateZ(double a)
{
    double dx, dy, sina, cosa;

    sina = Math.Sin(a);
    cosa = Math.Cos(a);

    dx = x * cosa - y * sina;
    dy = x * sina + y * cosa;
    x = dx;
    y = dy;
}

public void rotateZ(double a, Point3D center)
{
    this.Subtract(center);
    rotateZ(a);
    this.Add(center);
}

public void rotateX(double a, Point3D center)
{
    this.Subtract(center);
    rotateX(a);
    this.Add(center);
}

public void rotateY(double a, Point3D center)
{
    this.Subtract(center);
    rotateY(a);
    this.Add(center);
}

public void rotateY(double a)
{
    double dx, dz, sina, cosa;

    sina = Math.Sin(a);
    cosa = Math.Cos(a);

    dx = x * cosa - z * sina;
    dz = x * sina + z * cosa;
    x = dx;
    z = dz;
}
public void rotateX(double a) {
    double dy, dz, sina, cosa;
    sina = Math.Sin(a);
    cosa = Math.Cos(a);
    dy = y * cosa - z * sina;
    dz = y * sina + z * cosa;
    y = dy;
    z = dz;
}

public double dot(Point3D v) {
    return x * v.x + y * v.y + z * v.z;
}

public void invert() {
    x = -x;
    y = -y;
    z = -z;
}

public static Point3D cross(Point3D a, Point3D b) {
    Point3D p = new Point3D();
    p.x = a.y * b.z - a.z * b.y;
    p.y = a.z * b.x - a.x * b.z;
    p.z = a.x * b.y - a.y * b.x;
    return p;
}

private void Add(double x, double y, double z) {
    this.x += x;
    this.y += y;
    this.z += z;
}

private void Subtract(double x, double y, double z) {
    Add(-x, -y, -z);
}

private void Add(Point3D p) {
    Add(p.x, p.y, p.z);
private void Subtract(Point3D p)
{
    Subtract(p.x, p.y, p.z);
}

public static Point3D operator -(Point3D p1, Point3D p2)
{
    return new Point3D(p1.x - p2.x, p1.y - p2.y, p1.z - p2.z);
}

public static Point3D operator +(Point3D p1, Point3D p2)
{
    return new Point3D(p1.x + p2.x, p1.y + p2.y, p1.z + p2.z);
}

public double magnitude()
{
    return Math.Sqrt(x * x + y * y + z * z);
}

public bool normalize()
{
    double m = magnitude();
    if (m > 0.0)
    {
        x = x / m;
        y = y / m;
        z = z / m;
        return true;
    }
    else return false;
}

public bool normalize(double scale)
{
    double m = magnitude();
    if (m > 0.0)
    {
        x = scale * x / m;
        y = scale * y / m;
        z = scale * z / m;
        return true;
    }
    else return false;
}
public void scale(double scale)
{
    x *= scale;
    y *= scale;
    z *= scale;
}

public double this[int index]
{
    get
    {
        switch(index)
        {
        case 0:
            return x;
        case 1:
            return y;
        case 2:
            return z;
        default:
            throw new ArgumentOutOfRangeException();
        }
    }
}

public static Point3D Reflect(Point3D normal, Point3D direction)
{
    Point3D reflection = normal.Clone();
    reflection.scale(2 * direction.dot(normal));
    reflection = direction - reflection;
    if (reflection.magnitude() > 1.1 || reflection.magnitude() < 0.9)
        throw new Exception("bad vector");
    return reflection;
}

public static bool Refract(out Point3D refracted, Point3D direction, Point3D normal, double newIor, double oldIor)
{
    throw new NotImplementedException();
    refracted = new Point3D(0, 0, 0);
    return false; //TODO
}

using System;
using Utilities;

namespace RayTracer
{
    public class PointLightD
    {//this is becoming complicated, and is no longer a simple point light
        public PointLightD(double x, double y, double z)
        {
            center = new Point3D(x, y, z);
            intensity = new ColorD(1, 1, 1);
            minFalloff = 0.01;
        }

        public Point3D center;
        ColorD intensity;
        double minFalloff; //intensity begins to fall off after this distance
        private double radius;
        public Point3D shade;
        private double shadeAngle;
        private double shadeDot; //cosine of the shade angle for dot product comparison
        double minFalloffSquared;
        private double brightness;

        public double MinFalloffSquared
        {
            get
            {
                if (minFalloffSquared == 0)
                    minFalloffSquared = minFalloff*minFalloff;
                return minFalloffSquared;
            }
        }

        public double MinFalloff
        {
            get { return minFalloff; }
            set
            {
                minFalloff = value;
                minFalloffSquared = 0; //calculate when/if needed
            }
        }

        public ColorD Intensity
        {
            get { return intensity; }
            set
            { }
intensity = value;
brightness = Math.Max(Math.Max(value.r, value.g), value.b);
}

public double Brightness
{
    get { return brightness; }
}

public double Radius
{
    get { return radius; }
    set { radius = value; }
}

public double ShadeAngle
{
    get { return shadeAngle; }
    set
    {
        shadeAngle = value;
        shadeDot = Math.Cos(Math.PI * shadeAngle / 180.0);
    }
}

public double ShadeDot
{
    get { return shadeDot; }
}

using Utilities;

namespace RayTracer
{
    public abstract class Primitive
    {
        public Material material;

        //returns true if a ray hits the primitive
        //MUST set the distance in the trace info if there is a hit
        //otherwise may prepopulate as much or as little TraceInfo as you want
        public abstract bool intersectRay(Point3D origin, Point3D direction, TraceInfo info);

        public virtual bool intersectRay(Point3D origin, Point3D direction, TraceInfo info, double distance)
        {
            //optimized in case we know the ray can only travel at most a certain distance
            //if this is not overridden, it will use the standard calculation
        }
    }
}
namespace RayTracer
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Threading;
using Utilities;

namespace RayTracer
{
    public class RayTracer
    {
        private class RayTracerThread
        {
            public RayTracerThread(RayTracer tracer, int y1, int y2, RayTracerThread[] threads)
            {
                this.tracer = tracer;
                this.y1 = y1;
                this.y2 = y2;
                this.threads = threads;
            }
            return intersectRay(origin, direction, info);
        }
    }
}
public RayTracer tracer;
public int y1, y2;
private bool done;
public RayTracerThread[] threads;

private void Render(object threadContext)
{
    Render();
}

private void Render()
{
    done = false;

    tracer.Render(y1, y2);

    //critical region
    //only 1 thread can exit at a time
    //(otherwise 2 threads can think they are last)
    Mutex mutext = new Mutex(false, "ThreadComplete");
    mutext.WaitOne();
    done = true;
    foreach (RayTracerThread thread in threads)
    {
        if (!thread.done)
        {
            mutext.ReleaseMutex();
            return;
        }
    }
    mutext.ReleaseMutex();
    if (null != tracer.RenderComplete)
        tracer.RenderComplete();
}

public void Start()
{
    ThreadPool.QueueUserWorkItem(Render);
}

const double iorAir = 1.00027712; //index of refraction of air
//http://hypertextbook.com/physics/waves/refraction/

ColorD[,] buffer;
private Scene _scene;
double minWeight; //minimum contribution a ray must have to be traced
double maxDepth; //maximum number of times a ray 'bounces'
public delegate void ScanlineHandler(int y);
public delegate void RenderHandler();

public ScanlineHandler ScanlineComplete;
public RenderHandler RenderComplete;

private Random rand;

public RayTracer()
{
    minWeight = 0.1;
    maxDepth = 100;
    rand = new Random();
}

#region public properties

public ColorD[,] Buffer
{
    get { return buffer; }
}

public double MinWeight
{
    get { return minWeight; }
    set { minWeight = value; }
}

public double MaxDepth
{
    get { return maxDepth; }
    set { maxDepth = value; }
}

public Scene scene
{
    get { return _scene; }
}

public void setScene(Scene scene)
{
    _scene = scene;
}

#endregion public properties

/*ColorD trace(
    Point3D origin,
    Point3D direction,
    int depth,
double weight,
double indexOfRefraction,
Primitive exclude)
{
    ColorD color = new ColorD(0, 0, 0);
    Point3D reflection;
    TraceInfo info = new TraceInfo(),
    min = null;

    foreach (Primitive primitive in scene.Primatives)
    {
        if (primitive != exclude)
        {
            info.primitive = primitive; //TODO find a better way to link these 2 structures
            if (primitive.intersectRay(origin, direction, info))
            {
                if (min == null || info.distance < min.distance)
                {
                    min = info;
                    info = new TraceInfo();
                }
                if (min.distance < 0)
                    break;
            }
        }
    }

    if (min == null)
        return color; //no intersecting objects, or no objects at all
    
    min.calcIntersection(origin, direction);
    min.primitive.postIntersect(min);
    min.view = direction;

    color += blinn(min) * weight;
    if (depth < MaxDepth)
    {
        double nextWeight = weight * min.primitive.material.reflection;
        if (nextWeight >= MinWeight)
        {
            reflection = min.normal;
            reflection.scale(2 * direction.dot(min.normal));
            reflection = direction - reflection;
            //no object should reflect itself
            color += trace(min.Intersection, reflection, depth + 1, nextWeight, indexOfRefraction,
            min.primitive);
        }
    }

    nextWeight = weight * (1.0 - min.primitive.material.opacity);
if (nextWeight >= MinWeight)
{
    color += refract(direction, min, depth + 1, nextWeight, indexOfRefraction);
}
}

return color;
*/

private TraceInfo getIntersect(Point3D direction, Point3D origin, TraceInfo previous)
{
    TraceInfo info = new TraceInfo(),
    min = null;
    bool intersect;

    foreach (Primitive primitive in scene.Primitives)
    {
        if (previous == null
            || primitive != previous.primitive
            || previous.primitive is SphereD)
        {
            if (primitive.material == null
                || primitive.material.opacity > 0)//TODO remove this when refractions are put back on
            {
                info.primitive = primitive; //TODO find a better way to link these 2 structures
                if (min == null)
                    intersect = primitive.intersectRay(origin, direction, info);
                else intersect = primitive.intersectRay(origin, direction, info, min.distance);
                if (intersect)
                {
                    if (min == null || info.distance < min.distance)
                    {
                        min = info;
                        info = new TraceInfo();
                    }
                    if (min.distance < 0)
                        break;
                }
            }
        }
    }
    return min;
}

ColorD trace(
    Point3D origin,
    Point3D direction,
    int depth,
    double weight,
    Stack<double> ior,
TraceInfo previous)
{
    TraceInfo min = getIntersect(direction, origin, previous);

    if (min == null)
        return new ColorD(0, 0, 0); // no intersecting objects, or no objects at all

    min.calcIntersection(origin, direction);
    min.primitive.postIntersect(min);
    min.view = direction;

    return trace2(depth, direction, ior, min, weight);
}

private static ColorD getMaterialColor(TraceInfo info)
{
    ColorD result;
    Material material = info.primitive.material;
    if (null != material.texture)
    {
        result = getTextureColor(info, material);
        result *= material.multiplierForRender;
        return result;
    }
    else return material.color;
}

ColorD illumination(TraceInfo info, double weight, Stack<double> ior, out bool wasFast)
// does not scale by weight
{
    wasFast = false;
    // minimum distance from a light to calculate falloff from
    // needed to avoid ridicoulously bright values very close to a point light

    ColorD surfaceColor = getMaterialColor(info);

    // calculate primary and specular lighting
    return blinn(surfaceColor, info);
}

private ColorD trace2(int depth, Point3D direction, Stack<double> ior, TraceInfo info, double weight)
{
    Point3D reflection;
    ColorD color = new ColorD(0, 0, 0);
    double nextWeight;
    // primary illumination and diffuse reflection
    double w2 = weight * info.primitive.material.opacity;
    bool wasFast;
    color += w2 * illumination(info, w2, ior, out wasFast);
// glossy reflection and refraction
if (depth < maxDepth && !wasFast)
{
    double r = info.primitive.material.reflection;
    if (r > 0)
    {
        nextWeight = weight * r;
        if (nextWeight >= MinWeight)
        {
            reflection = Point3D.Reflex(info.normal, direction);
            color += trace(info.Intersection, reflection, depth + 1, nextWeight, ior, info);
        }
    }
}

double o = info.primitive.material.opacity;
if (o < 1)
{
    nextWeight = weight * (1.0 - o);
    if (nextWeight >= MinWeight)
    {
        color += refract(direction, info, depth + 1, nextWeight, ior);
    }
}

return color;

private ColorD refract(
    Point3D direction,
    TraceInfo info,
    int depth,
    double weight,
    Stack<double> ior)
{

double newlor, oldlor;
oldlor = ior.Peek();
newlor = setNextIOR(info, ior);
Point3D refraction;

if (Point3D.Refract(out refraction, direction, info.normal, newlor, oldlor))
{
    if (info.next != null)
    {
        TraceInfo next = new TraceInfo();
        // avoids starting inside the sphere to bypass roundoff error
        Point3D offset = info.Intersection;
        offset -= refraction;
        next.isInternal = true;
        if (info.next.primitive.intersectRay(offset, refraction, next))
        {  
            // do something with next
        }
    }
}
96

```c
{
    next.primitive = info.next.primitive;
    next.calcIntersection(offset, direction);
    next.primitive.postIntersect(next);
    next.view = direction;
    return trace2(depth, refraction, ior, next, weight);
}
return new ColorD(100, 0, 0);
// this is an error, kill the ray and report bright red
// black is better for production, but red is easy to debug!
}
else {
    return trace(
        info.Intersection,
        refraction,
        depth,
        weight,
        ior,
        info);
}
}

} else {
    // total internal reflection/refraction
    // this can and does happen all the time if we do things correctly
    ior.Push(oldIor);
    Point3D reflection = Point3D.Reflect(info.normal, direction);
    return trace(
        info.Intersection,
        reflection,
        depth,
        weight,
        ior,
        info);
}
}

/*
 * private ColorD refract(
 *     Point3D direction,
 *     TraceInfo info,
 *     int depth,
 *     double weight,
 *     double oldIndexOfRefraction)
 */
{
    double nr = oldIndexOfRefraction / info.primitive.material.indexOfRefraction;
    double a = info.normal.dot(direction);
    double b = 1.0 - nr*nr*(1.0 - a*a);
    if (b >= 0) { // if b < 0, we have total internal refraction
        Point3D refraction = info.normal;
    }
```
refraction.scale(nr * a - Math.Sqrt(b));
Point3D r2 = direction;
r2.scale(nr);
refraction -= r2;
refraction.normalize();
return trace(info.Intersection, refraction, depth, weight,
info.primitive.material.indexOfRefraction, null);
}
return new ColorD(0,0,0);
*/

private static double setNextIOR(TraceInfo info, Stack<double> ior)
{
    double newIor;
    if (info.isInternal)
    {
        ior.Pop();
        //HACK to fix a rare crash with an illogical ray that exist more than it enters
        if (ior.Count == 0)
            ior.Push(iorAir);
        newIor = ior.Peek();
    }
    else
    {
        newIor = info.primitive.material.indexOfRefraction;
        ior.Push(newIor);
    }
    return newIor;
}

bool traceShadow(Point3D origin, 
Point3D lightVector, 
double distance, 
Primitive exclude)
//returns true if we are in the light
//returns false if we are in shadow
//does not attempt refraction
{
    return true; //No shadows for this simulation
    /*
    Point3D direction = lightVector;
direction.invert();
    TraceInfo info = new TraceInfo();

    foreach (Primitive primitive in scene.Primatives)
    {
        //since we do not have primatives that can cast shadows onto themselves,
        //we can go faster by avoiding these calculations
// This also avoids shadows being cast from a triangle
// onto itself due to roundoff error
if (primitive != exclude)
{
    if (primitive.intersectRay(origin, direction, info))
    {
        if (info.distance < distance)
            return false;
    }
}

return true;/*

private ColorD blinn(ColorD surfaceColor, TraceInfo info)
{
    double diffuse;
    double distance;
    double falloff;
    Point3D h;
    Point3D l;
    double specular;

    ColorD result = new ColorD(0, 0, 0);
    Material material = info.primitive.material;

    foreach (PointLightD light in scene.Lights)
    {
        l = info.Intersection - light.center;
        distance = l.magnitude();
        //l.normalize(); avoid computing magnitude twice with library's normalization
        l.x /= distance;
        l.y /= distance;
        l.z /= distance;

        // check for a light shade in the direction we are coming from
        if (light.ShadeAngle > 0)
        {
            if (light.shade.dot(l) >= light.ShadeDot)
                continue;
        }

        if (traceShadow(info.Intersection, l, distance, info.primitive))
        {
            if (distance > light.Radius)
                falloff = distance * distance;
            else falloff = light.Radius * light.Radius;

            // diffuse = diffuse contribution of light, not diffuse reflection
diffuse = l.dot(info.normal) / falloff;
if (diffuse < 0)
    diffuse = 0 - diffuse; // 2 sided

if (diffuse > 0)
{
    result += light.Intensity * diffuse * surfaceColor;

    if (material.specularPower > 0 && material.specularShine > 0)
    {
        h = l + info.view;
        h.normalize();
        specular = info.normal.dot(h);
        if (specular > 0)
        {
            specular = Math.Pow(specular, material.specularShine);
            specular *= light.Brightness;
            specular *= material.specularPower;
            specular /= falloff;
            result.r += specular;
            result.g += specular;
            result.b += specular;
        }
    }
}
result += scene.Ambient;
return result;

/* ColorD blinn(TraceInfo info)
{
    ColorD color = new ColorD(0,0,0), color2;
    Point3D l;
    Point3D h;
    double diffuse, specular, distance, falloff;
    // minimum distance from a light to calculate falloff from
    // needed to avoid ridicoulously bright values very close to a point light

    Material material = info.primitive.material;
    if (null != material.texture)
    {
        Bitmap bmp = material.texture.bitmap;
        int width = bmp.Width;
        int height = bmp.Height;

        Color bmpColor = bmp.GetPixel((int) (info.uv.x * width) % width,
                                       (int)(info.uv.y * height) % height);
color2 = new ColorD(bmpColor.R, bmpColor.G, bmpColor.B);
    color2 *= material.multiplierForRender;
}
else color2 = material.color;

foreach (PointLightD light in scene.Lights)
{
    l = info.Intersection - light.center;
    distance = l.magnitude();
    //l.normalize(); avoid computing magnitude twice with libraries' normalization
    l.x /= distance;
    l.y /= distance;
    l.z /= distance;

    if (traceShadow(info.Intersection, l, distance, info.primitive))
    {
        if (distance > light.MinFalloff)
            falloff = distance*distance;
        else falloff = light.MinFalloffSquared;

        diffuse = 0 - l.dot(info.normal) / falloff;

        if (diffuse > 0)
        {
            color += light.Intensity * diffuse * color2;

            if (material.specularPower > 0 && material.specularShine > 0)
            {
                h = l + info.view;
                h.normalize();
                specular = info.normal.dot(h);
                specular = Math.Pow(specular, material.specularShine);
                specular *= light.Brightness;
                specular *= material.specularPower;
                specular /= falloff;
                color.r += specular;
                color.g += specular;
                color.b += specular;
            }
        }
    }
}
}

color += scene.Ambient;
return color;
}*/

private static ColorD getTextureColor(TraceInfo info, Material material)
{
    Bitmap bmp = material.texture.bitmap;
Color bmpColor1, bmpColor2, bmpColor3, bmpColor4;
double uw, vw;

lock(bmp)
{
    uw = info.uv.x * bmp.Width;
    vw = info.uv.y * bmp.Height;

    int u1 = (int)(uw) % bmp.Width,
            v1 = (int)(vw) % bmp.Height,
            u2 = (u1 + 1) % bmp.Width,
            v2 = (v1 + 1) % bmp.Height;

    bmpColor1 = bmp.GetPixel(u1, v1);
    bmpColor2 = bmp.GetPixel(u2, v1);
    bmpColor3 = bmp.GetPixel(u1, v2);
    bmpColor4 = bmp.GetPixel(u2, v2);
}

uw -= (int)uw;
vw -= (int)vw;

double w1 = uw * vw;
double w2 = (1 - uw) * vw;
double w3 = uw * (1 - vw);
double w4 = (1 - uw) * (1 - vw);

double r = bmpColor1.R * w1
            + bmpColor2.R * w2
            + bmpColor3.R * w3
            + bmpColor4.R * w4;

double g = bmpColor1.G * w1
            + bmpColor2.G * w2
            + bmpColor3.G * w3
            + bmpColor4.G * w4;

double b = bmpColor1.B * w1
            + bmpColor2.B * w2
            + bmpColor3.B * w3
            + bmpColor4.B * w4;

    return new ColorD(r, g, b);
}

private void RenderRay(Point3D dir, Point3D origin, int x, int y)
{
    Point3D dirN;
    dirN = dir;
dirN.normalize();
Stack<double> ior = new Stack<double>();
ior.Push(iorAir);
buffer[x, y] += trace(origin, dirN, 0, 1, ior, null);
}

private Point3D RenderX(Point3D dir, ref Point3D dir2, Point3D origin, int x, double xDelta, int y)
{
    buffer[x, y] = new ColorD(0, 0, 0);
    RenderRay(dir, origin, x, y);
    dir.x += xDelta;
    RenderRay(dir, origin, x, y);
    dir.x += xDelta;
    RenderRay(dir2, origin, x, y);
    dir2.x += xDelta;
    RenderRay(dir2, origin, x, y);
    dir2.x += xDelta;
    buffer[x, y] /= 4;
    return dir;
}

/*
   public void Render()
{
    Point3D dir, dirN, origin;
    int x, y;
    double xDelta, yDelta;
    double startX;

    dir = new Point3D(0, 0, 0);
    origin = new Point3D(0, 0, 10);
    xDelta = cameraWidth / bufferWidth / 180.0;
    yDelta = cameraHeight / bufferHeight / 180.0;

    startX = 0 - cameraWidth / 360.0;
    dir.x = startX;
    dir.y = cameraHeight / 360.0;
    dir.z = -1.0;

    for (y = 0; y < bufferHeight; y++)
    {
        dir.x = startX;
        for (x = 0; x < bufferWidth; x++)
        {
            dirN = dir;
            */
public void RenderMultiThread(int threadCount)
{
    int delta = scene.Camera.cy / threadCount, y;
    buffer = new ColorD[scene.Camera.cx, scene.Camera.cy];
    y = 0;
    RayTracerThread[] threads = new RayTracerThread[threadCount];
    for (int i = 0; i < threadCount - 1; i++)
    {
        threads[i] = new RayTracerThread(this, y, y + delta, threads);
        y += delta;
    }
    threads[threadCount - 1] = new RayTracerThread(this, y, scene.Camera.cy - 1, threads);
    foreach (RayTracerThread thread in threads)
    { thread.Start();
    }
}

public void Render(int startY, int endY)
{
    Point3D dir, dir2;
    int x, y;
    double xDelta, yDelta;
    double startX;
    dir = scene.Camera.direction;
    xDelta = scene.Camera.width / scene.Camera.cx / 180.0 / 2.0;
    yDelta = scene.Camera.height / scene.Camera.cy / 180.0;
    startX = dir.x - scene.Camera.width / 360.0;
    dir = scene.Camera.direction;
    dir.x = startX;
    dir.y += scene.Camera.height / 360.0;
    dir.y -= yDelta * startY;
    dir2 = dir;
    dir2.y -= yDelta / 2;
    for (y = startY; y <= endY; y++)
    {
        dir.x = startX;
dir2.x = startX;
for (x = 0; x < scene.Camera.cx; x++)
{
    dir = RenderX(dir, ref dir2, scene.Camera.origin, x, xDelta, y);
}
dir.y -= yDelta;
dir2.y -= yDelta;

if (ScanlineComplete != null)
    ScanlineComplete(y);
}
}
}
}

using System;
using System.Collections.Generic;
using System.IO;
using System.Xml;
using Simulation;
using Utilities;

namespace RayTracer
{
    public class Scene
    {
        private class Transform : IComparable<Transform>
        {
            public Transform()
            {
                matrix = new double[4,4];
            }

            public int CompareTo(Transform b)
            {
                return order.CompareTo(b.order);
            }

            public double[,] matrix;
            public int order;
        }

        readonly List<Material> materials;
        readonly List<Texture> textures;
        readonly List<Primitive> primitives;
        readonly List<PointLightD> lights;
        readonly List<Particle> particles;
        readonly List<ReleaseSite> releaseSites;
ColorD ambient;
Camera camera;
public Random rand;

public Scene()
{
    primitives = new List<Primitive>();
    lights = new List<PointLightD>();
    materials = new List<Material>();
    textures = new List<Texture>();
    particles = new List<Particle>();
    releaseSites = new List<ReleaseSite>();
    ache = new Species();
    ache.name = "ache";
    achr = new Species();
    achr.name = "achr";
}

public List<Primitive> Primitives
{
    get { return primitives; }
}

public List<PointLightD> Lights
{
    get { return lights; }
}

public ColorD Ambient
{
    get { return ambient; }
}

public Camera Camera
{
    get { return camera; }
    set { camera = value; }
}

public List<Particle> Particles
{
    get { return particles; }
}

public List<ReleaseSite> ReleaseSites
{
    get { return releaseSites; }
}

private void readSphere(Xmlelment element)
{ 
    SphereD sphere = new SphereD();
    sphere.radius = double.Parse(element.Attributes["radius"].Value);
    sphere.material = findMaterial(element.Attributes["material"].Value);
    foreach (XmlNode child in element.ChildNodes)
    {
        if (child.Name == "center")
        {
            double x, y, z;
            x = double.Parse(child.Attributes["x"].Value);
            y = double.Parse(child.Attributes["y"].Value);
            z = double.Parse(child.Attributes["z"].Value);
            sphere.center = new Point3D(x, y, z);
            Primitives.Add(sphere);
            return;
        }
    }
    throw new FormatException("bad sphere");
}

private void readPointLight(XmlNode element)
{
    PointLightD light;
    double x, y, z, r, g, b;
    x = y = z = 0;
    r = g = b = 1;
    foreach (XmlNode child in element.ChildNodes)
    {
        if (child.Name == "center")
        {
            x = double.Parse(child.Attributes["x"].Value);
            y = double.Parse(child.Attributes["y"].Value);
            z = double.Parse(child.Attributes["z"].Value);
        }
        if (child.Name == "intensity")
        {
            r = double.Parse(child.Attributes["r"].Value);
            g = double.Parse(child.Attributes["g"].Value);
            b = double.Parse(child.Attributes["b"].Value);
        }
    }
    light = new PointLightD(x, y, z);
    light.Intensity = new ColorD(r, g, b);
   XmlAttribute minFalloff = element.Attributes["minFalloff"]; 
    if (null != minFalloff)
        light.MinFalloff = double.Parse(minFalloff.Value);
    Lights.Add(light);
}

private void readAmbientLight(XmlNode element)
{  
double r, g, b;
  foreach (XmlNode child in element.ChildNodes)
  {
     if (child.Name == "intensity")
     {
       r = double.Parse(child.Attributes["r"].Value);
       g = double.Parse(child.Attributes["g"].Value);
       b = double.Parse(child.Attributes["b"].Value);
       ambient = new ColorD(r, g, b);
       return;
     }
  }
  throw new FormatException("bad ambient light parameter");
}

private Point3D getReleaseLocation()
{
  return new Point3D(0, 0, 0);
}

private void readRelease(XmlNode element)
{
  ReleaseSite r;
  double delay, n;
  Point3D location;
  delay = double.Parse(element.Attributes["DELAY"].Value);
  n = double.Parse(element.Attributes["NUMBER_TO_RELEASE"].Value);
  location = getReleaseLocation();
  //need to transform from mdl to 3d studio max coordinates
  double t = location.z;
  location.z = location.y;
  location.y = 0 - t;
  //hardcoded parameters are the same for each release site
  //these still should not be hard coded
  //  r = new ReleaseSite(location, delay, 2e-6, 4e-6, 1e-6, 1);
  r = new ReleaseSite(location, delay, 2e-6, 4e-6, 1e-6, 1);
  releaseSites.Add(r);
}

public void ReadFile(string filename)
{
  XmlDocument doc = new XmlDocument();
  doc.Load(filename);
  ReadFile(doc);
}

public void ReadFile(Stream stream)
{
XmlDocument doc = new XmlDocument();
doc.Load(stream);
ReadFile(doc);

public void ReadFile(XmlDocument doc)
{
    XmlElement element;
    if (rand == null)
        rand = new Random();
    foreach (XmlNode node in doc.DocumentElement.ChildNodes)
    {
        element = node as XmlElement;
        if (element != null)
        {
            switch (element.Name)
            {
                case "light":
                    switch (element.Attributes["type"].Value)
                    {
                        case "point":
                            readPointLight(element);
                            break;
                        case "ambient":
                            readAmbientLight(element);
                            break;
                    }
                    break;
                case "texture":
                    readTexture(element);
                    break;
                case "material":
                    readMaterial(element);
                    break;
                case "sphere":
                    readSphere(element);
                    break;
                case "indexedFaceSet":
                    readFaceSet(element);
                    break;
                case "release":
                    readRelease(element);
                    break;
            }
        }
    }
    int somethingToBreakON = 1;
}

Texture findTexture(string name)
//TODO findTexture and findMaterial use a linear search
//and should be replaced with a better structure (Hashtable?)
//This is trivial, since most files will have few textures and materials

פחד foreach (Texture t in textures)
{
    if (t.name == name)
        return t;
}

throw new FormatException("bad texture name");

Material findMaterial(string name)
{
    foreach (Material m in materials)
    {
        if (m.name == name)
            return m;
    }

    throw new FormatException("bad material name");
}

private void readTexture(XmlNode element)
{
    Texture texture = new Texture(element.Attributes["filename"].Value);
    texture.name = element.Attributes["name"].Value;
    XmlAttribute attr = element.Attributes["uTile"];
    if (attr != null)
        texture.uTile = double.Parse(attr.Value);
    attr = element.Attributes["vTile"];    
    if (attr != null)
        texture.vTile = double.Parse(attr.Value);
    attr = element.Attributes["multiply"];
    if (attr != null)
        texture.multiply = double.Parse(attr.Value);
    textures.Add(texture);
}

private void readMaterial(XmlNode element)
{
    Material mat = new Material();
    mat.name = element.Attributes["name"].Value;
    XmlAttribute attr = element.Attributes["IndexOfRefraction"];
    if (null != attr)
        mat.indexOfRefraction = double.Parse(attr.Value);
    attr = element.Attributes["opacity"];    
    if (null != attr)
        mat.opacity = double.Parse(attr.Value);
    attr = element.Attributes["reflection"];
    if (null != attr)
mat.reflection = double.Parse(attr.Value);
attr = element.Attributes["texture"];  
if (null != attr)
    mat.texture = findTexture(attr.Value);

foreach (XmlNode child in element.ChildNodes)
{
    if (child.Name == "color")
    {
        double r, g, b;
        r = double.Parse(child.Attributes["r"].Value);
        g = double.Parse(child.Attributes["g"].Value);
        b = double.Parse(child.Attributes["b"].Value);
        mat.color = new ColorD(r, g, b);
    }
    if (child.Name == "specular")
    {
        attr = child.Attributes["power"];  
        if (null != attr)
            mat.specularPower = double.Parse(attr.Value);
        attr = child.Attributes["shine"];  
        if (null != attr)
            mat.specularShine = double.Parse(attr.Value);
    }
}
materials.Add(mat);

private void AddSubdiveTriangleSet(TriangleSet set, int subdivide)
{
    TriangleSet left, right;
    if (subdivide == 0)
        primitives.Add(set);
    else
    {
        set.Subdivide(out left, out right);
        subdivide--;
        AddSubdiveTriangleSet(left, subdivide);
        AddSubdiveTriangleSet(right, subdivide);
    }
}

private void addSurfaceMolecule(TriangleD tri, int i, Species s)
{
    Point3D p, p2;
    bool good = false;

    while (!good)
    {
        good = true;
    }
}
p = tri.randomPointOn(rand);
for (int j=0; j < i && good; j++)
{
    p2 = tri.surfaceMolecules[j].p.Clone();
    p2 -= p;
    if (p2.magnitude() > .011)
        good = false;
}
if (good)
{
    Molecule sm = new Molecule();
    sm.p = p;
    sm.species = s;
    tri.surfaceMolecules[i] = sm;
}

private Species ache, achr;

private void addSurfaceMoleculesAchR(TriangleD tri)
{
    double density;

    Point3D center = tri.a + tri.b + tri.c;
    center.scale(1.0 / 3.0);

    //TODO interpolation
    if (center.y < -0.555)
        return;//density=0
    else if (center.y < -0.245)
        density = 7250 * .297;
    else density = 7250;

double a = tri.getArea() * 1000.0;
int molecules = (int)Math.Floor(a);
if (a - molecules > rand.NextDouble())
molecules++;
if (molecules > 0)
{
    tri.surfaceMolecules = new Molecule[molecules];
    for (int i = 0; i < molecules; i++)
        addSurfaceMolecule(tri, i, achr);
}

private void addSurfaceMoleculesAchE(TriangleD tri)
{
    double a = tri.getArea() * 1800.0;
    int molecules = (int)Math.Floor(a);
if (a - molecules > rand.NextDouble())
    molecules++;
if (molecules > 0)
{
    tri.surfaceMolecules = new Molecule[molecules];
    for (int i = 0; i < molecules; i++)
        addSurfaceMolecule(tri, i, ache);
}

private void addSurfaceMolecules(TriangleSet set, string species)
{
    Console.Write("Adding surface molecule " + species);
    if (species == "ache")
    {
        foreach (TriangleD tri in set.triangles)
            addSurfaceMoleculesAchE(tri);
    }
    else if (species == "achr")
    {
        foreach (TriangleD tri in set.triangles)
            addSurfaceMoleculesAchR(tri);
    }
    else throw new Exception("bad molecule species");
    Console.Write("Added surface molecule " + species);
}

private void readFaceSet(XmlNode element)
//TODO this function is too long
//should be split up
{
    /*triangle materials can be specified for either a set
    * of triangles or an individual triangle
    * If it is set in both places, the individual takes precedence
    * Either place may be blank, but if it is not set, that is an error
    */
    string surfaceMolecules = null;
    TriangleSet set = new TriangleSet();
    XmlAttribute attr = element.Attributes["material"];
    Material mat;
    mat = null != attr ? findMaterial(attr.Value) : null;
    set.material = mat;

    //first, load the Coordinates and TextCoords
    string points;
    List<Point3D> pts = new List<Point3D>();
    List<Point2D> txtPts = new List<Point2D>();

    char[] whitespace = new char[] { ' ', '', '
' };
    List<Transform> transforms = new List<Transform>();
    foreach (XmlNode node in element.ChildNodes)
    {
        XmlElement child = node as XmlElement;
if (child != null)
{
  switch (child.Name)
  {
    case "Coordinate":
    {
      points = child.Attributes["point"].Value;
      string[] ary = points.Split(',');
      foreach (string s in ary)
      {
        string[] ary2 = s.Split(whitespace,
          StringSplitOptions.RemoveEmptyEntries);
        if (ary2.Length != 3)
          throw new FormatException("bad coordinate");
        double x, y, z;
        x = double.Parse(ary2[0]);
        y = double.Parse(ary2[1]);
        z = double.Parse(ary2[2]);
        pts.Add(new Point3D(x, y, z));
      }
    }
    break;
    case "TextCoord":
    {
      points = child.Attributes["point"].Value;
      string[] ary = points.Split(',');
      foreach (string s in ary)
      {
        string[] ary2 = s.Split(whitespace,
          StringSplitOptions.RemoveEmptyEntries);
        if (ary2.Length != 2)
          throw new FormatException("bad texture coordinate");
        double x, y;
        x = double.Parse(ary2[0]);
        y = double.Parse(ary2[1]);
        txtPts.Add(new Point2D(x, y));
      }
    }
    break;
    case "transform":
    {
      string matrix = child.Attributes["matrix"].Value;
      string[] ary = matrix.Split(whitespace, StringSplitOptions.RemoveEmptyEntries);
      if (ary.Length != 16)
        throw new FormatException("bad transform matrix");
      Transform t = new Transform();
      for (int row = 0; row < 4; row++)
      {
        for (int col = 0; col < 4; col++)
          t.matrix[row, col] = double.Parse(ary[row * 4 + col]);
        t.order = int.Parse(child.Attributes["order"].Value);
      }
    }
  }
}
transforms.Add(t);

break;

case "surface_molecules":
surfaceMolecules =

break;

child.Attributes["species"].Value;

}
}
}
}

transforms.Sort();
foreach (Transform t in transforms)
{
  for (int i = 0; i < pts.Count; i++)
  {
    Point3D p = pts[i];
    double x = p.x * t.matrix[0, 0]
              + p.y * t.matrix[1, 0]
              + p.z * t.matrix[2, 0]
              + t.matrix[3, 0];
    double y = p.x * t.matrix[0, 1]
              + p.y * t.matrix[1, 1]
              + p.z * t.matrix[2, 1]
              + t.matrix[3, 1];
    double z = p.x * t.matrix[0, 2]
              + p.y * t.matrix[1, 2]
              + p.z * t.matrix[2, 2]
              + t.matrix[3, 2];
    double v = p.x * t.matrix[0, 3]
              + p.y * t.matrix[1, 3]
              + p.z * t.matrix[2, 3]
              + t.matrix[3, 3];
    if (v != 1)
      throw new Exception("TODO");
    p.x = x;
    p.y = y;
    p.z = z;
    pts[i] = p;
  }
}
// next, load the faces

bool reflectiveToMolecules;
if (attr != null && attr.Value == "0")
  reflectiveToMolecules = false;
else reflectiveToMolecules = true;

foreach (XmlNode child in element.ChildNodes)
{

if (child.Name == "face")
{
    TriangleD tri;
    Point3D a, b, c;

    attr = child.Attributes["coordIndex"];  
    if (attr == null)
        throw new FormatException("Bad triangle coordinates");
    string[] ary = attr.Value.Trim().Split(");
    if (ary.Length != 3)
        throw new NotImplementedException("Only triangle polygons are supported");
    a = pts[Int32.Parse(ary[0])];
    b = pts[Int32.Parse(ary[1])];
    c = pts[Int32.Parse(ary[2])];
    tri = new TriangleD(a, b, c);
    tri.reflectiveToMolecules = reflectiveToMolecules;

    attr = child.Attributes["material"];  
    if (attr != null)
        tri.material = findMaterial(attr.Value);
    else tri.material = mat;
    if (null == tri.material)
        throw new FormatException("Triangle must have material");

    attr = child.Attributes["normal"];  
    if (attr == null)
    {
        Point3D line1 = tri.b - tri.a;
        line1.normalize();
        Point3D line2 = tri.c - tri.b;
        line2.normalize();
        tri.normal = Point3D.cross(line2, line1);
        tri.normal.normalize();
    }
    else
    {
        double x, y, z;

        ary = attr.Value.Split( ");
        if (ary.Length != 3)
            throw new NotImplementedException("Bad normal");
        x = double.Parse(ary[0]);
        y = double.Parse(ary[1]);
        z = double.Parse(ary[2]);
        tri.normal = new Point3D(x, y, z);
    }

    attr = child.Attributes["textCoordIndex"];  
    if (attr != null)
    {

ary = attr.Value.Split(' ');
if (ary.Length != 3)
    throw new NotImplementedException("Only triangle polygons are supported");
Point2D ta = txtPts[Int32.Parse(ary[0])];
Point2D tb = txtPts[Int32.Parse(ary[1])];
Point2D tc = txtPts[Int32.Parse(ary[2])];
tri.setUV(ta.x, ta.y, tb.x, tb.y, tc.x, tc.y);
}
set.Add(tri);
}
//end for

int subdivisions;
attr = element.Attributes["subdivisions"];  
if (attr != null)
    subdivisions = Int32.Parse(attr.Value);
else subdivisions = 1;  //split once by default
    attr = element.Attributes["reflectiveToMolecules"];  
    if (surfaceMolecules != null)
        addSurfaceMolecules(set, surfaceMolecules);
AddSubdivideTriangleSet(set, subdivisions);
}//end readFaceSet
}//end class
}//end namespace

using System;
using RayTracer;

namespace Utilities
{
public class SphereD : Primitive
{
    public double radius;
    public Point3D center;

    double _r2;
    public double radiusSquared
    {
        get
        {
            if (_r2 == 0)
                _r2 = radius*radius;
            return _r2;
        }
    }

    public override bool intersectRay(Point3D origin, Point3D direction, TraceInfo info)
    {
        Point3D delta = (center - origin);
double distance = delta.magnitude();
double tca = delta.dot(direction);
double l2hc;
const double MIN_DISTANCE = 0.00001; // avoid error collisions from round off

if (distance > radius) // outside
{
    if (tca < 0)
        return false;
    l2hc = radiusSquared - distance * distance + tca * tca;
    if (l2hc < 0)
        return false;
    if (l2hc == 0) // tangent
        { 
            if (tca < MIN_DISTANCE)
                return false;
            info.distance = tca;
            return true;
        }
    else
        { 
            info.distance = tca - Math.Sqrt(l2hc);
            if (info.distance < MIN_DISTANCE)
                return false;
            return true;
        }
}
else // inside
{
    l2hc = radiusSquared - distance * distance + tca * tca;
    info.distance = tca - Math.Sqrt(l2hc);
    if (info.distance < MIN_DISTANCE)
        return false;
    return true;
}
}

public override void postIntersect(TraceInfo info)
{
    info.normal = info.Intersection - center;
    info.normal.normalize();
    material = info.primitive.material;

    if (material.texture != null)
        // map spheres like cylinders
        // this is probably easiest to generate graphics for
        { 
            double u, v;
if (info.normal.x > 0 || info.normal.z > 0)
{
    Point3D temp = new Point3D(info.normal.x, 0, info.normal.z);
    temp.normalize(); // 2D vector of x z, map to angle of sphere
    u = temp.dot(new Point3D(0, 0, -1));
    if (temp.x > 0) // quadrants 3 and 4
        u = .75 + u / 4.0;
    else // quadrant 1 and 2
        u = .25 - u / 4.0;
}
elserelative u = 0; // arbitrary coordinate for top and bottom of sphere

v = (1.0 - info.normal.y) / 2.0;

u *= material.texture.uTile;
v *= material.texture.vTile;
info.uv = new Point2D(u, v);
}
}

using System.Drawing;

namespace RayTracer
{
    public class Texture
    {
        public string name;
        public double uTile;
        public double vTile;
        public double multiply;
        public Bitmap bitmap;

        public Texture(Bitmap bitmap, double uTile, double vTile)
        {
            this.bitmap = bitmap;
            this.uTile = uTile;
            this.vTile = vTile;
            multiply = 1.0;
        }

        public Texture(string fileName, double uTile, double vTile)
        {
            bitmap = new Bitmap(fileName);
            this.uTile = uTile;
            this.vTile = vTile;
        }
    }
}
multiply = 1.0;

public Texture(string fileName) {
    bitmap = new Bitmap(fileName);
    this.uTile = 1;
    this.vTile = 1;
    multiply = 1.0;
}
namespace Utilities {
    public class TriangleD : Primitive {
        public Point3D a, b, c, normal;
        Point2D[] uv;
        public Molecule[] surfaceMolecules;
        public bool reflectiveToMolecules;

        public TriangleD(Point3D a, Point3D b, Point3D c) {
            this.a = a;
            this.b = b;
            this.c = c;
        }

        public TriangleD(Point3D a, Point3D b, Point3D c, Point3D normal) {
            this.a = a;
            this.b = b;
            this.c = c;
            this.normal = normal;
        }

        public override bool intersectRay(Point3D origin, Point3D direction, TraceInfo info) {
            //from Fast,Minimum Storage Ray/Triangle Intersection
            //Tomas Moller, Ben Trumbore
            //non-culling approach
            
            Point3D edge1, edge2, p, q, tvect;
            double determinant, u, v;
            const double EPSILON = 0.0000001;
            //avoids error values from rays near parallel to the plane of triangle
            //this can cause an artificial miss if the ray really is very close to the triangle
            const double MIN_DISTANCE = 0.0000001;
            //avoids artificial hits from roundoff error
            //usually this is a secondary ray hitting the object it bounced off of
            //this is mostly avoided by skipping these calculations for speed

            edge1 = b - a;
            edge2 = c - a;
            p = Point3D.cross(direction, edge2);
            determinant = edge1.dot(p);

            if (determinant == 0)
                return false;

            //if (determinant > -EPSILON && determinant < EPSILON)
            //    return false;
        }
    }
}
tvect = origin - a;
u = tvect.dot(p) / determinant;

if (u < 0 || u > 1)
    return false;

q = Point3D.cross(tvect, edge1);
v = direction.dot(q) / determinant;
if (v < 0 || u + v > 1)
    return false;

double distance = edge2.dot(q)/determinant;
if (distance <= MIN_DISTANCE)
    return false;
//do not modify info if returning false
info.distance = distance;
//return these in a format we can use later for uv mapping
info.uv.x = 1.0 - u - v;
info.uv.y = u;

return true;
}

public override void postIntersect(TraceInfo info)
{
    info.normal = this.normal;
    if (material.texture != null && this.UV != null)
    {
        //we already have uv coordinates, but we need to map them
        //so we can do a texture lookup
        double w1 = info.uv.x;
        double w2 = info.uv.y;
        double u = w1 * this.UV[0].x + w2 * this.UV[1].x + (1.0 - w1 - w2) * this.UV[2].x;
        double v = w1 * this.UV[0].y + w2 * this.UV[1].y + (1.0 - w1 - w2) * this.UV[2].y;
        u *= material.texture.uTile;
        v *= material.texture.vTile;
        info.uv.x = u;
        info.uv.y = v;
    }
    return;
}

public void setUV(double au, double av, double bu, double bv, double cu, double cv)
{
    uv = new Point2D[3];
    uv[0] = new Point2D(au, av);
    uv[1] = new Point2D(bu, bv);
    uv[2] = new Point2D(cu, cv);
}
public Point2D[] UV
{
    get { return uv; }
}

public double getArea()
{
    //Heron's formula
    //TODO optimize
    double l1, l2, l3, sp;
    l1 = Math.Abs((a - b).magnitude());
    l2 = Math.Abs((a - c).magnitude());
    l3 = Math.Abs((b - c).magnitude());
    sp = (l1 + l2 + l3) / 2.0;
    return Math.Sqrt(sp * (sp - l1) * (sp - l2) * (sp - l3));
}

public Point3D randomPointOn(Random rand)
//Barycentric coordinates
//from http://www.cgafaq.info/wiki/Random_Point_In_Triangle
{
    double ba, bb, bc;
    do
    {
        ba = rand.NextDouble();
        bb = rand.NextDouble();
    } while (ba + bb > 1);
    bc = 1.0 - ba - bb;
    return new Point3D(
        a.x * ba + b.x * bb + c.x * bc,
        a.y * ba + b.y * bb + c.y * bc,
        a.z * ba + b.z * bb + c.z * bc);
}

using System.Collections.Generic;
using RayTracer;

namespace Utilities
{/}
public class TriangleSet : Primitive
{
    public List<TriangleD> triangles;
    private Point3D min, max;

    public TriangleSet()
    {
        triangles = new List<TriangleD>();
        max = new Point3D(
            double.MinValue,
            double.MinValue,
            double.MinValue);
        min = new Point3D(
            double.MaxValue,
            double.MaxValue,
            double.MaxValue);
    }

    private void bound(Point3D p)
    {
        if (p.x < min.x) min.x = p.x;
        if (p.y < min.y) min.y = p.y;
        if (p.z < min.z) min.z = p.z;
        if (p.x > max.x) max.x = p.x;
        if (p.y > max.y) max.y = p.y;
        if (p.z > max.z) max.z = p.z;
    }

    public void Add(TriangleD tri)
    {
        triangles.Add(tri);
        bound(tri.a);
        bound(tri.b);
        bound(tri.c);
    }

    bool RaySlabsIntersection(Point3D origin, Point3D direction)
    {
        return RaySlabsIntersection(min, max, origin, direction);
    }

    static bool RaySlabsIntersection(Point3D min, Point3D max, Point3D origin, Point3D direction)
        //http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm
    {
    }
```csharp
double Tnear = Double.MinValue;
double Tfar = Double.MaxValue;

for (int axis = 0; axis < 3; axis++)
{
    if (!TestAxis(
        origin, direction,
        min, max,
        ref Tnear, ref Tfar,
        axis))
    {
        return false;
    }
}
return true;

static bool RaySlabsIntersection(Point3D min, Point3D max, Point3D origin, Point3D direction, double distance)
{ //http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm
    double Tnear = Double.MinValue;
double Tfar = Double.MaxValue;

    for (int axis = 0; axis < 3; axis++)
    {
        if (!TestAxis(
            origin, direction,
            min, max,
            ref Tnear, ref Tfar,
            axis))
        {
            return false;
        }
    }
    return (Tnear <= distance);
}

private static bool TestAxis(
    Point3D origin,
    Point3D direction,
    Point3D min,
    Point3D max,
    ref double Tnear,
    ref double Tfar, 
    int axis)
{
    if (direction[axis] == 0)
    {
        if (origin[axis] < min[axis])
        {
```
return false;
if (origin[axis] > max[axis])
    return false;
}
else
{
    double T1 = (min[axis] - origin[axis]) / direction[axis];
    double T2 = (max[axis] - origin[axis]) / direction[axis];
    if (T1 > T2)
    {
        double swap = T1;
        T1 = T2;
        T2 = swap;
    }
    if (T1 > Tnear)
        Tnear = T1;
    if (T2 < Tfar)
        Tfar = T2;
    if (Tnear > Tfar)
        return false;
    if (Tfar < 0)
        return false;
}
return true;

/*/bool RaySlabsIntersection(Point3D bmin, Point3D bmax, Point3D origin, Point3D direction)
{
    object inside = ray.Range();
    for (int i = 0; i < 3; i++)
    {
        inside = Intersection(inside, (slab[i].Range() - ray.Origin[i])/ray.Direction[i]);
        if (inside.IsEmpty)
            return false;
    }
    return true;
}*/

public override bool intersectRay(
    Point3D origin,
    Point3D direction,
    TraceInfo info)
{
    if (!RaySlabsIntersection(origin, direction))
        return false;

    return intersectRayAllTriangles(origin, direction, info);
public override bool intersectRay(
    Point3D origin,
    Point3D direction,
    TraceInfo info,
    double distance)
{
    //            if (!RaySlabsIntersection(origin, direction, distance))
    //                return false;

    return intersectRayAllTriangles(origin, direction, info);
}

private bool intersectRayAllTriangles(Point3D origin, Point3D direction, TraceInfo info)
{
    TraceInfo minHit = null;

    foreach (TriangleD tri in triangles)
    {
        if (tri.intersectRay(origin, direction, info))
        {
            if (minHit == null || info.distance < minHit.distance)
            {
                minHit = info;
                minHit.primitive = tri;
            }
            if (minHit.distance < 0)
                throw new System.Exception("minHit < 0");
        }
    }
    if (minHit != null)
        info = minHit;
    return (minHit != null);
}

public void Subdivide(out TriangleSet left, out TriangleSet right, int axis)
{
    double avg = (max[axis] + min[axis]) / 2.0;
    left = new TriangleSet();
    right = new TriangleSet();
    left.material = material;
    right.material = material;
    foreach (TriangleD tri in triangles)
    {
        if (tri.a[axis] < avg || tri.b[axis] < avg || tri.c[axis] < avg)
            left.Add(tri);
        if (tri.a[axis] >= avg || tri.b[axis] >= avg || tri.c[axis] >= avg)
            right.Add(tri);
    }
}
public void Subdivide(out TriangleSet left, out TriangleSet right)
{
    int axis;

    if (max.x - min.x > max.y - min.y)
    {
        if (max.x - min.x > max.z - min.z)
            axis = 0;
        else
            axis = 2;
    }
    else if (max.y - min.y > max.z - min.z)
    {
        axis = 1;
    }
    else
    {
        axis = 2;
        Subdivide(out left, out right, axis);
    }
}