View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DigitalCommons@WPI

Worcester Polytechnic Institute
Digital WPI

Major Qualifying Projects (All Years) | Major Qualifying Projects

October 2014

Simulation of Early C. elegans Embryogenesis

Rachel Ellen Wigell
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation
Wigell, R. E. (2014). *Simulation of Early C. elegans Embryogenesis*. Retrieved from https://digitalcommons.wpi.edu/mqp-all/3692

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.

Simulation of Early *C. elegans* Embryogenesis

A Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

Rachel Wigell CS '15

Advisors:

Professor Matthew Ward, Computer Science Advisor

Professor Elizabeth Ryder, Biology Advisor

Date: October 27, 2014

Abstract

This project simulates the development of the microscopic worm, *C. elegans,* up to the twenty-six cell stage. Advanced concepts in computer graphics, including metaballs, ray tracing, and the Marching Cubes algorithm, were explored in order to depict the irregular shapes found in nature. The forces that dictate cell motion in response to collisions were represented mathematically and included in the model. Finally, the results were compared to empirical data to show that early cell divisions appear to accurately simulate the appearance of biological embryos.

Acknowledgements

To Matt Ward, for his incredible tenacity and dedication to his students and his work during harrowing times.

To Liz Ryder for her endless support as an advisor and advocate.

To Sonia Chernova, for helping me out of a snag.

Table of Contents

1 Introduction 6
2 Background 7
2.1 Metaballs 7
2.2 Rendering methods 9
2.2.1 Ray Tracing 9
2.2.2 Marching Cubes 9
2.3 Picking 10
2.3.1 Color Buffer 10
3 Design 11
3.1 Unexpected complications 11
3.2 Comparison of Metaball rendering methods 12
3.2.1 Ray Tracing 12
3.2.2 Marching Cubes 13
3.3 Cell Motion 13
3.3.1 Local search 13
3.4 Change of equation 14
4 Implementation 15
4.1 Performance 15
4.2 Biological accuracy 17
5 Conclusions and Future Work 19
5.1 Accuracy of shapes 19
5.2 Biological factors 19
5.3 Improvements to the code 20
5.4 Goals that were not achieved 20
6 References 22
7 Appendices 23
7.1 User’s Guide 23
7.2 Developer’s Guide 24

Table of Figures

Figure 1: Metaballs.. 8

Figure 2: Interactions between metaballs.. 8

Figure 3: Four mutually-repulsive metaballs. 12

Figure 4: Effect of adding metaballs. 16

Figure 5: Effect of grid size 16

Figure 6: Two, three, and four cell stage 17

Figure 7: Eight cell stage, lateral and ventral views 18

Figure 8: Twelve-cell stage. 19

1 Introduction

This project simulates the development of *C. elegans* up to the twenty-six cell stage. *C. elegans* is a microscopic nematode worm that is used as a model system to understand many important processes in biology. The first two terms of the project, conducted in a group with biology and bioinformatics students, focused on the biological accuracy of gene expression and protein distribution as reported previously (Warden, 2014). The work for this term, conducted as a solo project and reported here, prioritized the improvement of cell shape and movement realism using advanced computer graphic techniques. The colloquial name for the project is “Simworm,” a term that will be used throughout the paper.

In general, the purpose of designing a simulation is to test hypotheses. Once a simulation is found to match real-world observations, one can alter parameters that are not as easily altered in real life (e.g. fast forward in time) and see a prediction of what the effects might be. This is not a replacement for empirical experiments, since a simulation can never be proven to be exactly correct, but it can serve as useful guidance.

To this end, a simulation must make calculations to discover its own subsequent state without being “told” much information. In order to achieve this, actions must be driven by modelled physical forces and rules corresponding to genetic knowledge of the biological system. By establishing a set of laws to spur action, a simulation can continue even if our knowledge on the system’s behavior ends at a certain time. It can also allow us to alter configurable parameters and see the effect that occurs. This particular simulation hopes to allow users to explore the effects of including different genes and gene rules. Another goal was to simulate mutant organisms, which develop differently due to missing proteins, and to see if establishing a different set of rules results in a different, but also biologically accurate, outcome for the mutant.

In addition to self-determination, other concepts that are explored in this project are the three-dimensional rendering of organic shapes. Shapes and motions are dictated by a set of physical forces, particularly the force of cells pushing against one another, and involve the use of some advanced computer graphics concepts to attain the irregular shapes that cells can take on. The performance of the simulation was then compared against lab data observed in developing *C. elegans* embryos.

2 Background

Since this project is a continuation of previous work, it is important to first note the status of the project at the start of the term (Warden, 2014). Last year, a simulation was developed from scratch in Java, with three-dimensional visuals rendered in Processing, a graphics language which serves as an abstraction for OpenGL (Fry, 2014). The initial iteration of the simulation focused on developing a back-end to handle the basic structure of the model, including cell division and protein inheritance, and was not very visually accurate.

In general, as described in the introduction, a good simulation is dictated by rules and avoids the use of hard-coded instructions. However, our ability to do this is constrained by our knowledge of the organism's development and how events are triggered, or by our ability to find a pattern in the way events occur. A few properties, such as the timing of cell divisions and the volume distribution among daughter cells, did not follow any visible pattern, so we developed an events queue that holds the information about when and how these should occur. Gene expression and protein inheritance to daughter cells were dictated by rules that simulated known protein functions (Warden, 2014).

The two main goals of the project this year were to improve the accuracy of cell shape and movement in the simulation. During early embryogenesis, the overall shape of the organism is roughly ellipsoidal, with the cells housed in a shell that confines them to this shape. The cells are space-filling, and total volume is preserved as they divide. This results in many inter-cellular forces as the cells are all pushing against one another inside the confined space. These forces cause the cells to take on complex shapes and move in complicated ways dictated by a sum of many forces occurring within the shell at all times. These irregular shapes are challenging to render in computer graphics, but if we can accurately model the forces as they occur in nature, we can theoretically achieve similar behavior in the simulation. This fits in with our principle of defining behavior with a set of rules.

2.1 Metaballs

One major challenge that modelling organic objects poses is the need to render irregular shapes in three dimensions. To address this need, a graphics concept called Metaballs or "blobby objects" was developed by Jim Blinn in 1982 (Blinn, 1982).

Each metaball has a center point and contributes an influence on its surroundings that is strongest at the center point. This influence can be defined by many equations, but a common one is a signed inverse-square equation similar to that of an electric point charge:

$$Influence\ of\ a\ metaball\ at\ point\ p = \frac{q}{(distance\ between\ p\ and\ c)^2}$$

Where c is the center point of the metaball in three dimensions and q is a signed value that determines the "strength" of the metaball.

A field of values is generated by summing the contribution of each metaball's influence for each point in space. A threshold value is chosen such that locations with field value above this threshold are considered to be within the metaball (Figure 1).

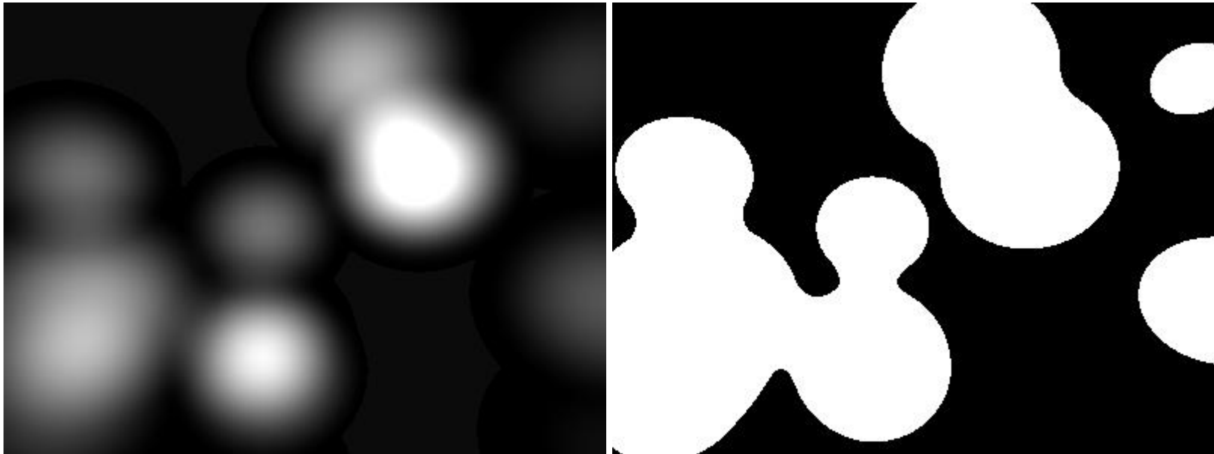

Figure 1: Metaballs. The left image uses a variety of gray shades to indicate the strength of the field at each point. The right image specifies a threshold and divides the space into points considered to be within metaballs (white) and points falling outside of metaballs (black) (Geiss, 2000).

In two dimensions, the image can then be rendered by calculating the field value at each pixel of the screen and assigning a different color to those above the threshold, but in three dimensions, rendering is a more complicated problem that will be addressed in section X.

Because metaball strength is a signed value, metaballs can be thought to have positive or negative "charge" and the way two metaballs interact varies based on whether the two metaballs have the same or differing signs (Figure 2).

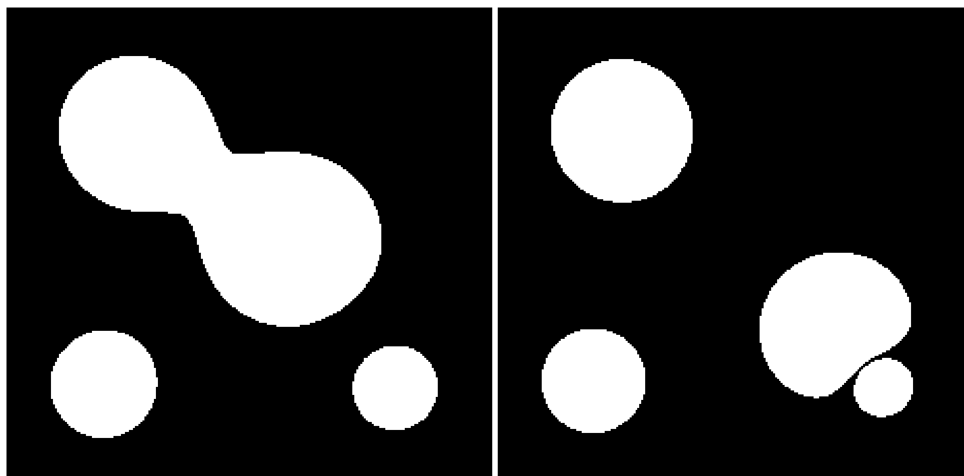

Figure 2: Interactions between metaballs. Interactions between same-signed metaballs (left) create images that are visually similar to cells undergoing division, which could prove useful in a biology simulation. Interactions between differently-signed metaballs (right) give the visual impression of two deformable shapes pressed against each other, which is how individual cells look within the *C. elegans* shell most of the time, since they can change shape to fill the shell.

2.2 Rendering methods

Rendering metaballs in three dimensions is a much more challenging task than in two dimensions. In two dimensions, the field of values translates literally to the screen; each pixel is its own position in both model space and screen space, so the value at each pixel can be calculated in the model and the pixel can be altered as needed.

In three dimensions, the image on the screen is a projection of the three-dimensional model space. Each pixel on the screen correlates to a large number of points in model space, and only the one that is closest to the camera and is above the threshold should determine the pixel color at that location.

There exist multiple strategies to solve this problem, and the two that were most heavily considered for this project are discussed next.

2.2.1 Ray Tracing

Ray tracing attempts to directly solve the problem of having multiple points "under" any one pixel by mathematically constructing a ray that travels through all of those points (Rademacher, 1997). In camera space, this ray has fixed x and y coordinates, starts at the z coordinate of the near plane, and ends at the z coordinate of the far plane. The near and far planes constrain locations in which drawing can occur in three-dimensional graphics, so we know we will not find any objects to render beyond these planes.

One can then mathematically solve for intersection between the ray and the metaball surface to find points at this x-y location where metaballs exist (Terzopoulos, 1987). The one with the largest z coordinate will be the closest to the camera, so the color of the object at this location will be rendered. If no intersection points are found, then there are no objects anywhere at this x-y location. Ray tracing thus allows us to handle three dimensional rendering in screen space just like in two dimensional rendering.

If intersection is difficult to find mathematically, a variation called "ray marching" can be used in which we iterate over the ray in discrete steps and calculate the field value at each place (Geiss, 2000). The first one that is found to be above the threshold should then determine the pixel color.

2.2.2 Marching Cubes

Marching cubes is a rendering method that works in model space. The algorithm was developed in 1987 by William Lorensen and Harvey Cline. The idea is to divide the three dimensional space into a grid of cubes. The program then checks the field value at each of the eight vertices of the cube to determine whether the field is above or below the threshold at each point. There are 256 combinations that can occur, with a unique shape to render in each situation (Lorensen, 1987). These shapes interconnect from cube to cube to form a surface that approximates the true shape of the metaball surface.

2.3 Picking

Object picking, or the ability to click on a cell to highlight it and learn more about it, was a feature that we wanted to have in the simulation. However, this is a more complex problem than it initially appears to be, because any one location in the two-dimensional space of the screen corresponds to many locations in three-dimensional space. One must first discover which objects are located at the mouse position, and then determine which of these is closest to the camera. There are several ways to do this. Ray tracing, discussed earlier as a rendering method, can also be used to do object picking. Another method that was found and ultimately implemented is discussed next.

2.3.1 Color Buffer

The color buffer method involves assigning each pickable object a unique color that isn't necessarily the same as its usual display color. This color is stored in memory in association with the object's identity. In our case, pickable objects are individual cells. When a mouse click occurs, the scene is briefly rendered with the unique colors instead of the usual colors. We then check the color of the pixel that was just clicked and can match it up, using the unique color key, with the cell it belongs to (Clavaud, 2013). Then it is possible to run code that highlights that cell and prints information about it. This is guaranteed to select the cell that the user desired, because it finds the cell that is visible on the screen at that location.

3 Design

There is a big difference between understanding the theory behind a computer science concept and actually implementing it in code. It is nearly impossible to foresee every possible snag and plan in advance. For that reason, among others, most programmers discourage doing a “big design upfront” and instead promote AGILE methodologies, in which some design occurs, followed by some implementation, and then the design is reevaluated using the knowledge gained from the experience (Beck, 2001). Oftentimes that new knowledge comes in the form of an unforeseen complication; other times, it might be that the run time of an algorithm could not be known until it was implemented, and it proves to be too slow. Below is a discussion of ideas that were considered, tried, and frequently tossed in favor of alternate routes.

3.1 Unexpected complications

When first studied at a high level, it seemed metaballs were the ideal way to model the cell shapes and forces. Once metaballs were well understood, including the differing visual behavior between same-signed metaball interactions and differently-signed ones, as discussed earlier, a problem soon became apparent. This problem was, fortunately, able to be solved after some time, so metaballs were able to be used.

As discussed earlier, two interacting metaballs only take on the appearance of pushing against one another when they have opposite signs (positive and negative). However, the project required that many metaballs all repel one another. This would seem to imply that each metaball would have to have a different sign from each other metaball, but with only two sign options available, this is impossible.

A first instinct was to approach it as a constraint satisfaction problem: simply never allow two like-signed metaballs to be adjacent to one another. However, this would pose a huge constraint on the model. No sets of three or more mutually-adjacent metaballs could ever be allowed to exist, and this is a configuration that occurs in the real organism all the time.

A discussion with Worcester Polytechnic Institute’s Sonia Chernova yielded a solution to this problem. Professor Chernova suggested temporarily treating one metaball as positive and all others as negative (Chernova, 2014). The values of the field that are above the specified threshold are then stored; these will surround just the positive metaball. After completing the algorithm iteratively over every metaball, the resultant field will show each metaball as though it is the only positive one in the scene. The visual result is exactly what we wanted (Figure 3). Having to iterate over every metaball to calculate the field slows the computation significantly, but in the final product, this effect is minimized by only iterating over space in the vicinity of the chosen metaball.

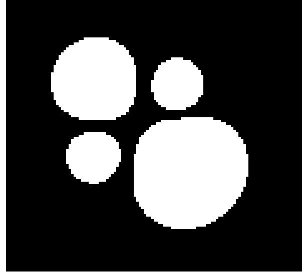

Figure 3: Four mutually-repulsive metaballs. The algorithm suggested by Professor Chernova yields the intended visual: each cell seems to push against each other cell.

3.2 Comparison of Metaball rendering methods

We knew from the start that there were multiple ways to render metaballs in three dimensions. We started off by considering ray tracing, as it was thought to be the faster method. However, some disheartening discoveries were made as soon as it was explored in more detail, and it soon fell out of favor. Another algorithm called Marching cubes was used in the end.

3.2.1 Ray Tracing

The mathematics behind ray tracing vary based on the shapes with which the rays are colliding. This alone makes it a less favorable solution, as new code needs to be written any time we want to depict a new kind of shape. The idea is to represent the ray parametrically and then substitute it into the equation that defines the shapes in the model (Rademacher, 1997). Then one can solve for time to locate the position along the path of the ray at which the intersection occurs.

A point along a ray can be represented parametrically as:

$$Point(x, y, z) = \begin{cases} \text{ray origin. x } + \text{ ray normal. x } * \text{ t} \\ \text{ray origin. y } + \text{ ray normal. y } * \text{ t} \\ \text{ray origin. z } + \text{ ray normal. z } * \text{ t} \end{cases}$$

This formula must be substituted into a formula representing your shape (Terzopoulos, 1987). Then, if it is possible to solve for t and obtain at least one positive real value, the shape and the ray intersect. This t can then be substituted back into the parametric ray equation to get the location of the point.

A point is on the surface of the metaball field if it satisfies the following equation:

$$threshold = \frac{charge1}{(center1.x - point.x)^2 + (center1.y - point.y)^2 + (center1.z - point.z)^2} + \cdots + \frac{charge\ n}{(center\ n.x - point.x)^2 + (center\ n.y - point.y)^2 + (center\ n.z - point.z)^2}$$

Where there is a term for each of the n metaballs present in the field.

However, solving for t becomes very complex when we substitute in the parametric ray equation:

$$threshold = \frac{charge1}{\left(center1.x-(rayOrigin.x+rayNormal.x*t)\right)^2+\left(center1.y-(rayOrigin.y+rayNormal.y*t)\right)^2+\left(center1.z-(rayOrigin.z+rayNormal.z*t)\right)^2} + \cdots + \frac{charge\ n}{(center\ n.x-(rayOrigin.x+rayNormal.x*t))^2+(center\ n.y-(rayOrigin.y+rayNormal.y*t))^2+(center\ n.z-(rayOrigin.z+rayNormal.z*t))^2}$$

For a scene containing only two metaballs, the equation for time reduces to a fourth order polynomial, this equation only getting more complex as additional metaballs are added. These polynomials must be solved numerically, which is a slow process.

Because of the complexity of the math, this is a situation in which we would instead use ray marching (Geiss, 2000). Ray marching, as described earlier, adds another loop to each pixel's calculations by performing calculations over discrete steps along the ray. This makes ray tracing a much slower process.

Furthermore, ray tracing has other downsides that do not apply to its competitor, marching cubes. Ray tracing requires additional calculations to reinstate lighting effects into the scene, since the pixel array is being altered directly. Also because it works in screen space, the calculations need to be redone every time the image on the screen changes, such as when camera rotations occur.

At this point, we switched gears to focus on marching cubes instead.

3.2.2 Marching Cubes

Marching cubes was soon decided upon as a good alternative to ray tracing. Because it constructs shapes using Processing polygons, automatic lighting effects provided and optimized by the Processing language can continue to be used. Because it works in model space, the calculations must only be re-performed every time the actual model changes, so camera rotations can occur without repeating calculations (Lorensen, 1987).

3.3 Cell Motion

Besides cell shape, the other focus of this project was obtaining accurate cell motion. The goal was to get the cells to move to their real locations within the shell without hard-coding the known behavior. We hoped to achieve this by modelling natural pushing forces, both between cells and between cells and the shell wall.

3.3.1 Local search

Since objects in real life take the path of least resistance at any given moment with no "knowledge" on where the lowest-energy global locations are, that is how the cells behave in Simworm as well. At each time step, each cell object looks at each of its adjacent locations and calculates the pushing forces at that place, compared to its current location. It then moves to the location where forces are found to be the lowest.

One interesting thing we found was that, near the beginning of the simulation, the organism is still very symmetrical on the D-V and L-R axes. When the third cell stage is reached, there is not one optimal direction for the third cell to move. Up and down are equally favorable, and the direction that it “chooses” depends solely on which one is considered first in the code. We ordered the lines of code such that the cell would go dorsal, as it does in the real organism, but this felt dishonest. After some research, however, we found that the dorsal-ventral axis is defined based on which direction the third cell moves (Gönczy, 2005). The dorsal direction is defined to be the direction that the third cell moves. So, what was initially thought to be an oversimplification in the simulation turned out to be completely consistent with the biology.

3.4 Change of equation

After implementing all the metaballs graphics, the resulting visual was assessed in terms of biological accuracy. It had two major problems. The cells were not adequately filling the space of the shell as they would in nature; as more cell divisions occurred, more empty space appeared between the cells. In nature, of course, volume is preserved when cells divide. The cells also weren’t moving to the correct locations.

Metaballs do not necessarily preserve volume exactly; as two oppositely signed metaballs approach each other, they tend to shrink. This effect is minimized by the metaballs’ movement pattern: they are programmed to minimize intercellular forces, and this preserves volume as much as possible. However, with the inverse-square equation that we were using to define our metaball field, there were still significant forces existing in the space between metaballs.

For this reason, the metaball equation was soon changed to an inverse-fourth-order one instead, as shown below.

$$Influence\ of\ a\ metaball\ at\ point\ p = \frac{q}{(distance\ between\ p\ and\ c)^4}$$

This resulted in better space filling and cell positioning. It is easy to understand why this is more realistic. In the real world, two objects that are not touching exert basically no force on one another. Forces only begin to exist when they actually come into contact. By using an equation that drops off very quickly as distance increases, this effect is more accurately modelled.

4 Implementation

Once the code had been implemented as planned, there was some analysis to be done. As with any computer science program, efficiency of algorithms was a major goal. Because many of these graphics concepts, particularly Marching Cubes, were very computationally-intense, performance was an especially large concern. And, of course, analysis of the simulation's output compared to real biological data was an important step in evaluating the success of the project outcome.

4.1 Performance

One major concern with the simulation was the performance of the chosen rendering method, marching cubes. An analysis was performed of the render time against two variable parameters: the number of metaballs present in the scene and the granularity of the image. It was found that the algorithm scales fairly well as the number of metaballs in the scene increases (Figure 4), but image granularity has a much more significant impact on rendering time (Figure 5). Because this is such a big factor in the speed of the program, the user has an option on screen to sacrifice image quality for runtime speed, or vice-versa, if desired. This makes the simulation useable on more computers; people running the simulation on lower-end computers can probably still get decent frame rates if they user a lower quality image.

It should be noted that these tests measure just the performance of the marching cubes algorithm, not the full simulation. The tests were performed in a separate application that simply renders a number of metaballs using the marching cubes algorithm. This allows us to separate the effects of other computations occurring in the simulation from the computation time of marching cubes. It also allows for testing of the performance of marching cubes past the twenty-six metaball threshold to which the simulation is constrained.

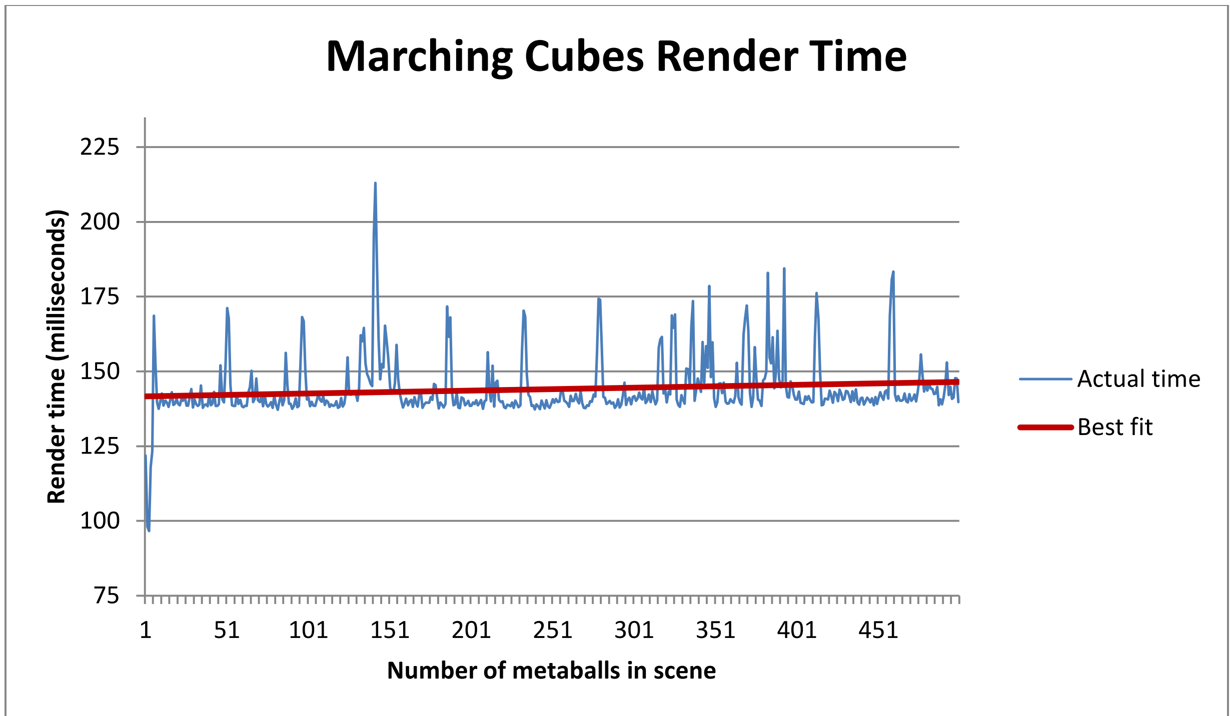

Figure 4: Effect of adding metaballs. The time it takes the scene to render with 1-500 metaballs present. The values formed by the blue line are the actual data; this is the average over 10 runs per number of metaballs with a fixed grid size of 8. The red line is the linear best fit for the data. It is not well understood why exactly the empirical data is so noisy.

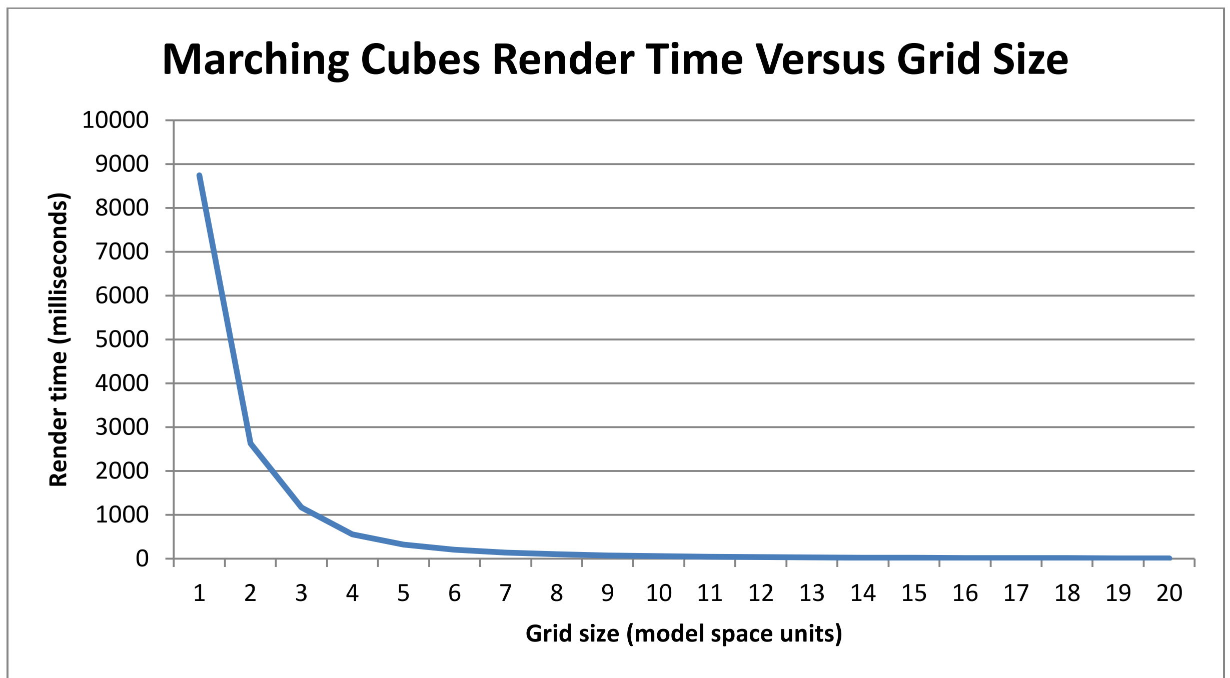

Figure 5: Effect of grid size. The time it takes the scene to render versus the size of the grid used to divide the model spaces into discrete cubes. A higher grid size corresponds to lower image granularity. The data is the average over 10 runs per grid size value with a fixed value of 30 metaballs present in the scene.

4.2 Biological accuracy

Another primary concern with the simulation is the desire for it to calculate and output an image that looks similar to the real *C. elegans* organism. What follow are some images comparing the simulation output to diagrams and images of *C. elegans.*

In an effort to also demonstrate improvement over the simulation at the end of last year, comparison screenshots of last year's output are included. Last year, the focus was on gene expression and protein movement, so the simulation was visually very different and less accurate than it is now (Warden, 2014). The efforts put into improving cell shape and positioning have resulted in a much more accurate visual representation (Figures 6, 7).

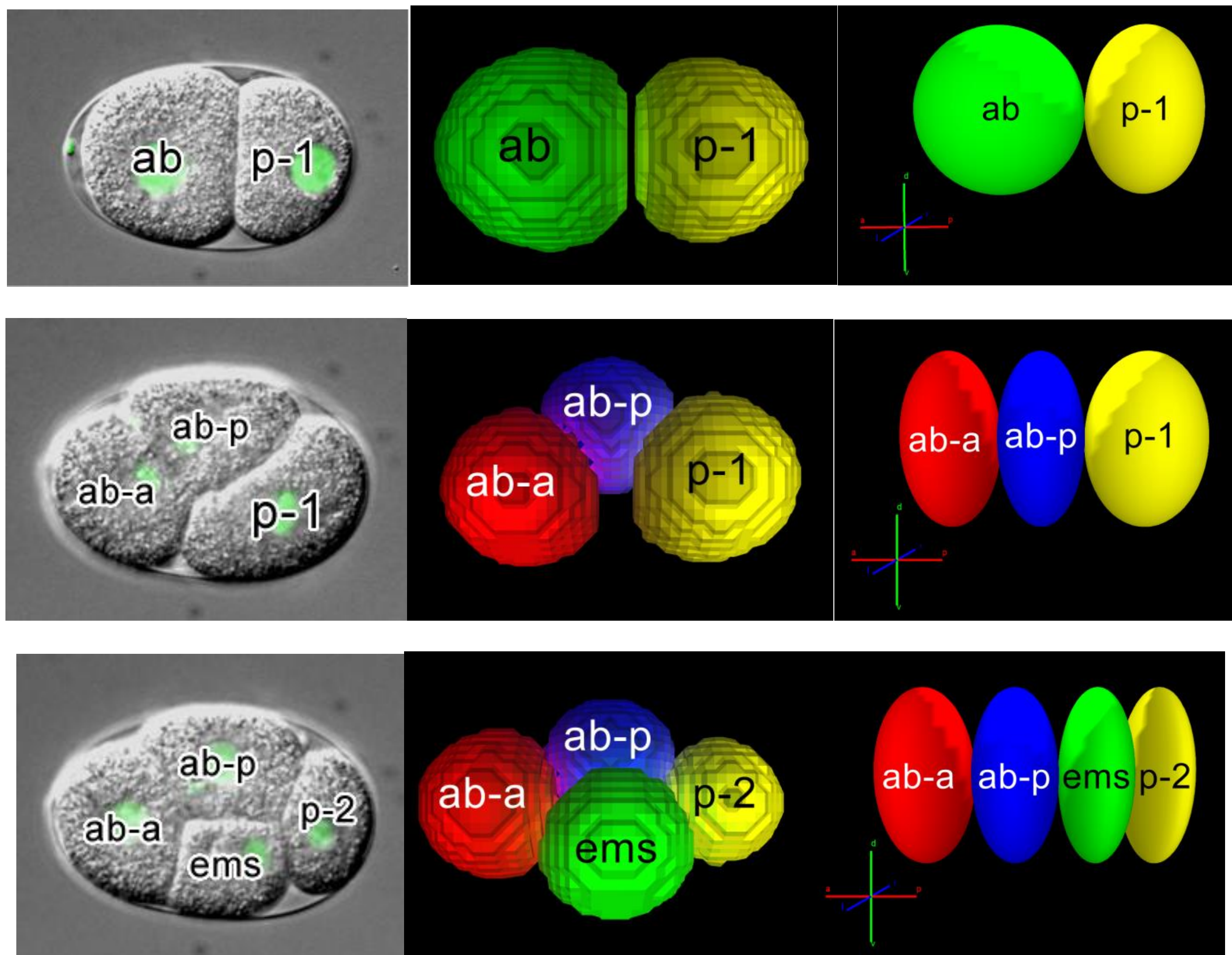

Figure 6: Two, three, and four cell stage. From left to right, the images are of a real organism, this year's simulation at the same developmental stage, and last year's simulation (Gönczy, 2005).

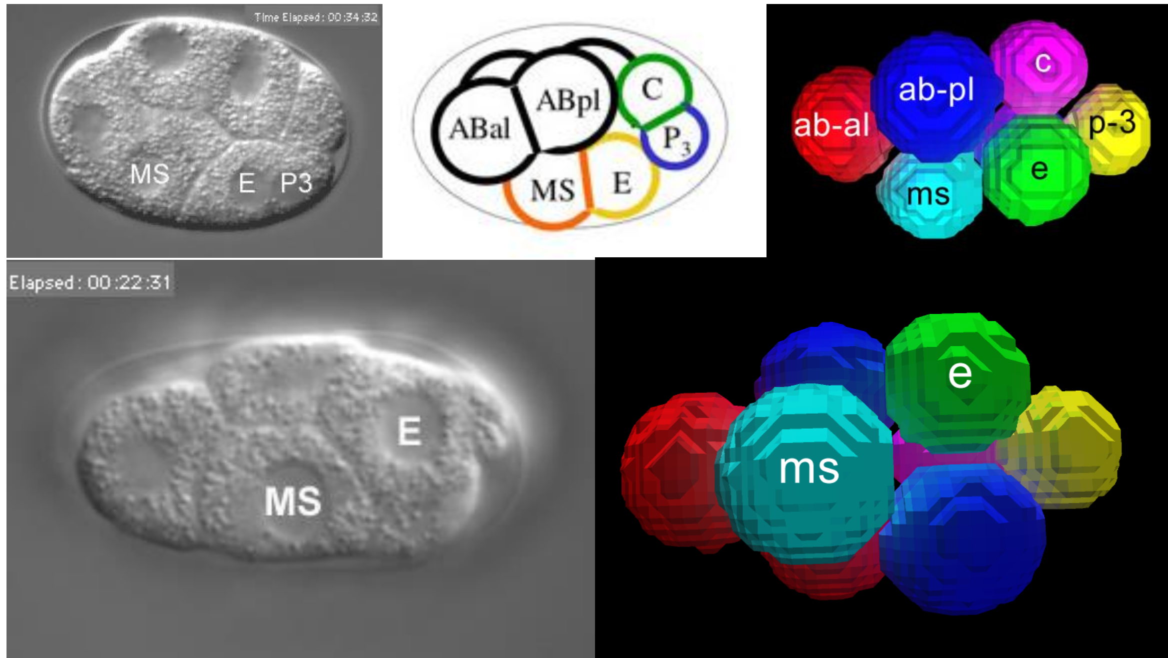

Figure 7: Eight cell stage, lateral and ventral views. The top images compare empirical data to the simulation in the lateral view, the bottom images show the ventral view (Goldstein, 2003). A labelled diagram is also included in the lateral view for clarity (Gönczy, 2005).

5 Conclusions and Future Work

This year's project began modelling the physical forces that shape cells and push them into their true locations, a topic that was barely touched last year. The result is a far more visually-accurate simulation.

However, once an erroneous position has been reached, the likelihood of getting back to an accurate configuration at a later time is very low. More likely, the errors will magnify as time progresses.

The simulation in its current state is very accurate until about the 11 cell stage. At that time, the descendants of cell ab become a little bit skewed on the D-V and L-R axes rather than staying lined up as the diagrams seem to indicate they should. Immediately after the division, these cells are parallel to the A-P axis, but they push each other around and settle into inaccurate positions.

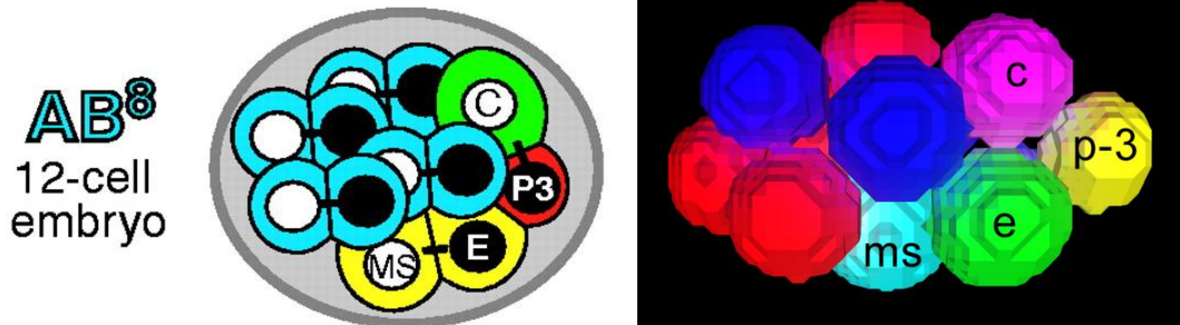

Figure 8: Twelve-cell stage. The cells in blue and red in the simulation are descendants of AB and are beginning to develop tilt on the D-V and L-R axes, whereas the diagram of the real organism shows them being fairly parallel to the A-P axis (Park, 2003). The posterior-side cells that are descendants of p-1 are all still in quite accurate positions.

Thought should be put into how the physical model can be changed to maintain accuracy for longer than the current model is able to. There is also much more future work to consider.

5.1 Accuracy of shapes

At the current time, each cell is rendered with a single metaball. This allows the cell to take on a variety of approximately ellipsoidal shapes. More complex shapes can only be achieved with the addition of more metaballs. This was not a huge problem before the twenty-six cell stage, where most of the cells are approximately spherical, but as development progresses, the organism gains some more elongated cells. Even the early stage would benefit from the inclusion of more metaballs to improve the accuracy of shapes. Thought will have to be put into optimizations, as the simulation is already a little slow with one metaball per cell, and it will get slower when additional metaballs are added.

5.2 Biological factors

Last year's project group focused mainly on achieving accurate gene expression and protein location, but one thing we were forced to put off until a future time was the implementation of inter-cellular

gene expression rules. It would not have made sense to implement them last year, since adjacency of cells was not accurate. Now that more accuracy has been achieved in cell location, we are in a good position to begin implementing inter-cellular communication. However, that was not tackled this year due to a shortage of time.

The twenty-six cell stage was chosen as the end stage of this iteration of the project because this stage marks the threshold between two distinct biological phases in the real organism. At the twenty-six cell stage, a process called gastrulation begins, in which cells start to involute in response to changes in cytoskeletal forces (Goldstein, 2003). These forces are not modelled at all in the current simulation. A new set of physical rules will need to be developed to handle this stage of the organism's growth.

Similarly, as the cells forming the epidermis begin to adhere to each other to form sheets, new forces will need to be modelled to capture this relationship. As the organism begins to truly resemble a worm, the epidermal cells will need to be capable of containing other cells to form a long tube-like shape. Other more complex biological phenomena were not modelled in this simulation. Forces caused by the extracellular matrix are not addressed, for example.

5.3 Improvements to the code

An analysis of the object oriented design of this codebase reveals some concerns. Right now, class cohesion is low. I have only recently learned about the principle that each class should serve a very specific purpose, and that it is always better to create new classes to handle new tasks as they arise (Christensen, 2010). My instinct as a programmer, before learning this, was to reuse classes if they seemed relevant to the new task. For example, the class Coordinate is being used for multiple different "triplet" data situations. It is sometimes used to represent a set of three lengths, such as the length of a cell on each axis, and other times, as its name implies, it is being used to represent a point in three dimensional space. Both of these situations involve sets of three numerical values, which is what the Coordinate data structure contains, but they serve different purposes and should really be abstracted into two different classes. Similar mistakes exist all throughout the code. Knowing what I know now, I wish there were time for extensive refactoring.

5.4 Goals that were not achieved

Last year's paper defined a list of user-interface features that we would like to have in the final simulation (Warden, 2014). Although some of these are very small tasks, they were lower priority than the extensive work that was needed in improving cell shape and location, and there wasn't time to implement them. Many of these could be added to the simulation with minimal work.

- Capability to identify a time point for the simulation to begin at instead of starting at time 1
- Capability of viewing different planes to see "inside" the embryo when cells are covered by other cells
- Label cells according to name based on the lineaging table
- Be able to highlight all cells from one lineage

- Be able to filter out cells – hide them from view
- Capability of adjusting window proportions – having a split screen and then adjusting the proportion of the screen that shows the simulation and the part that tracks data

In addition to these, one improvement that could be made fairly easily would be the depiction of the process of cell division. Currently, the simulation just makes the two daughter cells immediately separate. But as alluded to earlier, metaballs with like charge interact in a way that is visually reminiscent of cell division. Depicting cell division would simply require temporarily treating the dividing cells as having like signs and switching over to differing signs after the distance between them reaches some threshold.

The changes to the simulation this term were done in a very limited timeframe, and there was not time to accomplish as much as desired. However, the visual accuracy of the simulation has improved greatly over last year. Last year, the cell position was dictated only by the axis of division. After division, cells did not move. Because the divisions occurred mostly on the A-P axis, cells also became very flat after a few generations, which is not at all realistic. The shapes were also limited to ellipsoids, which is inaccurate and doesn't properly represent the shapes formed by cell-to-cell interaction. Through the study and implementation of advanced computer graphics concepts, we have achieved a more accurate visual representation, which will be much more useful to biologists utilizing the model for both educational and research purposes.

6 References

Beck, K. (2001, February 1). Manifesto for Agile Software Development. Retrieved October 18, 2014 from http://agilemanifesto.org/.

Blinn, J. (1982). A Generalization of Algebraic Surface Drawing. *SIGGRAPH*, 1(3), 235-256. Retrieved October 13, 2014, from ACM Digital Library.

Chernova, S. (2014, September 22). Personal interview.

Christensen, H. (2010). Coupling and Cohesion. In *Flexible, Reliable Software* (pp. 157-159). Boca Raton: Taylor and Francis Group, LLC.

Clavaud, Nicolas (2013). Picking. Retrieved March 13, 2014, from http://n.clavaud.free.fr/processing/library/picking/.

Fry, B., & Reas, C. (2014). Processing. Retrieved January 14, 2014 from https://www.processing.org/.

Geiss, R. (2000, March 10). Metaballs (also known as: Blobs). Retrieved September 1, 2014, from http://www.geisswerks.com/ryan/BLOBS/blobs.html.

Goldstein, B. (2003, January 1). Gastrulation in C. elegans. Retrieved October 12, 2014, from http://labs.bio.unc.edu/Goldstein/gastr.html.

Gönczy, P. and Rose, L.S. Asymmetric cell division and axis formation in the embryo (October 15, 2005), *WormBook*, ed. The *C. elegans* Research Community, WormBook, doi/10.1895/wormbook.1.30.1, http://www.wormbook.org.

Lorensen, W., & Cline, H. (1987). Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH*, 21(4), 163-169. Retrieved September 4, 2014, from ACM Digital Library.

Park, Frederick., & Priess, J. (2003). Establishment of POP-1 asymmetry in early C. elegans embryos. *Development, 130*, 3547-3556. Retrieved October 10, 2014, from http://dev.biologists.org/content/130/15/3547.figures-only.

Rademacher, P. (1997). Ray Tracing: Graphics for the Masses. *ACM Crossroads.* Retrieved September 7, 2014, from https://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html.

Terzopoulos, D., Platt, J., Barr, A., & Fleischer, K. (1987). Elastically Deformable Models. *SIGGRAPH*, 21(4), 205-214. Retrieved September 9, 2014, from ACM Digital Library.

Warden, R., Thayer, S., & Wigell, R. (2014). *Simulation of Early C. elegans Embryogenesis* (Undergraduate Major Qualifying Project No. E-project-043014-213618). Retrieved from Worcester Polytechnic Institute Electronic Projects Collection: https://www.wpi.edu/Pubs/E-project/Available/E-project-043014-213618.

7 Appendices

7.1 User's Guide

Running the simulation

Extract the files in this .zip folder and double click Simworm14.jar to run it. Be sure to keep the jar, genes.csv, wormbaseGeneInfo.txt, antecedentsAndConsequents.csv, and eventsQueue.csv in the same directory together or the simulation will not work as intended.

If this doesn't work, you may not have Java installed. Visit oracle's website and download Java: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Explanation of what this folder contains

genes.csv: The list of genes tracked by the simulation. This is currently populated with a small subset of real C. elegans genes; those whose behavior is best understood currently.

antecedentsAndConsequents.csv: The list of antecedent and consequent rules obeyed by the simulation. This is currently populated with a small subset of the rules that are most thoroughly studied and understood.

eventsQueue.csv: The times and properties of the cell divisions depicted by the simulation. Currently, the simulation only goes up to the 26 cell stage.

wormbaseGeneInfo.txt: A resource obtained from wormbase that allows for error-checking on gene names (only valid C. elegans genes will be accepted in genes.csv).

Altering the csv files (optional)

The csv files can be altered to run the simulation with a different set of genes or antecedent and consequent rules. However, one must be *very carefuL* when touching these files, as they are read by the program in a very specific way and a single typo can result in an un-runnable simulation. Here is the format of each of three csv files:

Events Queue:

Each row is a new event. The columns, from left to right, represent:

- Name of the cell that is dividing
- Percentage of the volume that goes to daughter1 (daughter1 is always the more anterior, dorsal, or right child)
- Axis along which the division takes place (choices are X, Y, Z - capitalization matters!)
- The time (simulation time) of the division

Antecedents and Consequences:

Each row is a new rule. The columns, from left to right, represent:

- Name of the *consequent* gene
- State that the consequent gene will be set to (choices are A and I - capitalization matters - meaning active or inactive)
- Start time of the rule
- End time of the rule - put 0 if the rule should never end
- Name of an *antecedent* gene
- State that the antecent gene must be set to in order for a consequence to occur
- Name of another antecedent gene, if present
- State for that gene...
- ...etc (can repeat for as many antecedents as necessary, so some lines will be longer than others - this is okay)

Genes:

Each row is a new gene. The columns, from left to right, represent:

- Name of the gene (only lower-case letters, hyphens, and digits are accepted, and this will be checked against the wormbase file - only valid C. elegans genes will be accpted)
- Whether its state is initially active, inactive, or unknown (written as A, I, or U)
- Compartment on the x axis (choices are center, anterior, or posterior - be sure to use lower case)
- Compartment on the y axis (choices are center, dorsal, or ventral)
- Compartment on the z axis (choices are center, left, or right)

Some genes end there. Others have 4 more entries. These are genes that switch compartments at some point. For example, par-6 has to switch from anterior to center after the first division (that is, within cell ab). These optional 4 entries are:

- New compartment on the x axis
- New compartment on the y axis
- New compartment on the z axis
- Name of the cell in which the change takes place

If a gene switches more than once, an additional 4 entries can be made containing the same information for the second switch.

7.2 Developer's Guide

Set up in Eclipse:

Simworm 2014 was developed in Eclipse IDE. Because Eclipse can be a little confusing to use, here are the basics for importing Simworm code to be extended upon.

1. **Download code from GitHub**

Go to https://github.com/rachelwigell/Simworm-2014 and click Download ZIP in the lower right hand corner. Extract the files from the zip folder.

2. **Download Eclipse**

Go to https://www.eclipse.org/downloads/packages/release/Kepler/SR2 and find the download link associated with your operating system for Eclipse Standard 4.3.2. After downloading eclipse, you should be able to click the eclipse.exe file to run eclipse (no installation necessary)

3. **Import code into Eclipse**

When you open Eclipse, you'll be asked to choose a workspace. Give it a new folder; this is where your work will be saved. Then when you're presented with the main screen (exit out of any welcome screens shown), go to File → Import → General → Existing Projects into Workspace. Hit Next. Next to where it says "Select root directory" click the Browse button. Navigate to the folder where the Simworm code was extracted. Click through it until you reach the folder called Simworm. Inside this folder, hit OK. In the box below the text that says "Projects," an option should have shown up that says Simworm. Now you can click Finish.

Everything should import in a runnable state. You are ready to alter code.

Explanation of the code:

The meat of this document will attempt to thoroughly explain the code base. I will start by explaining what each package and file does, and will go on to talk through any particularly confusing functions.

What's in each package:

Components: The simulation depends on a few CSV and TXT files to run. It reads the information contained in these files to determine some of the organism's properties, such as what genes it contains. These files are contained in the components package. One text file called Read Me.txt is not necessary for running, but provides a lot of useful instructions to people who wish to run the program. This package also contains the executable JAR, which is what is distributed to users of the simulation to easily run it. This JAR should be updated any time a significant change is made to the code base (instructions on how to do so follow in a later section).

Docs: This folder contains a zip folder with the javadocs for the code. Javadocs are a convenient way to look through the comments I have written in the code without having to skim the actual code for them. These will need to be updated periodically as new methods and comments are written. Instructions on exporting Javadocs are in another section below.

Test: This package contains my testing code for SImworm. The tests can be run with JUnit to ensure that the tested features are working properly.

Processing: This folder contains classes relevant to the visualization of the simulation. This includes the main visual class, BasicVisual, which extends the Processing class called PApplet. This allows Processing visuals to be used with Java code. This also includes classes relevant to the Marching Cubes algorithm, Cube and Metaball.

Data Structures: This package contains most of the meat of the code base; the various classes that are used to represent concepts within the simulation. Among the most important classes are Shell, Cell, Gene, and Coordinate.

Libraries:

The simulation uses several external libraries to run. These can be found by expanding the menu "Referenced Libraries" in the package explorer in Eclipse. They should be added to the build path automatically, so no action is required. However, I would like to explain what each one is and what it does.

Processing: The most significant library used is Processing. This is what allows all of the visuals to exist. The library file core.jar allows access to all Processing code and thus must be imported into any class that hopes to use Processing methods. These classes must also extend PApplet. All of the jar files with prefix jogl and gluegen are also Processing libraries. These are not used in code, but are necessary for export. They allow the executable to run on different operating systems.

Peasycam: Peasycam is a very convenient and user-friendly camera for use with Processing. It allows us to rotate 360 degrees, pan, and zoom, all with minimal set up. This library file is called peasycam.jar.

ControlP5: The ControlP5 library contains all elements used to obtain user input: buttons, sliders, check boxes, etc. The menu on the right-hand side of the main simulation is entirely constructed from ControlP5 components. This jar file is called controlp5.jar.

It's worth noting here the way that ControlP5 elements work. A button, for example, can be declared with new Button(controlp5, String). The first argument provided is the ControlP5 object that the button belongs to. The second argument is the name for that button. That name will not only be written upon that button, but will be used to define action listeners for the button. An action listener is a method describing what should happen when the button is clicked. So if you declare a button and name it "button1," in order to define what happens when button1 is clicked, you must make a separate method called button1. This method header should be in this format: void button1(float theValue).

Events:

Processing is an events-based language, meaning that the program has two main subsections: set up, and draw. Set up occurs once at the beginning of the program. After set up, draw occurs

repeatedly until an event occurs. An event is something like a mouse click or a keyboard press. Below, I will summarize the events that occur in set up, draw, and each type of event.

Set up: Simworm actually has two set up functions: one that occurs at the very beginning of the program, which presents the menu that allows users to choose mutants, and one that occurs after the menu selection has been made, which sets up the main screen of the simulation.

Primary set up: This function is called setup() and can be found in BasicVisual. Processing automatically calls this function at the start of runtime. It defines the PApplet, which is the main Processing visual, using the size() function. When calling size(), we automatically detect the screen resolution of the running computer, such that the application will be full-screen on any computer. Primary set up also initializes the camera (Peasycam) with the Peasycam constructor. The ControlP5 menus are also initialized with the ControlP5 constructor. Then our first ControlP5 menu is built; the one that allows the user to select which pars they want to mutate. This is composed of a non-interactive label, a series of check boxes each corresponding to a par protein, and a button that says “Create shell.” All elements’ sizes and positions are defined as a percentage of the screen size such that elements will be aligned and fit on any screen resolution. When the user clicks create shell, their choice of check boxes is stored into a HashMap called mutants and secondary set up is called to draw the main simulation screen.

Secondary set up: This function is called secondarySetup() and can be found in the BasicVisual class. It first removes the ControlP5 elements for the mutant choosing menu. It then creates the initial shell by calling the Shell constructor. It initializes a couple of HashMaps that are used to allow backwards time steps to take place. These will be explained in more detail later. It calls the marching cubes algorithm to display the shell. Then it does an immense amount of initialization to create the ControlP5 menu that inhabits the right-hand side of the screen for the rest of the simulation.

Draw: The draw loop is a method called draw() in BasicVisual. This is called automatically by Processing in a tight loop. Until secondary set up occurs, the draw loop does almost nothing. After the user hits the “create shell” button to call secondary set up, a Boolean value called mutantsChosen is set to true. This allows the events in the main drawing loop to begin occurring.

In the main drawing loop, many things occur. The coordinate axes are drawn to the screen. If shell depiction is enabled, the shell is drawn to the screen. The text in the box on the right hand side of the screen is updated, in case the user typed anything recently. We check for a recent change in color mode and update the colors of the cells if one occurred. If the cells are currently undergoing an animation, we progress the animation forward by one frame. And if the simulation is set to automatic time flow mode, we call the progressForward() function periodically.

Mouse input: The mouse is used to receive many kinds of user input, so the mouse click function has to check for many different situations. All mouse input is handled in the BasicVisual class.

A mouse drag can be used for multiple reasons depending upon where it occurred. In the left part of the screen, a mouse drag rotates the Peasycam. This is automatically handled by the Peasycam

library, so you won't see explicit code to handle this situation. However, you will note that in the draw() method, the camera is set to be inactive when the mouse is on the right fifth of the screen, where the menu is located. This prevents the camera from being rotated when the user is adjusting a slider bar in the menu, for example.

This leads me to the other use of a mouse drag: setting the slider bar that adjusts image granularity. The mouseReleased() function is a default Processing function whose body can be filled in. The given actions will then automatically occur any time the user releases a mouse button. In this function, we check to see if the slider bar value has changed from what it was last time we looked at it. If so, we reset the grid size used by the marching cubes algorithm, then call marching cubes to update the display to reflect the change.

A mouse click can cause many things to happen too. If the user clicks on any ControlP5 element, its action listener will automatically be called. This is handled automatically by ControlP5, so you will not see any code for detecting whether the mouse is over a button and then calling the action listener.

The one mouse click operation that did have to be handled manually was object picking. The simulation has the ability to detect when a cell is clicked and to highlight that cell. Much like the mouseReleased() function, mouseClicked() is automatically called by Processing when the mouse is clicked. This function ignores the right fifth of the screen where the menu is, since pickable objects only exist on the left side. When a click occurs on the left, picking code is called. I will explain how the object picking algorithm works later.

Keyboard input: Keyboard input is also used extensively by the animation. The user can type cell names to select them, and the left and right arrow keys control time flow. The keyReleased() function in BasicVisual is called automatically by Processing any time a keyboard key is released, and handles all keyboard input.

When the Esc key is pressed, the simulation exits.

When the right arrow key is pressed, the code for progressing forward by one time step is called. This algorithm will be explained later.

Similarly, when the left arrow key is pressed, the code for progressing backwards by one time step is called. I will also explain this algorithm later.

When the user presses a character key, the character gets added to the text that appears in the text box in the right hand menu. If this string gets long, it will automatically be overwritten. This is to prevent the user from ever having to erase a huge amount of text, because there aren't any valid commands that are long anyway. When the user presses backspace, the string will be shortened by one character.

When the user presses enter, the text currently located in the menu is entered as a command. Valid commands are names of cells that are present in the simulation at the current time. If the user presses enter and a command isn't valid, the text changes to give an error message. If the command was

valid, the cell whose name the user wrote will be highlighted, and the text in the box will change to display the list of genes located in that cell.

Complicated Algorithms:

I now want to explain the most complex algorithms in the simulation. These are also extensively commented within the code, but they might be difficult to follow without some explanation.

File parsing: The simulation has the ability to parse CSV and TXT files for information. This was implemented to allow non-techy people to easily change the inputs into the simulation to reflect different biological situations. However, this means the simulation has to do somewhat complicated parsing at the start of each run. The following sets of data are obtained from file parsing: the events queue, the set of genes given to the first cell, and the set of antecedent and consequent rules.

Each of the parsers works in the same basic way. They use a Java-provided class called BufferedReader. BufferedReaders are declared by passing in a FileReader object, which in turn is declared by passing in a string containing the file location. So a BufferedReader would be declared like so: new BufferedReader(new FileReader("insert file name here")). BufferedReaders give us the convenient ability to divide the file input by line simply by calling readLine(). The resulting string can then be tokenized with split(). This means that you give it as input the character that separates individual items (in our case, commas), and it returns an array of strings where each string is one "word." We can then go through the array and assign each "word" to the variable it represents. We'll get more specific when talking about each individual CSV below.

The strings are error checked as we go along such that the user cannot enter nonsensical information into the simulation without invoking an error. For example, some fields are expecting a number, so if the user passes in a word, for example, the program will not accept it. The program will also generate a helpful error message. The enumeration class called FormatProblem describes the different types of errors that can occur with reading input. The exception class InvalidFormatException takes in the type of FormatProblem that occurred and generates a useful error message based on that information.

The file name you must provide changes depending on what type of run we are trying to do. If we are trying to run the simulation as a Java application or a JUnit test, we have to give the file name relative to the project directory; that is, "src/components/filename.csv". If we're running as a Java applet or an executable, the file name given is relative to the directory where the executable is stored, meaning that just the filename alone can be given: "filename.csv". The program will first attempt to find the file using the first format, and if it fails, will try with the second format. This ensures that the program will find the files successfully no matter what type of run you're doing.

Events Queue: The events queue is a series of steps describing when cell divisions occur and some properties about each cell division. In the file eventsQueue.csv, this information is stored with each row representing a separate event and each column, from left to right, representing:

- Name of the cell that is dividing

- Percentage of the volume that goes to daughter1 (daughter1 is always the more anterior, dorsal, or right child)
- Axis along which the division takes place (choices are X, Y, Z - capitalization matters!)
- The time (simulation time) of the division
- What generation the dividing cell belongs to

Parsing occurs in the Shell class in the method called readEventsQueue(). The DivisionData class exists to hold this type of information. It has a field for each of the bulleted pieces of information, so each division in the events queue can be represented as a single DivisionData object. In the Shell class, we have a HashMap of strings to DivisionData objects called divisions. The string is the name of the dividing cell, and serves as the key for the HashMap. This means that the DivisionData for a cell can be accessed by calling divisions.get("insert name of cell here"). Each Cell object also has a field of type DivisionData which stores this information too.

The file eventsQueue.csv is tokenized as described above. Then the first element of the result array can be stored as DivisionData.parent, the second as DivisionData.d1Percentage, and so on. These are error-checked as described above.

Genes: The genes.csv file is parsed when the first cell is being created. It is populated with the given genes and their properties. Parsing occurs in the Cell class in the method called readGeneInfo(). Each row of the CSV is a gene, with each column, left to right, representing:

- Name of the gene (only lower-case letters, hyphens, and digits are accepted, and this will be checked against the wormbase file - only valid C. elegans genes will be accpted)
- Whether its state is initially active, inactive, or unknown (written as A, I, or U)
- Compartment on the x axis (choices are center, anterior, or posterior - be sure to use lower case)
- Compartment on the y axis (choices are center, dorsal, or ventral)
- Compartment on the z axis (choices are center, left, or right)

Some genes' rows end there. Others have 4 more entries. These are genes that switch compartments at some point. For example, par-6 has to switch from anterior to center after the first division (that is, within cell ab). These optional 4 entries are:

- New compartment on the x axis
- New compartment on the y axis
- New compartment on the z axis
- Name of the cell in which the change takes place

If a gene switches more than once, an additional 4 entries can be made containing the same information for the second switch.

The Gene class contains fields for each of these pieces of information, so as the sheet is parsed, the information is stored in a new Gene object. The total list of genes is then given to the cell p-0 upon its creation. The data is error-checked as it is read, as described above.

Gene names are also checked against the text file wormbaseGeneInfo.txt, which is a list of valid *C. elegans* genes obtained from wormbase.org. This requires that the TXT file be parsed first and stored. This occurs in the method compileValidGenes() in the Shell class. A HashMap called validGenes exists in the Shell class to store this information. Then a gene name provided in genes.csv will only be accepted if it exists inside of validGenes.

Antecendent and Consequent Rules: Antecentdent and Consequent rules are a concept from biology. The idea is that if a certain set of antecedents are fulfilled, a consequence will occur. This occurs in genes, where the antecedents involve a certain set of genes having particular active or inactive states, and the consequence is that another gene is switched to become active or inactive.

In antecedentsAndConsequents.csv, each row is a new rule, and each column, from left to right, represents:

- Name of the *consequent* gene
- State that the consequent gene will be set to (choices are A and I - capitalization matters - meaning active or inactive)
- Start time of the rule
- End time of the rule - put 0 if the rule should never end
- Name of an *antecedent* gene
- State that the antecedent gene must be set to in order for a consequence to occur
- Name of another antecedent gene, if present
- State for that gene...
- ...etc (can repeat for as many antecedents as necessary, so some lines will be longer than others - this is okay)

The Consequence class has fields for storing all this information, so each row can be represented by one Consequence object. The ConsequentsList class then has a couple of lists of Consequence objects, which hold all of the rules from the CSV file. One of these lists is the set of rules that are currently active, called antecedentsAndConsequences, and the other is a list of rules that will become active at some point in the future, called startLate. As rules become active, they will be removed from startLate and added to antecedentsAndConsequences, which is discussed later. antecendentsAndConsequents.csv is parsed in the constructor for the ConsequentsList class, which is only called once, at the start of the simulation. The BasicVisual class contains one instance of a ConsequentsList called conslist which is initialized immediately and then holds the antecedent and consequent rules data for the rest of the simulation.

Metaballs: Metaballs are a graphics concept used to render irregular shapes. Metaballs have "charge" and at each point in three dimensional space, the net charge of all metaballs present on the scene can be calculated according to some pre-defined equation to form a "field value" at this space. A

threshold is set such that locations whose field values above the threshold are considered within a metaball, and those below are outside the metaball. The surface of the metaballs is then defined by all the locations where the field value is equal to the threshold.

The contribution of a single metaball to a field value at a point (x, y, z) is defined in the simulation by the following expression:

Charge of the metaball / (distance between the metaball's center and point (x, y, z))^4

The net field value at (x, y, z) is then the sum of these values for each metaball in the scene. More commonly, the equation people use to define metaballs is similar to this, but with a second power value in the denominator instead of fourth power. The fourth power equation was chosen to model real forces more accurately: cells that are not in immediate contact should exert essentially zero force on one another, so we choose an equation that drops off very rapidly with distance.

Because metaball charge is signed, we have access to different types of metaball-to-metaball interaction depending on whether two metaballs are same-signed or differently signed. Same-signed metaballs will visually attract one another when brought close together. They will appear to merge into a single blobby shape. Differently-signed metaballs will repel and will visually deform to appear as though they are pushing against one another. For most purposes in the simulation, metaballs need to be differently signed, as the majority of inter-metaball interaction used in the simulation involves separate cells pushing one another. This, however, presented a problem because there is no way to make every metaball have a different sign than every other metaball. This problem is handled by repeating calculations on a per-metaball basis: select one metaball to treat as being positively signed, temporarily treat all others as negatively signed, and perform calculations on the field near the positively signed ball. Repeat over every other ball. This is handled in the function setFieldNearBall() in the BasicVisual class.

The Metaball class handles most functionality necessary to perform operations over a single metaball, however many of the simulation calculations need to occur in the BasicVisual class because it has access to all metaballs in the scene. The most important function is setFieldNearBall() in BasicVisual. It looks at a single ball and applies the treatment described in the previous paragraph. It then calculates the field value at all points within the ball's radius of influence, which is a value proportionate to the magnitude of the ball's charge. Outside of this radius, the contribution of the ball to the metaball field is considered to be so small as to be negligible. The function then sets the value of the field array to true for all points found to have charge above the threshold value. This information is used by the marching cubes algorithm to render the metaballs in 3D. This is described in more detail in the next section.

The metaball field is also used to model forces in the simulation. Since a metaball contributes a larger charge to the field at points closer to its center, these locations with higher field values would also be locations in which "pushing" forces coming from that metaball are stronger. Using the metaball field to model the forces of cells pushing against each other in the shell was a natural logical progression.

Cell motion in the simulation works with a local search algorithm. That is, per cell at each time step, the force field at each adjacent location is calculated and the cell will move to the one that is found to be

lowest. Much like real life, in which cells take the path of least resistance but do not have any "knowledge" about globally optimal positions in which the forces would be minimized, the simulated cell is taking the best option that it can "see." This functionality occurs in the function considerAllSides() in the BasicVisual class.

Because the shell also exerts a pushing force on the cells within it, when the total field charge is calculated at a given point (this occurs in the functions netChargeHere() and netChargeMinusThis()), the contribution of the shell's force is also added in by calling the method ellipsoidContribution(). This is a mathematical function that returns its highest value at the surface of an ellipsoid (the shell is an ellipsoid) and drops off with the square of the distance from the ellipsoid surface. So, much like real life forces, these will be strongest if a cell is pushing into the shell surface, and weaker if it has some distance from the shell.

Marching Cubes: Marching cubes is an algorithm that was developed in the 1980's for rendering metaballs in 3D. It approximates the surface of the metaball field as a series of polygons. Because this is a famous algorithm, I won't explain exactly how it works here, but will cover the ways in which my implementation is unique. My own implementation of it attempts to trade time at the expense of space: a huge amount of data is stored so that it can be accessed quickly in the future without having to repeat time-expensive calculations.

The marching cubes implementation is mostly contained in the Cube class. The methods that are located there perform the marching cubes algorithm on one cube. The three dimensional space of the model is divided up into a grid of cubes. The field gridSize in BasicVisual defines how granular the image is. A larger number means the grid is coarser; each cube in the grid is larger. This also makes the calculations faster. The grid size is a user-configurable variable, but it's limited to sane ranges by a slider bar.

The method in BasicVisual called iterateThroughGrid() is responsible for calling marching cubes on each cube of the grid. It is the most time-expensive function in the simulation. It must be called any time the image needs to change, but it should be called only when absolutely necessary.

There are a couple of fields in BasicVisual that hold a very large amount of marching cubes data. The Boolean array called field indicates whether each vertex of each cube is above or below the threshold value (see the previous section on Metaballs for information on the threshold), which is all the information that marching cubes needs in order to make its calculations.

iterateThroughGrid()calculates the field information, uses this data to determine what shapes need to be drawn, then stores the locations of the vertices of each shape into the variable called vertices. This variable is a list of list of coordinates. The reason for that complexity is that each shape needs to be specified separately. A single list of vertices makes up one shape. The vertices variable is a list of shapes. This variable holds a very large amount of data, but storing this data saves a lot of time because rendering the pre-calculated shapes is much faster than recalculating the shapes. The function printVertices() in BasicVisual iterates through the list of shapes and actually renders them. This method is relatively fast and is called constantly in the draw() loop; the expensive method is iterateThroughGrid() which does the actual calculations.

Other pertinent fields in BasicVisual are displayColorField and uniqueColorField. displayColorField holds the colors of each shape in vertices. uniqueColorField serves the same purpose for the unique colors that are used in picking (see the picking section below for more information on unique coloring and why it is necessary). Again, a lot of data is stored here, but polling the pre-stored information is much faster than recalculating it.

Picking: Object picking is the concept of being able to click an object in 3D space and detect which item was clicked. Picking is used in this simulation; individual cells are pickable, and when they are picked, their color becomes more prominent and information about the genes it contains is printed on the right half of the screen.

Object picking can be implemented in multiple ways. This simulation uses one of the common ways, which is the use of a color buffer. Conceptually, the way this works is that each cell has a unique color associated with it. When a click occurs, the cells are very briefly rendered in their unique colors instead of their normal display colors. This is handled by the method getHiddenColor(). The program then detects the color of the pixel that was clicked, and because this is unique, we can then use it to associate it with the proper cell object.

All of this functionality is handled in the mouseClicked() function in BasicVisual.

Progressing forward in time: Progressing forward in time happens at many levels: the gene level, the cellular level, the "shellular" level, and the visual level.

Gene: The method that handles time progression in genes is in the Gene class and is called updateCons(). It has two inputs: an integer called stage, and a Boolean value called recentGrowth. Stage is the number of cells present in the shell at the time that the function call is occurring. RecentGrowth indicates whether or not the shell has gained new cells (cell division has occurred) since the last time this function was called.

Each Gene object contains a list of relevant rules, which are rules in which the gene of interest is an antecedent. This is represented in the field relevantCons which is a list of Consequence objects. updateCons() is responsible for updating this list for one Gene object to remove old rules whose timeframes have expired and to add new rules whose timeframes have just begun.

Cell: The method that handles time progression at the cellular level is in the Cell class and is called timeLapse(). It has the same two inputs as updateCons() from the gene level. The function starts by looking at each gene located within the cell and applying gene-level time progression to it as described above.

At the cellular level, the next thing that happens is that a method named applyCons() is called. This is one of the more complex methods in the simulation. This method's job is to look at all of the genes within a cell that have recently changed, then look at that gene's list of relevant rules (see gene level time progression for details) to see if any of those rules' antecedents have been fulfilled. We look only at recently changed genes for the sake of efficiency; clearly if no changes have occurred in any of the

relevant genes, then the rule will not be invoked. For each rule, the rule will be invoked only if it fulfills this set of requirements:

- The antecedent genes and the consequence gene all exist in this cell
- These genes are not split among different compartments in the cell
- The antecedent genes are all set to their proper state
- The consequence gene is not set to its consequential state (reduces redundancy)
- No other rule already altered the consequence gene in this time step (reduce redundancy)

The consequences that occur are compiled into a list called “effects”. The effects are applied after all of the genes have been analyzed, such that changes cascade on a per-time step basis. All of the consequence genes for rules that were invoked are then added to the cell’s list of recently changed genes for analysis during the next call of applyCons().

Shell: At the shell level, the function that handles forward progression is called timeStep() and does not take in any inputs. It first performs any cell divisions that need to occur on this time step, then calls the cell-level time progression function over each cell in the shell.

First we must look at every cell’s field called divide, which is a DivisionData object containing a field called time, indicating when that cell’s division occurs. The cell division method must then be called on any cells whose division time has arrived.

Cell division is handled in a function called cellDivision() found in the Shell class. It takes in the information from the events queue relevant to the dividing cell (this is stored in a DivisionData object) and outputs an object of type CellChangesData. This type holds a list of cells to be removed (cells that are dividing and thus ceasing to exist) and a list of cells to be added (the daughters of the divided cells). It outputs this information so that the effects can be propagated after the method is done computing.

The cellDivision() method first calculates the names of the daughter cells from the name of the parent cell by calling nameCalc(). Then it determines which genes each daughter will inherit from the parent cell by calling childGenes(), which will be discussed later. Then the colors of the daughter cells are determined by first looking at the current color mode (lineage, fate, or pars-based) and then calling the corresponding function.

cellDivision() then performs some math to determine size and position of the daughter cells. Some of the provided DivisionData includes the axis along which the division occurred and the percentage of the volume that went to daughter1 vs. daughter2. The daughter cells will retain the dimensions of their parent in both of the axes that are *not* the axis of division. The axis of division will be reduced according to the percentage assigned to each daughter (e.g. a division occurring along the x axis in which daughter1 percentage is 60 will produce a daughter1 cell with the same Y and Z lengths as the parent, but an X length that is 60% of the parent’s X length). Note that the daughter cells will not visually reflect these values exactly, since the Metaball representation is more complex than what was described; however, this is how numbers will be updated in the Cell objects. The center point of the cells will be the

same as the center point of the parent cell in all axes except the axis of division. On that axis, the center point will move to the middle of the newly calculated length.

As briefly alluded to before, childGenes() is a fairly complex method called by cellDivision() that determines inheritance of genes from parent to daughter cell. It takes in the name of the parent cell, the axis upon which the parent is dividing, and a Boolean value indicating whether we're calculating daughter1 or daughter2's inheritance. First it looks for and then moves any genes that are changing cell "compartments" during this division. Then it iterates through all of the genes contained in the parent cell and looks at the compartment of each. If the cell is dividing along the X axis, genes in the anterior compartment will go to only daughter1 and genes in the posterior compartment will go to only daughter2. Genes in the central compartment will go into both daughters. Equivalent cases exist for dorsal/ventral and left/right divisions and compartments. This was the model chosen to simplify gene movement in this simulation. After verifying the location of each gene and adding it to a list if it is inherited by the relevant daughter, we return that list.

After cell division, all cells that have divided are removed from the list of present cells, and their children are added to the list. This information was stored in the CellChangesData object returned by the cellDivision() method. We then run cell-level time progression (as discussed above) over each cell in the shell. The display is then updated if anything visual has changed.

Visual: At the visual level, forward progression is handled in a method called progressForward() in the BasicVisual class. Its functionality varies greatly depending on whether we're currently at the farthest point in time that we've ever seen, or if we're moving forward from a past time that we've returned to. Major calculations only occur if we're moving to a never-before-seen time, otherwise, the state of the shell at this time is already stored and we just need to retrieve it.

We actually have many Shell objects in the BasicVisual class at all times, but only one, displayShell, is depicted. The other important Shell is called farthestShell and it holds the information about the farthest time step we have seen. We also have a collection of Shells that are not displayed, each of which holds a "snapshot" of what the shell looked like at each of its n-cell stages that we have already visited. At any point in time, displayShell can be set to be equal to any one of these n-cell stages or to farthestShell, and that will determine what shell's image appears on the screen.

So, if the time progression that is occurring is a new, never-before-visited future time, we call the shell-level time progression described in the previous section on farthestShell. This causes farthestShell to perform all the calculations necessary to move it forward in time by one step. Then displayShell is set equal to farthestShell to update the picture. If a cell division occurred, we store a "snapshot" of what farthestShell looks like right now into the HashMap called shellsOverTime. Then this n-cell stage can be easily revisited later if we move backwards in time.

If the time progression that's occurring is one that has already been visited, we poll the HashMap shellsOverTime to see if we need to show a new shell (only necessary if a cell division occurred on this time step). If so, we set displayShell equal to the relevant shell.

We then call the marching cubes algorithm to update the display, but only if strictly necessary because this algorithm is very expensive. We call it only if a cell division has occurred or if we're in fate color coding mode (since cell colors can occur at any time in fate mode).

Progressing backward in time: Backwards time progression is handled in the BasicVisual class in a method called progressBackward(). It is called when the left arrow key is pressed. As described above in visual-level forward time progression, we keep a HashMap called shellsOverTime in the BasicVisual class. This is basically a collection of all the different n-cell stage shell configurations we have seen. If the time step we are trying to "undo" included a cell division, we will need to poll shellsOverTime to retrieve the shell configuration that had one fewer cell than what we have now. displayShell will be set equal to this shell, and the display will be updated to reflect this change.

Mutation: Mutations are handled in the simulation by first defining a "normal" or wild-type shell behavior, and then altering this behavior with some variability in a manner consistent with empirical biological observations. Mutation occurs at the cellular and shellular level.

Cell: The algorithm that handles cell-level mutation is called perCellMutations() and is in the Shell class. If there are any mutant genes, it needs to be called on each cell that is created, so once at the beginning of the simulation and each time cell division occurs. The different types of effects that different mutant pars can create are handled each in their own function (for example, par3Mutations()), and perCellMutations() calls each one that is relevant. These functions do such things as remove the mutant gene from the cell, mislocalize other genes to a new compartment (with some variability – for example, there could be a 95% chance that a gene will be in the center compartment and 2.5% chance for each of the outer compartments on the A-P axis). The exact genes that are mislocalized for different types of mutations, as well as their probability distributions, are all based on empirical biological data.

Shell: The algorithm that handles shell-level mutation is called perShellMutations() and is in the Shell class. If there are any mutant genes in the shell, this function needs to be called once near the beginning of the shell's creation, so a call to this method is included in the Shell object constructor.

At the shellular level, the effect of mutation is that all cells in the same generation divide at the same time step. The generation of a dividing cell is one of the values stored in the cell's DivisionData field. This mutation also has the effect of changing the volume distribution between daughters one and two. The daughters will tend to be more evenly split than they are in the wild-type. A random number between 40 and 60 is chosen to be the percentage that daughter1 obtains. The times that divisions occur are subjected to some variability as well.

Exporting the simulation:

Exporting the simulation allows other people to run it without having access to all of the code behind the program. What follows are the steps to export and distribute the simulation successfully.

1. **Export JAR file through Eclipse**

With the Simworm code open in Eclipse, click File → Export → Java → Runnable JAR file. Click Next. Click the Browse button next to "export destination" and choose where you'd like the JAR to be saved. Be sure to click the radio button for "Package required libraries into generated JAR." Also ensure that the dropdown under "launch configuration" says "BasicVisual – Simworm."

2. Put CSVs and TXTs in the same directory

The executable is still dependent upon the CSV and TXT files from the Components package. Copy and paste these files into the same directory as the newly created JAR file. They can be found in your Eclipse workspace folder under src → components.

3. If you want to distribute it, place all relevant files into a .zip folder

The executable JAR, genes.csv, antecedentsAndConsequents.csv, eventsQueue.csv, wormbaseGeneInfo.txt, and Read Me.txt must all be placed in a zip folder together. Do not rename any files. Give this zip folder to anybody who wants to run Simworm!

Exporting Javadocs:

As new comments are written, the Javadocs should be re-exported and added to the docs package. Here's how to export Javadocs.

1. Write Javadocs

Firstly, an explanation on writing Javadocs. Javadocs are specially formatted comments. Writing them in a particular format allows Eclipse to detect and extract them automatically into the nice Javadocs documents. Javadocs should be written above each method after the method header is complete. Above a method, type /** and then press enter. Eclipse should auto-generate the Javadoc stub which will contain an @param tag for each input into the method, an @return tag for the output of the method, and @throws tags for any exceptions that the method throws. Type some details about each of these fields after the tag to produce your documentation.

2. Export Javadocs through Eclipse

In Eclipse with the Simworm code open, click File → Export → Java → Javadoc. Click Next. In the box below "Javadoc command," navigate to the folder where you have installed Java. Go into the subfolder for the JDK, → bin → javadoc.exe. Ensure that Simworm is checked off in the box below, and click the radio button next to the word "Private." Browse to the folder where you'd like to save the Javadocs and then press Finish.