Project number: EFR-1301

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 Warden BB ’14 and CBC ‘14

Sarah Thayer BCB ‘14

Rachel Wigell CS ‘15

Advisors:

Professor Elizabeth Ryder, Biology Advisor

Professor Matthew Ward, Computer Science Advisor

Date: April 18, 2014

Abstract:

Simulations are powerful tools that can be utilized to understand complex mechanisms within a system. Our project focuses on simulating the first several cell divisions of *C. elegans* embryogenesis. *C. elegans* is used as biological model for development, aging, and cell biology and is an ideal simulation candidate. Our four-dimensional simulation contains known information and hypotheses about molecular interactions within cells. Using rules to represent biological functions, our project visually and computationally shows the effects of several different mutations.

Acknowledgements:

Thank you to our wonderful advisors, Matthew Ward and Elizabeth Ryder, for their guidance throughout this process. Anthony Santella, Yicong Wu, and Hari Shrof were invaluable in helping our group attain 3D images of nuclei positions of the wildtype *C. elegans*. Finally, a special thank you to Ken Kemphues for sending us videos from his lab.

1 Introduction

As the amount of information on biological systems grows, so does the need for a way to integrate all of this information into one source. Even the most simple of biological interactions is much more complicated than can be fully described in a static drawing or in a paragraph of text. Computational models allow researchers to observe many different phenomena at once.

There are two main computational approaches to studying a complex system. The first of these, data-mining, is extracting patterns and pieces of information from large amounts of experimental data. The findings of data-mining can be used to pose hypotheses. Data-mining has a variety of biological applications including predicting protein structure from an amino acid sequence and creating gene regulatory networks. In contrast, simulations of a system are created using assumptions for that system and tested for accuracy (Kitano, 2002). If the output from the simulation is comparable to the biological system, then one can infer that the assumptions used to create the simulation are correct. Once a simulation is verified to be accurate, it can be used to test further hypotheses and to make predictions. Due to recent advances in high-throughput research, simulations are becoming more feasible to generate from available data.

Simulations offer a variety of advantages when compared to other computational methods or laboratory experimentation. One such example is understanding a system holistically by using data for different key components. For example, if we have an organism's genome, proteome, transcriptome, etc., do we know how a cell within the organism actually operates? Simulations offer a way of putting several puzzle pieces together and understanding the way different system components interact. Simulations also allow researchers to study one mechanism within an organism in depth and allow hypothesis testing for that one system before they understand the whole organism. For example, simulations have been used to model signaling crosstalk in cells independent from the entire organism.

As previously mentioned, simulations can be integral in hypothesis testing. There are two forms of hypothesis testing that are performed during a simulation cycle. The first of these concerns the validity of the assumptions originally used to create a simulation. These hypotheses can be tested by checking the simulation against real, biological data to see if the simulation is running as expected. If the simulation deviates too far from the biological system, then researchers need to reevaluate the original assumptions. This also allows clarity in understanding a system as a set of rules that come together to create the entire system.

The second form of hypothesis testing happens once the simulation is complete. Then, data generated from the simulation can be tested against hypotheses for the system and analysis can be performed with relative confidence that the results match the biological system. This offers logistical advantages by reducing the amount of time and cost that goes into generating quantitative data. Once a simulation is deemed accurate, it can be used to generate additional data at a fraction of the time that it would take to perform the laboratory experiments necessary to generate the same amount of data. This information gained from the simulation can be trusted as relatively accurate and be used to guide the direction of future research and laboratory experimentation. Consequently, simulations also reduce the number of costly laboratory experiments that need to be run.

This project is concerned with a simulation of the model organism *C. elegans*. *C. elegans* is studied for various biological functions such as neurobiology, aging, cell biology, and development. The advantages of using this organism in a laboratory setting are numerous. *C. elegans* is easy to culture, has a high reproduction rate, matures quickly, is transparent and therefore allows for easy observation of fluorescent markers, and has only 959 somatic cells in the adult hermaphrodite (Kaletta, 2006). Despite the advantages that make *C. elegans* a model organism, there is still a lot to learn from this model. For example, researchers use *C. elegans* to elicit information about the more complex neurological functions of humans. Because of its scientific importance and its relative simplicity, *C. elegans* is the ideal subject to create a biological simulation.

By simulating *C. elegans*, we hope to create both a teaching and research tool. Interactive simulations can be helpful tools in conveying complex mechanisms that occur on the cellular or molecular level that are hard to show in real time in a lab setting. Since *C. elegans* is used to research many different genetic phenomena, a simulation about a particular pathway in this organism can give insight about a similar pathway in a more complex organism. In addition to being used to teach biological concepts, this project also allows students to manipulate and change variables in the current simulation. By making changes to the simulation settings, students can generate and test their own hypotheses as well as learn how to work in an interdisciplinary environment.

To encourage future improvement, our simulation allows for open access modification. The code is written in Java and uses Processing, a user-friendly way to create visuals with Java. The simulation code is hosted on Github, a forum which allows anyone to modify and borrow posted code. User-friendliness is a major consideration for the project as ultimately we would like anyone to be able to collaborate on improving and adding to the functionality of the simulation. One team alone was not able to fully simulate all of the possible genetic variations for *C. elegans*, but by allowing researchers to modify and collaborate on the simulation it can become more detailed over time.

There were several defined goals of this project. The first was to create a rule based simulation of early *C. elegans* embryogenesis up to the 26-cell stage. The rules include when the cells divide, the volume distribution among daughter cells, and how the proteins interact and are inherited. The second goal was to make the simulation have stochastic, or probabilistic, features. We also wanted to track the proteins that affect cell polarity and cell fate as well as the effects of mutations in genes that affect cell polarity in embryogenesis. Specifically, we wanted to look at PAR proteins, which have a major role in assymetrically distributing proteins in the embryo. Much information is known about these proteins such as their impact on early embryo development and mutation phenotype. The final goal was to demonstrate the usefulness of simulations in the study of biological systems.

More broadly, we hope to demonstrate the advantages of computational analyses of biological systems and the potential for interdisciplinary collaboration in biological research. To help future parties develop and learn from this process and simulation, the code will be implemented in an extensible way.

In this paper, we discuss some background information that allows readers to understand basic *C. elegans* biology as well as relevant computation methods for analyzing a biological system. We also discuss

related works that not only include biology simulations but also include *C. elegans* –specific simulations. The rest of the paper goes through the process of creating the simulation, SimWorm14, the outcome of our project, and future recommendations.

2 Background

2.1 *C. elegans*

SimWorm14 focuses on the beginning of development of the nematode *C. elegans*. This organism is ideal for developing a computer simulation because of its simplicity and the wealth of knowledge that has been documented on the species. The worms have two sexes: male and a self-fertilizing hermaphrodite. As adults, the animals have only approximately 1,000 somatic cells which allow researchers to study their development and neurological systems in great detail; this is a unique feature of *C. elegans* because it is nearly impossible in more complex organisms (Atlun, 2006). The worms are microscopic in size and can be easily maintained on agar plates with *E. coli* as food. Their transparency and size also make them easily observable at a cellular level. The worms become reproducing adults approximately 3 days after they are born, which means that breeding and repetition of experiments can happen relatively fast (Atlun, 2006). This allows researchers to obtain a lot of information in a relatively short period of time.

C. elegans is also extremely well documented as a species. The fate and lineage of every somatic wildtype cell has been traced and is highly invariant between individuals. During embryogenesis 671 cells are generated, with either 113 (hermaphrodites) or 111 (males) undergoing apoptosis. The remaining cells either terminally differentiate or become postembryonic blast cells (Sulston, 1983). Besides the lineage of the cells, many of the genes in *C. elegans* have known functions and many of the interactions between these genes are known.

The data on the growth and development of *C. elegans* helped us create an accurate simulation of the species. Since there is such a vast knowledge of the species and only a short amount of time to create our project, we focused on some of the genes and proteins that affect the very beginning of embryogenesis. In the future, other data can be added to the project to help simulate what happens further on in *C. elegans* development.

2.1.1 Some Basic *C. elegans* Terminology and Nomenclature

C. elegans is described by three principle axes: anterior-posterior (A-P), dorsal-ventral (D-V), and left-right (L-R) (Fig. 2-1). These axes help provide a coordinate system for the nematode and are used to describe the position of individual cells. The A-P axis refers to the front and back end of the embryo, the D-V axis refers to the top and bottom of the embryo, and the L-R refers to the two sides of the embryo (Gönczy, 2005).

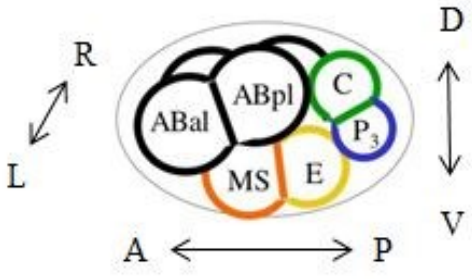


Figure 2-1: Depiction of a *C. elegans* embryo at the 8-cell stage (Adapted from Gönczy, 2005). The axes are labeled with the abbreviations for the three principle axes. The most anterior part of the embryo is to the left. It is customary to orient photos and drawings of *C. elegans* with the anterior (A) to the left and posterior (P) to the right.

The embryo starts as a single fertilized cell and divides into smaller daughter cells until it hatches at 558 cells (Sulston, 1983). These cells all originate from the six founder cells: AB, E, MS, C, and D are the five somatic founder cells and P_4 is the germline founder cell. The five somatic founder cells were named arbitrarily and progeny are named by adding lowercase letters indicating their approximate axis of division (Sulston, 1983). For the first part of *C. elegans* development all of the cells follow these nomenclature rules, except for the daughters of P_4 which are named Z_2 and Z_3.The figure below shows the six founder cells and the basic nomenclature of the descendants of the founder cells.

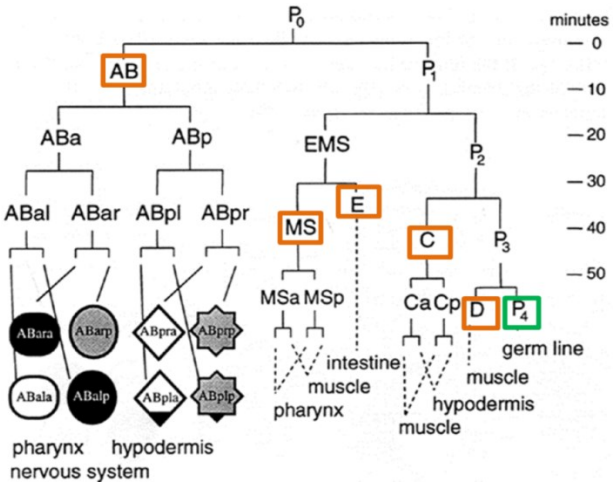


Figure 2-2: Early cells in embryogenesis (Adapted from Riddle, 1997). This lineage diagram shows the six founder cells with the somatic cell founders blocked in orange and the germline founder blocked in green. The two daughters are labeled below the parent cell. The vertical axis labels the time of development at 25°C.

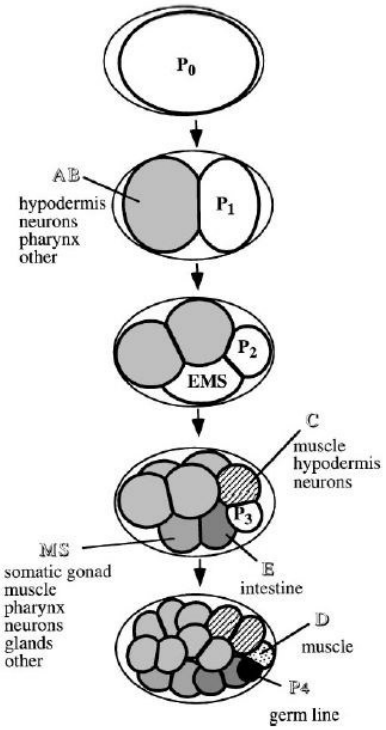


Figure 2-3: Founder cells' formation (Adapted from Rose, 2005). It can be seen that P0, P_1, P_2, P_3, and EMS divide asymmetrically to form the six founder cells.

The parent of each founder cell asymmetrically divides with its daughter cells inheriting different mRNA and proteins as well as obtaining two different sizes. The asymmetrical sister cells also divide at slightly different times and have different tissue fates. The descendants of each founder cell generally all have the same volume and divide synchronously. As displayed below, P_1, P_2, P_3, and P_4 inherit a smaller volume than their sister cells and E inherits a smaller volume than MS (Figure 2-3).

2.1.2 PAR Proteins

Before the oocyte is fertilized, the cell is fairly uniform. Once the sperm enters the cell, the centrosome it donates breaks this symmetry and causes anterior and posterior polarization of the cell (Motegi, 2013). After this initial break in symmetry, the partitioning defective (PAR) proteins come into play. PAR proteins are a set of cytoplasmic proteins that are involved in the asymmetric divisions of *C. elegans*. PAR proteins control cell polarity and help determine cell fate. In the zygote PAR-3, PAR-6, and atypical protein kinase C-like 3 (PKC-3) form the anterior domain, PAR-1 and PAR-2 form the posterior domain, and PAR-4 and PAR-5 remain uniform throughout the cytoplasm and cortex (Figure 2-4; Nance, 2005b).

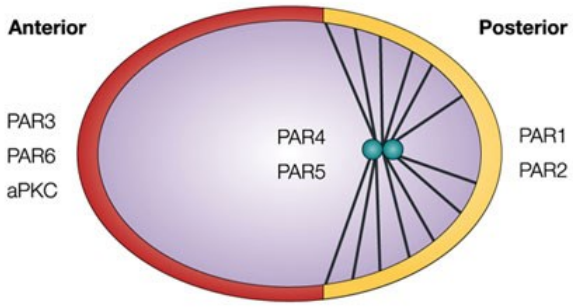


Figure 2-4: Distribution of the PAR proteins in *C. elegans* zygote (Macara, 2004). PAR-3, PAR-6, and PKC-3 are enriched in the anterior cortex, PAR-1 and PAR-2 are enriched in the posterior cortex, and PAR-4 and PAR-5 are uniform throughout the cytoplasm and cortex.

Before the first cell division, cytoplasmic flow helps guide the PAR-3, PAR-6, and PKC-3 towards the anterior of the cell and restricts PAR-1 and PAR-2 to the posterior. PAR proteins help to localize each other and maintain their distinct domains of the embryo. The asymmetrically distributed PAR proteins also direct the asymmetric localization of MEX-5/6, MEX-3, and GLP-1 toward the anterior and PIE-1, POS-1, MEX-1, and SKN-1 toward the posterior (Rose, 2005; Gönczy, 2005). When there is a mutation in any of the *par* genes, these proteins can become mislocalized and alter the tissue fate of many of the cells in the embryo. In addition, mutations in the *par* genes generally cause equal and synchronous cleavages throughout development (Rose, 2005).

Between the 4-cell and 26-cell stages the PAR proteins become important to the apicobasal polarization of the embryo cells. PAR-3, PAR-6, and PKC-3 become restricted to the apical surfaces of the cells starting in the late 4-cell stage. PAR-1 and PAR-2 become restricted to the sites of cell to cell contact. PAR-5 remains uniform throughout the cytosol and is thought to help with the interplay between the anterior and posterior PAR proteins. The PAR proteins gradually degrade after gastrulation begins (Nance, 2005b).

Table 2-1: PAR Proteins and their Function in Early Embryogenesis

Protein	Wildtype location at 1 cell stage/ early embryo	Function
PAR-1	Posterior cortex/ basolateral cortex	A kinase that excludes PAR-3 from the posterior of the zygote and helps localize SKN-1 and PIE-1
PAR-2	Posterior cortex/ basolateral cortex	Helps localize PAR-1 and restricts PAR-3 to anterior in presence of PAR-1
PAR-3	Anterior cortex/ apical cortex	Forms a complex with PAR-6 and PKC-3 and helps localize PAR-1 and PAR-2
PAR-4	Uniform throughout	A kinase that helps activate PAR-1 and localizes P granules to the posterior and GLP-1 to the anterior.
PAR-5	Uniform throughout	Helps localize P granules, PAR-3, and MEX-5. Helps maintain the boundary between the anterior and posterior PAR proteins.
PAR-6	Anterior cortex/ apical cortex	In a complex with PAR-3 and PKC-3. Helps maintain PAR asymmetry during early embryogenesis.
PKC-3	Anterior cortex/ apical cortex	Phosphorylates PAR-1 and PAR-2 excluding them from the anterior.

Information for this table was compiled from several sources: (Morton, 2002), (Boyd, 1996), (Aceto, 2006), (Nance, 2005b), (Reese, 2000), (Motegi, 2013), and (Crittenden, 1997)

2.1.3 Determining of Cell Fate

In the wildtype, the cell fate of every cell is known. These fates are determined partly by the proteins that the cells express. Mutations that change protein expressions can change a cell's fate. As described in the previous section, *par* mutants can cause certain proteins to become mislocalized. The proteins that are mislocalized can change the cell fates by inhibiting normal protein expression or by activating abnormal protein expression. These reactions can be fairly complicated, but the table below (Table 2-1) lists a few of the proteins that this project focuses on and their role in determining cell fate.

Table 2-2: Proteins in the Early Embryo and their Function

Protein	Description
SKN-1	Required to specify EMS cell fate, suppresses PAL-1 expression
PIE-1	Required to specify germline cell fate, suppresses SKN-1 and PAL-1 expression
PAL-1	Required to specify C and D cell fate
GLP-1	Transmembrane protein that binds to APX-1 and other ligands to produce the different AB cell fates
APX-1	A protein secreted from P_2 that binds to GLP-1 to determine ABp cell fate
MEX-1	Highest concentration in the germline cells, involved in APX-1 regulation
MEX-3	Inhibits PAL-1 translation in the anterior in wildtype
MEX-5/6	Prevents PIE-1 accumulation in the anterior in wildtype

Information for this table was compiled from several sources: (Tabara, 1999), (Huang, 2002), (Gönczy, 2005), and (Rose, 1998)

Although this is not the full list of proteins that are involved in fate determination, one might imagine that even if these few proteins were in the wrong cells that there might be significant consequences. For

example, if MEX-3 were distributed to all of the cells and not only to the AB lineage, PAL-1 would not be translated in any of the cells and thus prevent any cell from having the C or D cell fate.

2.2 *i*SPIM and AceTree

In order to be able to compare our model to biological data, we used *i*SPIM (Wu, 2011) and AceTree (Boyl, 2006) to track *C. elegans* nuclei through early embryogenesis. Inverted selective plane illumination microscopy (*i*SPIM) is a noninvasive high-speed volumetric imaging technique that was developed by Yicong Wu and Hari Shroff with help from their colleagues at Yale University and Memorial Sloan-Kettering Cancer Centre. With this tool, *C. elegans* embryos can be constantly monitored by having their volumes scanned every 2 seconds over the course of embryogenesis with no detectable phototoxic effect (Wu, 2011). Using *i*SPIM, nuclei labelled with histone:mCherry and other fluorescent markers can be tracked and then lineaged using StarryNite and AceTree software.

The StarryNite (Boyl, 2006) software is used to track the nuclei recorded in the thousands of *i*SPIM images and record the nuclei locations and lineage relationships. AceTree is then used to edit the information that StarryNite produces and can create a 4D visualization of *C. elegans* cells. Other programs, such as SIMI BioCell (Schnabel, 1997), AceDB (Stein & Thierry-Mieg, 1998), and Virtual WormBase (Rogers, 2008) produce comparable 4D modeling. However, these programs are optimized for 4D differential-interference-contrast (DIC) image series but are not ideal for showing fluorescence to track genes through embryogenesis. AceTree was developed specifically for the purpose of using fluorescence, making it a useful addition to the existing programs. It was also useful to develop AceTree in conjunction with StarryNite to maintain compatibility and create an open source package from which other developers can learn (Boyl, 2006).

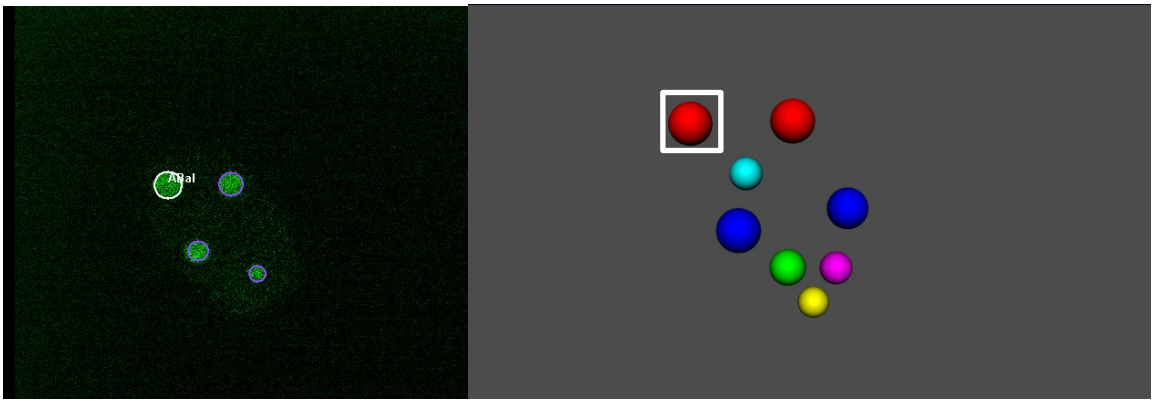


Figure 2-5: AceTree (Boyl, 2006) visualization pane and 3D rendering. The visualization pane showing the 2D tracking of nuclei at time point 37 and plane 15 (left), 3D rendering of the nuclei at time point 37 (right). In both images the cell ABal is highlighted.

AceTree provides various features to help view the data. A cell can be tracked using the nucleus identity through the visualization pane which displays the labeled nuclei at a specific time point and plane. The nuclei can be annotated using conventional Sulston naming standards by editing the lineage. This involves deleting false nuclei identifications and connecting daughter cells to the parent cell. Using the lineaging data, various ancestral trees can be constructed to better track the cellular divisions. One of the more important features of the program relevant to this project is the ability to create a 3D image of the

nuclei positions in the embryo. This 3D rendering not only tracks the cell identities but the lineage according to a user designated color scheme.

The motivation behind developing this tool was to track the nuclear position of cells expressing certain transcription factors. For example, several of the strains designed and generated during this project contain the marker *ceh-10*:GFP. This marker enables the visualization of the cells that express the CEH-10 transcription factor. CEH-10 is important for the differentiation and development of a subset of neurons, and by labelling these neurons, their migration can be followed throughout embryogenesis (Wu, 2011). By imaging different mutant strains, researchers can see how different genes affect the migration of these neurons.

AceTree continues to be developed, constantly increasing accuracy and decreasing the amount of time it takes a user to annotate the data. The long term goal of the project is to help researchers analyze data that is less easily quantifiable.

2.3 Simulations and the Simulation Cycle

Simulations are computer generated methods of studying a system. Mathematical or logical assumptions are made about a system to make up the model. Sometimes these mathematical relationships can give an answer to the question at hand to produce an analytic solution. Oftentimes, however, a real-world system is too complex for this and a simulation must be used. A simulation evaluates a model using mathematics which are then used to estimate characteristics of a system. There are three important characteristics that describe a model.

The first characteristic describes a simulation as being either static or dynamic. A static simulation is used for either a system at one time point or a system where time does not play a role. A dynamic simulation, however, deals with the changes of a system over time. The next characteristic is either deterministic or stochastic. A deterministic model does not use probabilities to predict the events in a simulation and will always produce the same output given a specific input. A stochastic model introduced variability by using probabilities to produce random outputs given a specific input. The third characteristic is either discrete or continuous. Discrete simulations, or discrete event simulations, have events that occur at specific time points which change the state of the system. Continuous simulations do not use specific time points to change events, but rather, events change the variables continuously (Law, 2000).

Simulations are becoming an integral part of analyzing biological data because programs can produce a high quantity of results which can then be compared to experimental data. The comparison of simulation generated data against experimental data is called the simulation cycle. As described in Fisher, 2007, the simulation cycle is the process of developing a program over time based on two findings: the simulation accurately matches biological observations or the simulation does not accurately match biological observations. The first case is that if the simulation's output is comparable to the experimental results, then the model can be considered accurate. Thus, researchers can move forward generating more data from the model and assume that it is biologically accurate. However, if the simulation and experimental data are not comparable, then the simulation needs to be adjusted (Fisher, 2007).

Harvest algorithms are an example of the simulation cycle in progress (Cooke, 1999). These algorithms use available data to recommend a sustainable harvest level which is integrated into the fisheries'

management policy. Figure 2-6 shows a summary of this process which begins with data being used to develop the algorithm. Then, the algorithm is used to make decisions for the fishery. The results of these decisions are recorded in research data as well as used as fishery data to compare against the model. If the fishery data shows that the model requires improved accuracy, further research data is collected as denoted by the dotted arrow.

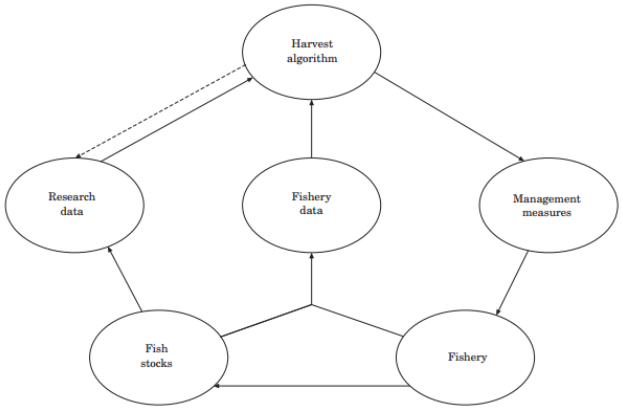


Figure 2-6: Summary of the simulation cycle for developing a harvest algorithm (Adapted from Cooke, 1999)

Another notable feature of developing a harvest algorithm includes starting with a simple array of possibilities. This means that researchers must take a minimalistic approach and limit the number of factors and scenarios that they take into account. This is for simplicity and allows researchers to make sure the model works in the most basic form. Then, if the results are verified for the simplistic model, further developments can be made to make the model more complex (Cooke, 1999).

There are several advantages to using the simulation cycle for generating data versus only conducting biological research. For the fishery, it would take years or decades to generate the type of data that they were looking for to make smart management decisions. It would also require a large sampling of fish given that different populations vary and could produce varying results on an individual level. For the simulation, this data could be modelled theoretically without needing a physical population. Simulations also offer the benefit of hypothesis testing. The fisheries could consequently predict failures using a simulation and try to prevent them from occurring in reality. This allows them to understand the cause of a problem whereas, without a simulation, they would only see the issue and not the underlying causes. Finally, the simulation allows the fisheries to conduct their research at a lower cost compared to real time.

Our project closely followed the simulation cycle by constantly checking the simulation's output with biological data. We spent the first part of our project completing in depth research and organizing collected information in an easily accessible medium. Then, the simulation was created using that data and the output was compared to the previous research. The simulation was updated according to the accuracy that was observed.

2.4 Object Oriented Programming

Object oriented programming is a style of handling data that is used by many languages, including Java, the language in which our project is programmed. It provides a useful abstraction that allows the

programmer to divide the codebase into classes, which represent different concepts that the programmer hopes to keep distinct. Classes house their own pertinent methods, which are processes which take in input, perform computations, and produce output. For example, in our project, cells are a class, and all functionality pertaining to cells is written inside of the cell's class.

Using an object oriented language to program the simulation was a logical decision. Simulations tend to involve many different types of objects and complex interactions between different objects. Staying organized within the class structure helps the programmer to keep track of an immense amount of data.

The code contains classes to represent shells, cells, and genes, as well as some other abstract concepts such as three dimensional coordinates. The class hierarchy forms itself naturally since it is modelled after biological reality – shells contain cells, which contain genes.

3 Related Works

In order to gain a background of current simulations, our group researched other simulations to understand their purposes, functions, and contribution to scientific discovery. In this section, we highlight some general biological simulations as well as simulations specific to *C. elegans*.

3.1 Biological Simulations

While simulations are diverse in application, the particular interest of this project is understanding biological processes. Specifically, we are focusing on *C. elegans* at the cellular level. This section highlights select biological simulations that have been previously created on the cellular level and used by researchers to gain understanding on cellular processes.

3.1.1 The Virtual Cell

Developed at the University of Connecticut Health Center, the Virtual Cell (Loew, 2001) is a platform for modeling chemical reactions, such as diffusion, within a cell. The basic mechanism is that it takes in mathematical equations describing the model using Virtual Cell Mathematic Description Language, VCMDL, and converts it to C++. The program then produces a VCMDL description based on the input of the program user and resulting model.

Because of the interdisciplinary nature of Virtual Cell, it brings together experimental biologists and mathematical modelers, thus bridging the knowledge between these communities. Another desirable feature of Virtual Cell is that it can be used by all levels of disciplines, including those with little programming background. The model uses a java applet to take in molecule identities, reaction and transport properties, and cell compartmentalization data, producing a tailored prediction of the molecular interactions.

Since its creation, Virtual Cell has been used for several research projects. These projects range in application from pathogen research to cellular insulin secretion. One study used Virtual Cell to simulate the passive transport system of drugs across membranes. Figure 3-1 shows the setup of the system described in this paper as created in the Virtual Cell environment.

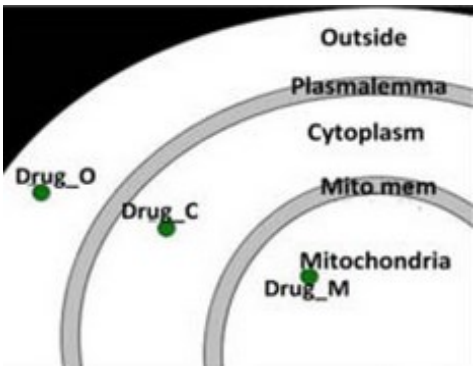


Figure 3-1: Setup of Virtual Cell environment to test drug transport in the cell (Adapted from Baik, 2013)

The researchers noted that Virtual Cell was an excellent environment for scientists of all backgrounds to study systems pharmacology and biopharmaceuticals (Baik, 2013). All documented works using Virtual

Cell can be found on the Virtual Cell website:
http://www.nrcam.uchc.edu/vcell_models/published_models.html?current=five.

In the future, Virtual Cell developers would like to include automatic linkage to database information so the simulation can draw previous knowledge from the literature to automatically use those values as input. The researchers also recognize a fundamental flaw with their simulation – there is no function to handle changing geometries and therefore simulations of cell migration and mitosis are not suited for this program (Loew, 2001). In contrast, we would like our simulation to allow for migration as it develops over time. This is just an example of the various ways simulations can be approached and how the needs of a researcher can be used to prioritize the development.

Both SimWorm14 and Virtual Cell deal with subcellular interactions, even though our simulation deals specifically with proteins and Virtual Cell is used for various other interactions. Virtual Cell is also an important stepping stone when it comes to interdisciplinary research in the computational biology community. The program is written in a way that appeals to both biologists and programmers, a goal that we hope to accomplish through our simulation. Virtual Cell is also an example of a simulation with a narrow scope.

3.1.2 SmartCell

The purpose of SmartCell is to be able to model various biological processes (Ander, 2004). Developers hoped to compare the predictive results of SmartCell with that of differential equations. It is written in C++, uses Extensible Markup Language (XML), and works with systems biology markup language (SBML). There are three categories for entities, or the objects in the simulation: reactants, products, and effectors. The program outputs a text file that has snapshots of the cell that can be converted to animated movies or graphs.

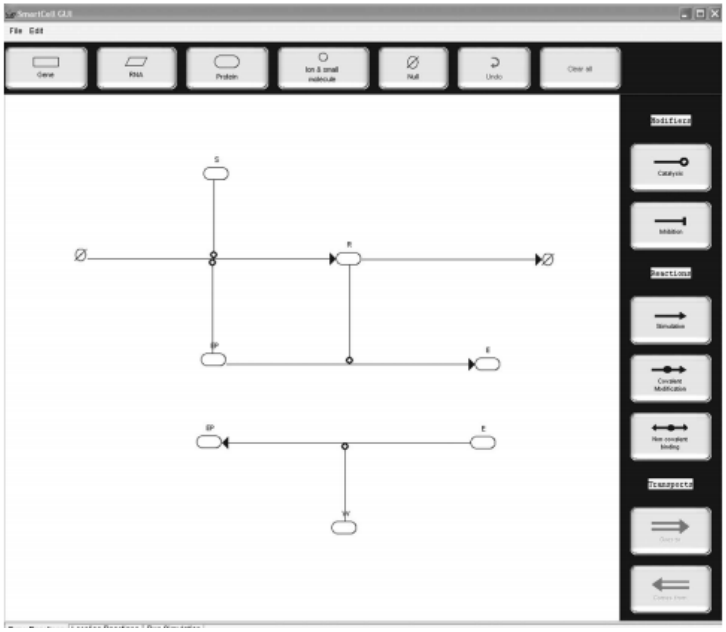


Figure 3-2: The SmartCell interface where interactions are visually shown (Adapted from Ander, 2004)

SmartCell developers prioritized cell geometry in their simulation. To help with the geometry of the cell, the volume is subdivided into smaller pieces. Then, a set G is defined G = {S, E} where S is the set of vertices and E is the set of edges.

In the simulation, a stochastic events queue is designed to model either reaction or diffusion events. The general algorithm is described in the excerpt below:

1. *Set the initial numbers of molecules.*
2. *Calculate the probability a_i for each event i.*
3. *For each event i, sample a putative reaction time T_i from an exponential distribution with parameter a_i; and add it to the queue of events.*
4. *Pick the event with the lowest T from the queue of events.*
5. *Execute the event, recalculate a_i; generate a new T_i and add it to the queue of events.*
6. *Check dependencies and update 'dirty' Ts in the queue of events.*
7. *If the queue of events is not empty, go to step 4, otherwise terminate.* (Randel, 2004)

Here, dirty Ts are the reaction times of events that have been changed by subsequent event times.

Because SmartCell uses a stochastic algorithm, it is useful to gain information about certain interactions. For example, SmartCell has been used in two published articles: "Noise in transcription negative feedback loops: simulation and experimental analysis" (Mol Syst Biol. 2006;2:41. Epub 2006 Aug 1) and "Cell type-specific importance of ras-c-raf complex association rate constants for MAPK signaling" (Sci Signal. 2009 Jul 28;2(81):ra38). In the first article, researchers set up three different reaction circuits and tested them using SmartCell. In the second article, researchers used SmartCell to simulate the effects of negative feedback for Ras-cRas binding in correlation with extracellular signal-related kinase activation.

Overall, SmartCell simulates diffusion and localization using a stochastic approach. The results of the study showed that the stochastic model was much more accurate than the ordinary differential equation (ODE), deterministic model. Future improvements include automatically scanning images to pick up on cell geometry, linking the graphical user interface (GUI) to databases, parallelization of the code, and incorporating other approaches such as ODEs and hybrid modeling.

3.2 *C. elegans* Simulations

Although *C. elegans* is a model organism for various reasons, it is still complex and therefore simulations are useful in fully understanding different molecular interactions. The following section outlines some previous simulations used to research *C. elegans*.

3.2.1 The Perfect *C. elegans* Project

The Perfect *C. elegans* Project (Ketano, 1998) developers' main goal was to see if researchers were able to computationally recreate genetic interactions that match the phenotypic response in living organisms. It can also be used to visualize embryogenesis and identify cells that are interacting during that period. Although this project made three simulations, the one that pertains most to our MQP is the visualization and simulation of embryogenesis.

The embryogenesis simulation gives a 4D model that shows cell interactions and dynamics that runs from the first cell to approximately 600 minutes after the initial cell division. Shapes were derived from qualitative data, drawings, cell lineage charts, and migration data. This data was compiled into a computer readable chart. Although cell shape was not completely realistic in the simulation, around 200 minutes the embryo shape changes to a "comma" which was achieved by bending the cylindrical coordinate system (Ketano, 1998) as shown in figure 3-3.

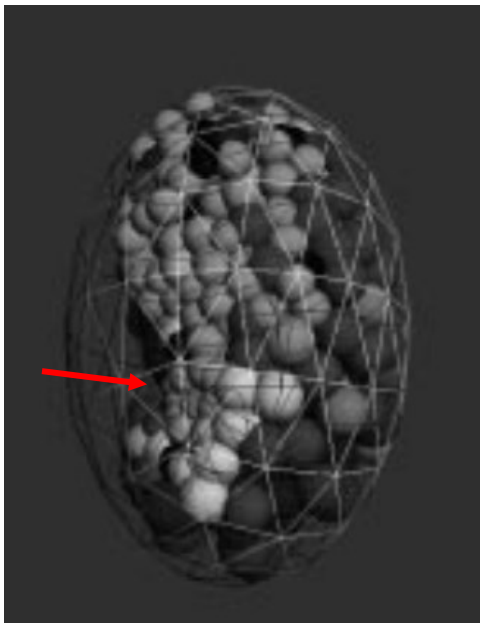


Figure 3-3: Cellular model of *C. elegans* for the Perfect *C. elegans* Project (Adapted from Ketano, 1998). The comma shape of the embryo can be seen by the indentation in the middle, left of the cell mass as indicated by the red arrow. The cells are encompassed by a shell structure.

The achievement of modeling embryogenesis on a cellular level is a large step in *C. elegans* research but our project group endeavors to include greater protein detail than this model from The Perfect *C. elegans* Project. While visualization is a large component of our project, we are also incorporating gene expression data into our simulation in a way that makes it stochastic. Making the simulation probabilistic is what really sets our project apart from previous research.

3.2.2 Modeling Signaling Crosstalk

In the study presented in (Fisher, 2007), developers recognized the importance of having a dynamic, phenomenon-based simulation compared to a static model. They wanted to model the development of *C. elegans* vulva, specifically tracking the inductive and lateral signaling pathways. Because the model is written in reactive modules (RM), modules are used to describe objects in the system and contain variables pertaining specifically to the module.

The three main modules used in this simulation are worms, a gonadal anchor cell (AC) and six identical vulval precursor cells (VPCs). The AC module has variables that define if it is ablated or formed and the sensitivity to inductive signals from the VPCs. The VPC module variables are pathway behaviors including lateral signaling and inhibition. To test the model, experimental cell fates were compared to simulation cell fates (Fisher, 2007). This shows not only the relationship in biology between creating

objects, such as the worms, VPC and AC, but also the importance of using simulations to test hypotheses about experimental work.

This model focuses on very distinct intercellular interactions, an example of simplifying a system using a simulation. Similarly, our project begins looking at another specific piece of development; We narrowly looked at PAR protein interactions during early embryogenesis.

3.2.3 OpenWorm

OpenWorm (Idili, 2011) is a project that intends to make an open source simulation of adult *C. elegans* using biological data available from various databases and experiments. Their ideal result would be a model that simulates every single cell in *C. elegans*. However, to accomplish this, developers are simulating one feature at a time. Currently, they have a movement model which emulates the movement of the organism, as shown in Figure 3-4. They plan to continue development by modeling the nervous system next.

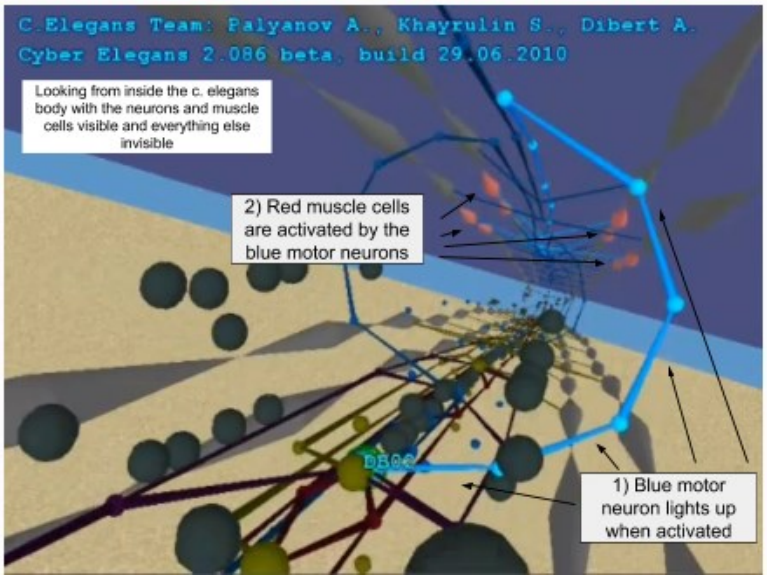


Figure 3-4: Muscle cells within *C. elegans* as simulated for OpenWorm (Adapted from Idili, 2011). Understanding *C. elegans* movement at different levels helped developers create a successful model to simulate the complex movement. In this figure, muscle cells are shown and color coded according to function.

This data is compiled into a platform entitled Geppetto that can run the different models simultaneously. The code is available on GitHub and those interested are encouraged to study and alter it for their own learning. Ultimately, the researchers aim to create an open environment where interested individuals can come to learn. This is also shown through their website, openworm.org, which helps any viewer understand the project and gives access to many of the project's documents. These documents include the working code, media downloads for both CS oriented individuals and biologically oriented individuals, and data spreadsheets. Anyone is invited to the core team meetings via Google hangouts because this practice does "a good job at expressing the true spirit of OpenWorm: openness, transparency and of course, science!" The developers have also created an interactive model of *C. elegans* online where an audience can come and learn about the different systems and rotate the model freely. OpenWorm is important because its priority is education to the public, and it bridges multiple data sets to create a complete picture of *C. elegans*.

Compared to our model, the approach is very similar in that this project will be completed by reducing the characteristics of a cell and completing one phase of the simulation at a time. However, unlike OpenWorm, we are focusing on early embryogenesis and how the worm develops over time. OpenWorm is strictly looking at a fully developed adult.

4 Laboratory Methods

4.1 Growing and Maintaining *C. elegans*

C. elegans were grown on Nematode Growth Media agar plates spotted with the mutant *E. coli* strain OP50. Strains not currently being used in experimants were grown at 15^{o}C and maintained once a week by transferring three L4s to a new plate, with the exception of N2Ms which were maintained by transferring three L4s and ten males to a new plate. Transferring or picking of individual worms involved observing the worms under a dissecting microscope and using a worm picker with a blob of *E. coli* at the end of it to transfer the worms. The worms stuck to the *E. coli* and were transferred to plates or slides. The worm picker had a platinum wire at the end of it which was flamed before and between transfers to reduce contamination.

4.2 Strains Bred for *i*SPIM Imaging

The Tables below list the fluorescent markers, strains used to create additional strains, and the strains that were newly constructed during the course of this project.

Table 4-1: Fluorescent Marker Description

pie-1 :: H2B:: wcherry	Red histone marker for early embryonic cells
his-72 ::HIS-72::wcherry	Red histone marker for larval and adult cells
ceh-10 ::GFP	Green marker for the CAN neuron
pgp-12 ::GFP	Green marker for the excretory cell
vha-1::GFP	Green marker for the excretory cell that can be imaged
dlg-1::GFP	Green marker for epithelial cells

Table 4-2: Strain Description

N2M	Wild Type mating strain
BV117*	*ceh-10::GFP; his-72:: HIS-72 :: wcherry; pie-1:: H2B :: wcherry*
BW315	*mig-10 (ct41)*
RY1212	*mig-10 (ct41); pgp-12::GFP*
VA74M	*abi-1(tm494); pgp-12::GFP*
unknown	unc-34(gm104)
RY1221**	*mig-10 (ct41); ceh-10 :: GFP; his-72 :: HIS-72 :: wcherry; pgp-12::GFP*
FT48	*dlg-1::GFP;him-8*

*Kindly provided by Zhirong Bao's lab. This wildtype strain was used to produce the AceTree (Boyl, 2006) images.
**Developed by the previous simulation group (Brandon, 2013). Was used in the cross that produced RY1320.

Table 4-3: New Strains Constructed

RY1320	*mig-10 (ct41); ceh-10::GFP; his-72::HIS-72::wcherry; pie-1::H2B::wcherry; pgp-12::GFP*
RY1321	*mig-10 (ct41); ceh-10::GFP; his-72::HIS-72::wcherry; pie-1::H2B::wcherry*
RY1323	*abi-1(tm494); ceh-10::GFP; his-72::HIS-72::wcherry; pie-1::H2B::wcherry*
RY1325	*unc-34(gm104); ceh-10::GFP; his-72::HIS-72::wcherry; pie-1::H2B::wcherry*
RY1328	*mig-10(ct41); dlg-1::GFP*

4.3 Strain Design

This section describes the crosses that were performed to obtain the strains that are presented in Table 4-3. For each mating, approximately ten males and three L4s were transferred to the same plate, and the *C. elegans* that were produced from these matings were used in the next step of the cross. The plates that were used for these crosses were stored at either 15°C or 20°C.

To check if the strains contained the desired markers, L4s and gravid, or pregnant, hermaphrodites were picked to a glass slide with an agar pad and observed under an inverted fluorescent microscope. Location of the worms on the slide were recorded prior to observation under the fluorescent microscope. Desired worms were recorded on the location map then picked from the slides and put on a new agar plate with *E. coli*.

Table 4-4: Legend for Crosses

x	indicates breeding between two different strains of *C. elegans*
↓	indicates that the desired offspring of mating directly above the arrow were selected to continue the cross
↻	indicates that the offspring of this step were created by self-fertilization and were produced by picking three L4s to a plate unless indicated otherwise
+	indicates that the worm is heterozygous for the marker or mutation
?	indicates that it is unknown if the worm is heterozygous or homozygous for the marker or mutation
::G	short for *::GFP*
::W	short for *::wcherry*

4.3.1 RY1320

A) N2 ♂ x *ceh-10::G*; *his-72::W*; *pie-1::W*

↓ pick males

B) *ceh-10::G*; *his-72::W*; *pie-1::W* ♂ x *mig-10*; *ceh-10::G*; *his-72::W*; *pgp-12::G*

\+ + +

↓ pick L4s with green full-length excretory and red labeled germline cells in gonad

C) *mig-10*; *ceh-10::G*; *his-72::W*; *pie-1::W*; *pgp-12::G*

\+ ? ? + +

↻ single; look for truncated excretory cell and all of the markers

D) *mig-10*; *ceh-10::G*; *his-72::W*; *pie-1::W*; *pgp-12::G*

? ?

↻ single until all markers are homozygous

E) *mig-10*; *ceh-10::G*; *his-72::W*; *pie-1::W*; *pgp-12::G*

4.3.2 RY 1321

Same as RY1320 cross except that after step D) the worms were singled until *pie-1::wcherry* was homozygous and *pgp-12::GFP* was bred out of the strain (no longer has green excretory cell).

4.3.3 RY1323

A) *abi-1; pgp-12::G* ♂ x *mig-10; ceh-10::G*; *his-72::W*; *pie-1::W*

↓ pick L4s with green full-length excretory cells

B) *mig-10* + ; *ceh-10::G*; *his-72::W*; *pie-1::W*

\+ *abi-1* + + +

↻ single L4s with green full-length excretory cells, check in the next generation to confirm that all the offspring produced have full-length excretory cells

C) *abi-1*; *ceh-10::G*; *his-72::W*; *pie-1::W*; *pgp-12::G*

? ? ? +

↻ single until *ceh-10::G*, *his-72::W*, and *pie-1::W* are homozygous and *pgp-12::G* is bred out

D) *abi-1*; *ceh-10::G*; *his-72::W*; *pie-1::W*

Note: The genes *abi-1* and *mig-10* are linked and so are almost always inherited together. This means that when the *mig-10* mutant and *abi-1* mutant are bred together their offspring will have a copy of the mutant *mig-10* on one allele and a mutant *abi-1* on the other allele but none of the descendants will have an allele with both of those genes mutated or wildtype.

4.3.4 RY1325

Same as RY 1320 cross except in step B) *mig-10*; *ceh-10::G*; *his-72::W*; *pgp-12::G* was replaced by *unc-34* and after step C) L4s were singled until the markers were homozygous and there were only *unc-34* mutants found on the plate. The *C. elegans* that were homozygous for the *unc-34* mutation were selected based on their characteristic phenotype, uncoordinated movement which includes rolling and curling.

Note: It was found through earlier crossing attempts that *unc-34* and *pgp-12::G* are linked. After crossing an *unc-34* with a strain containing *pgp-12::G* a much smaller percentage of offspring than expected actually became homozygous for both the marker and the *unc-34* mutation. Due to this linkage, *unc-34* mutants were selected based on their uncoordinated phenotype rather than their excretory cell appearance.

4.3.5 RY1328

A) N2 ♂ x *dlg-1::G*

↓ pick males

B) *dlg-1::G*; ♂ x *mig-10*; *pgp-12::G*

\+

↓ pick L4s with green excretory cell and *dlg-1* marker

C) *mig-10*; *pgp-12::G*; *dlg-1::G*

\+ + +

↻ look for truncated excretory cell and *dlg-1* marker

D) *mig-10*; *dlg-1::G*; *pgp-12::G*

? ?

↻ single until dlg-1::GFP is homozygous and pgp-12::GFP is no longer present

E) *mig-10*; *dlg-1::G*

Note: It was also attempted to create *mig-10; dlg-1::G; pgp-12:G* but *pgp-12::G* did not become homozygous after over 5 generations of singling, so the attempt was dropped. This difficulty suggests that *pgp-12::G* may be linked to *dlg-1::G*.

4.4 Freezing and Thawing Strains

Strains obtained and constructed were stored at -80^{o}C in order to preserve the strains for future use and to provide a backup of the strains that were currently being used. To prepare for a freeze, four plates of a particular strain were grown until just after starvation, so that eggs hatched and larvae were in L1 arrest. Worms were then rinsed off with M9 and allowed to settle on ice. Excess M9 was removed until there was only 2 mL of *C. elegans* mixture. This was then mixed with 2 mL of warmed freezing solution, quickly pipetted into two cryotubes, and stored in a foam storage container at -80^{o}C.

To thaw, a chunk of the frozen mixture was removed from the cryotube using a flamed spatula and was placed around the edge of a new agar plate. If after a few days viable *C. elegans* are observed on the plate then the thaw was successful and these worms can be transferred to a new plate. If the thaw was not successful and the *C. elegans* did not survive, then it is advisable to refreeze the strain.

5 Computational Methods

After understanding project background, it was important to begin the planning process of how the program design was going to be executed and what biological data was going to be represented in the simulation. The following section outlines the process of narrowing down project scope and documenting biological data in a way that is readable for a simulation.

5.1 Project Scope

Before designing the simulation itself, we first determined how complex the model should be. We had to decide how far into embryogenesis we should model and what type of biological data we should represent in our model.

We first considered simulating gastrulation, which starts at the 26-cell stage and positions the germ layers in the embryo (Nance, 2005a). This process occurs during the first 100 cell divisions which the Simulation Worm group before us had previously attempted to simulate because it is an important process during *C. elegans* development (Brandon, 2013). Gastrulation involves cell migration, cell-cell interactions, cell polarization, and morphogenesis, all of which would be important to simulate throughout embryogenesis. By starting with gastrulation, we thought we could provide the groundwork for more complex migration that occurs later in embryogenesis and be able to simulate several different mutations that affect this process. Unfortunately, after observing the simulation the group before us had created, we decided that the simulation had too many biological inaccuracies to provide a groundwork for continued development.

After our first consideration, we decided to focus our simulation on the cell divisions that occur before gastrulation. With this smaller scope we could focus on making our simulation as biologically accurate as possible and still develop rules for important biological phenomena. We focused mainly on the PAR proteins because they are active from the very beginning of embryogenesis and their function in cell polarization has been studied extensively. The embryos that have mutations in any of the genes involving these proteins have highly characteristic phenotypes. These phenotypes include evenly sized cells and an excess of cells with a particular cell fate, which can be easily shown in a simulation. Our simulation focuses on accurately representing division timing, cell fate, and protein interactions that occur within cells. It also allows the user to produce different mutants by turning off any of the selected *par* genes. Working with these early proteins lays the groundwork for groups who want to pursue simulating phenomena that occur later in development.

Aside from limiting the scope of *C. elegans* development and protein interactions, we made simplifications to movement of the cells within the shell. In this simulation, we have the cells only dividing on one of the three axes (x-axis, y-axis, or z-axis) and there is no movement of the cell once it has divided. Additionally, there are no collision forces which cause the cells to move once another cell is created. This is a simplification of reality because once a cell divides along one axis it moves small amounts due to the pushing forces within the shell when subsequent cells divide.

5.2 Data Collecting

There has been significant research done on the very beginning of *C. elegans* embryogenesis. Even though our lab does not focus on the very beginning of embryogenesis there is enough literature written on early development to create an accurate representation.

We first looked to WormBook and other reviews to get an overview of early embryogenesis. Several WormBook chapters helped provide information on early cell divisions and the determination of cell fate (Gönczy, 2005; Priess, 2005; Evans, 2005). The review "Early Patterning of the *C. elegans* Embryo" provided us with an overview of the maternally expressed genes that help determine cell fate and the patterning of the earlier cells (Rose, 1998). We also read "The Embryonic Cell Lineage of the Nematode *Caenorhabditis elegans*" to get an overview on the cell lineage and cell fates of *C. elegans* (Sulston, 1983). All of these resources provided us with the background research we needed to understand the basic concepts involved in early embryogenesis.

After looking at the overviews of the early development, we started to look into specific research papers that focused on certain pathways and protein interactions. First we found literature on the different PAR proteins, how they interacted with each other, and their specific functions (Nance, 2005b; Cheeks, 2004; Guo, 1995; Boyd, 1996; Hao, 2006; Etemad-Moghadam, 1995; Watts, 2000; Hoege, 2013). Next we looked into research on Notch-signaling and the determination of tissue fates of the AB lineage (Mango, 1994; Neves, 2005; Crittenden, 1997; and Mickey, 1996). Additionally, we looked into the Wnt signaling pathway and the effects on E and MS cell fates (Rocheleau, 1999; Eisenmann, 2005; Lo, 2004; Lin, 1998). Any information we could not easily find in research articles we supplemented with the information provided on WormBase and the online textbook C. elegans *II* (Riddle, 1997). All of the research found from the sources above helped provide us with information that we used in our data tables and event queues.

5.3 Computational Design

After collecting data that was necessary to run the simulation, we needed to figure out how to store the data in a readable structure from which a program could create a simulation. To understand our options, we looked at all of the pieces of information that we collected and how they related to each other. The very basic information was the names of all the cells and their daughter cells. To create a progression of this information each parent cell needed to be connected to its time of division. We also collected volume information for daughter cells to make the visualization somewhat accurate for *C. elegans* development. In addition, we needed information for each cell to help define its location within the embryonic shell, such as axis of division.

For the protein interactions, we needed to track protein expression in a given cell as well as interactions of the cells since this often affects protein expression. The amount of information we needed to organize presented several challenges for data storage. We explored different methods and examined the degree for which the method would allow future development. The following explains our thought process and the current structure of the data.

5.3.1 Methods Considered

Looking for an effective way to represent cell lineage information as well as protein regulation, our group looked carefully at previous versions of the project which were executed in 2004 and 2013 (Bogdanova, 2004; Brandon, 2013). Based on the available information and the decision to focus on protein regulation, it appeared that the 2004 project had better documentation of an events queue and so we based our events queue on this.

We began a table similar to a discrete events table that was used in the 2004 project. Their table included a parent cell, each of the two daughter cells, the axis for which the cell division occurred, and the time of division. However, the 2004 table did not include protein information and so our group adjusted for the additional data we researched. We incorporated a new function to the table that showed the gene expression for each cell. This portion of the table can be seen in Table 5-1. We looked at several different genes, as well as cell fate, and gave them a probability of being expressed in a given cell, organized by each daughter cell. Each column represents a cell fate and tracks expression while each row represents a parent cell. This allowed for each parent cell to carry information about cell fate for the two daughter cells.

For example, look to the value that occurs in the p-0 row and AB_Probability_D1 column. This piece of information records the expression probability of the AB cell fate in the first daughter (D1) of p-0. Specifically, the data shown below is for the wild type so the expression values are either 100%, for a 100% chance of demonstrating a characteristic, or 0%, for no chance of demonstrating a characteristic. As we progressed, we anticipated creating separate tables for mutant phenotypes where the probabilities would be more variable. This would ensure that our simulation was stochastic.

Table 5-1: Version 1 of Expression. This is a portion of the expression table that we first considered to represent a read-in to the simulation. There are six cell fates chosen and their values are given for daughter 1 (D1) and daughter 2 (D2). A value of 100 means that the given cell fate will be inherited into that specific daughter and a value of 0 means that a given cell fate will not be inherited.

Parent	AB_Probability_D1	P_Probability_D1	MS_Probability_D1	C_Probability_D1	E_Probability_D1	D_Probaility_D1	AB_Probability_D2	P_Probability_D2	MS_Probability_D2	C_Probability_D2	E_Probability_D2	D_Probaility_D2
p-0	100	0	0	0	0	0	0	100	100	100	100	100
ab	100	0	0	0	0	0	100	0	0	0	0	0
p-1	0	0	100	0	100	0	0	100	0	100	0	100
ab-a	100	0	0	0	0	0	100	0	0	0	0	0

After working on this table, we discussed some of the flaws of using it; mainly, it creates a large problem for expanding the table either by increasing the number of cell divisions, adding genetic information, or increasing the number of mutant phenotypes. Increasing the number of cell divisions would allow the simulation to include a larger time span of embryogenesis. This is useful so that researchers could learn more about how interactions change over time. However, this would mean that data needed to be manually entered for each of the cell fates for each new division.

Another development would be adding genetic information; this would increase the biological accuracy of the simulation. This would mean that information for all of the additional genes would need to be

manually entered for the current cell divisions. Likewise, an entirely new table would need to be constructed for each mutant phenotype to track the separate probabilities for expression. Ultimately, these developments would have to occur at some point in the future as the project progresses. We decided this table was not effective because it would be too difficult to grow with the project and the large table would not be efficient for the simulation's implementation.

5.3.2 Event Handling

Despite the impracticality of the previous method discussed, we decided to maintain the cell division portion of that table that tracks the parent cell, daughter cells, volume of daughter cells, time of division and axis of division (shown in Table 5-2). A priority of our group was maintaining the stochastic nature of the simulation. This implementation of events was instead deterministic. Ultimately, we decided that it was more important to create a stochastic handling of protein interactions and simplify the model by using deterministic lineaging.

Table 5-2 maintains important information about naming, visualization, and timing. The Sulston name for the parent is given in relation to the Sulston names of the two daughter cells. The daughter cells are assigned to be Daughter 1, for the most anterior or right cell, or Daughter 2, for the most posterior or left cell. For cell location purposes, we have the cell axis of division. Essentially, if a cell divides along the x-axis, then the two daughter cells will appear side-by-side along that axis. This is used to determine cell location rather than explicitly reading nuclei locations into the program. Our group felt it was important to explore axis of division as a location indicator instead of nuclei positions because it would make the simulation less deterministic in this respect.

The time of division is researched data on when the division of the parent cell occurs. The simulation time is 10 minutes added to the biological time to account for showing the first cell division; otherwise the simulation would start with the first two daughter cells present (division 1 of p-0 happening at time 0). The volumes of the daughter cells are represented as percentages of the parent where an even division of cellular volume is given as 50 to each daughter cell. Then, based on observation, we can adapt the proportions for unequal divisions such as 60/40, 30/70, 20/80, etc. These values would change for certain mutated genes based on published phenotypes.

Table 5-2: Daughter Cells, Division Times, and Volume Distribution. This shows the parent, each daughter, the division axis, time of division, the time adjusted by 10 minutes to account for the simulation time, and the two volumes by percentage.

Parent	Daughter1	Daughter2	Division Axis	Time of Division	Sim Time	V_Daughter1	V_Daughter2
p-0	ab	p-1	x	0	10	60	40
ab	ab-a	ab-p	x	17	27	50	50
p-1	ems	p-2	x	18	28	60	40
ab-a	ab-ar	ab-al	z	35	45	50	50
ab-p	ab-pr	ab-pl	z	35	45	50	50

In terms of protein regulation, we decided to look more abstractly at the data we wanted to represent. First, we used a program called Cytoscape (Shannon, 2003) which allows users to construct networks of interactions and nodes. This was primarily used to help us visually map out the protein interactions we

researched and understand how they were connected. An example of an interaction within the map is shown in Figure 5-1. Activation is denoted by an arrow, with the arrow's direction pointing to the activated gene. Inhibition is denoted by a "T" with the line intersection pointing to the protein being inhibited. For example, Figure 5-1 shows that PIE-1 inhibits both SKN-1 and PAL-1; additionally, SKN-1 inhibits PAL-1. In the cell this would equate to PIE-1 inhibiting the transcription of SKN-1 and PAL-1 and SKN-1 inhibiting the transcription of PAL-1 when PIE-1 is not present in the cell.

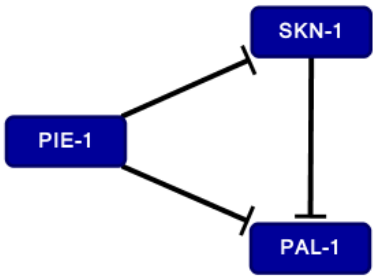


Figure 5-1: Protein interactions in Cytoscape (Adapted from Shannon, 2003). Protein interactions in the anterior involving PIE-1, SKN-1, and PAL-1.

We created an encompassing map that had all of the protein interactions we initially looked at, which helped us understand the complexity of the data we were trying to represent and helped us move toward a rule-based structure. We used the concept of antecedents and consequents to structure the rules. Antecedents are the initial condition that must be attained to see a change in the system and consequents are the actual change. For our project, the antecedents are the protein states that have an effect on another protein, and the consequent is the resulting change in another protein. Although the mechanism for how proteins can affect the state of another protein can vary, all of these types of interactions were grouped together as the same to simplify the rules for the simulation. For example, proteins can indirectly affect the other proteins' state through transcription or translation of a protein or directly interact with the protein through phosphorylation.

Table 5-3 shows a sample of the antecedents and consequents table. Upward pointing arrows show activation and downward pointing arrows show inhibition. In the first interaction, we can see that an activated PIE-1 and an activated PAL-1 result in an inhibited PAL-1. A notable trend in the table is that all of the antecedents contain the opposite state of the consequents. This was a change our group found necessary since, in the first interactions, the antecedent would be negligible if PAL-1 was already inhibited.

Table 5-3: Antecedents and Consequents. This shows the preceding states of proteins necessary to have an effect on other protein states.

Antecedents	Consequents	P or T	Start Stage	End Stage
pie-1 ↑, pal-1 ↑	pal-1 ↓	T	1	end
pie-1 ↑, skn-1 ↑	skn-1 ↓	T	1	end
skn-1 ↑, pal-1 ↑	pal-1 ↓	T	1	end

Additionally, the table we created as shown in Table 5-3, documents whether the change is through phosphorylation (P), transcription (T), or unknown (U). If a protein is a kinase and thus regulates through

phosphorylation, then the consequent would occur instantly since a physical change is happening to the protein. If the protein is regulated through transcription, then there is a delay in the consequent since the change is occurring through DNA expression. Otherwise, interactions that we could not find regulation data for are marked as unknown. Currently, our simulation does not take this information into account but it is compiled into the table for future development.

Each rule is applicable for different cell stages so we also kept track of the first cell stage for which the interaction would occur as well as the last cell stage. We considered using time to designate this interval but decided cell stage was more biologically accurate. Not only would this would help to maintain biological accuracy, but it would increase the efficiency of SimWorm14 during runtime. Otherwise, the program would have to check an antecedent even if the simulation timeframe is outside of when the interaction is relevant.

Finally, a third table for storing information about the genes initially present in p-0 was necessary. This table holds a complete list of genes currently modelled in our simulation, which currently is only composed of those genes whose behavior is very well understood. Each gene also has some data associated with it, such as whether it is active or inactive at the start of the simulation and its general location within the cell. Several example entries in this table are shown below.

Table 5-3: Genes incorporated into the simulation. This shows the initial status of the genes at the start of the simulation. A status of A indicates that the gene is active, while a status of I would indicate the gene was inactive.

Gene name	Status	Location
par-1	A	posterior
par-2	A	posterior
par-3	A	anterior
par-4	A	center
par-5	A	center
par-6	A	anterior
pkc-3	A	anterior
skn-1	A	posterior
pie-1	A	posterior
pal-1	A	posterior
mex-3	A	anterior
mex-5	A	anterior

5.4 Implementation

5.4.1 Creating Visuals Using Processing

Processing 2.0 (Fry, 2014), a free, open source software that has a java library to handle graphics, can easily be integrated into a program written in Java. It is compatible with Windows, Mac OS X, and Linux. The developers created a user friendly website to offer a wide range of examples and tutorials to help any individual from any background start using Processing (Fry, 2014). Consequently, our team decided to use Processing to visualize the simulation. The main goal of Processing's developers is to bridge the gap between visual artists and programmers and it has been implemented in a way that is easy to learn.

Because Processing is easy to learn, it allowed our team to continue program development without taking significant time to learn a new language. It was also ideal for the interdisciplinary nature of this group because Processing was designed with interdisciplinary development in mind.

Processing provides an abstraction for OpenGL, a more challenging graphics programming language. Because Processing is executing OpenGL code, it, like OpenGL, is an events-based language. This means that programs written in Processing have two distinct phases – a setup, in which anything that needs to be initialized one time occurs, and a main loop that the program repeatedly executes. This means that the program is, in essence, "waiting" for the majority of the time, but when an event is detected, such as a mouse click, code may be executed. It is up to the programmer to write what actions occur upon detecting an event.

Processing interacts very cleanly with Java, behaving in practice as though it were a Java library. It allows relatively easy and user-friendly rendering of three dimensional graphics within a Java project. Additionally, many other libraries have been built on top of Processing to add extra functionality in a way that is very easy for a programmer to implement. Several libraries for Processing played an integral role in developing the user interface. These are noted below.

5.4.2 Processing Libraries

Picking 0.2.1 is a library by Nicolas Clavaud that implements a feature called object picking (Clavaud, 2013). Object picking is the means by which a program detects when one of its components has been selected by the user. Picking in three dimensions is complicated because any number of objects can be located in one position in the two-dimensional space of the screen, and the program should select only the nearest one. Additionally, object picking within a view that uses a dynamic camera poses a challenge because it rules out the option of hard-coding correct behavior for each pixel of the screen; the view can change at any time. Usually these problems are solved by using a method called ray tracing. Ray tracing involves sending out a ray between the camera's point of view and the mouse cursor. This ray then reflects off the first object it hits and can report back information about the object.

Because the picking library has already been written, ray tracing should not need to be implemented from scratch for the project. However, at the current time, users have difficulty running programs that incorporate the Picking library outside of the dedicated Processing application.

Object picking will be used in the simulation to allow the user to select any cell on the screen by clicking on it in order to gain more information about it. At the present, a workaround exists that requires the user to type in the name of a cell they want to know more information about. Information about which genes are present in the cell and the state of each gene is displayed when a cell is requested.

Peasycam v201 is an implementation of a very powerful and user-friendly camera. It was written by Jonathan Feinberg. Peasycam requires only one line of code to set up, and can rotate 360 degrees, zoom in and out, and pan (Feinbeg, 2013).

ControlP5 2.0.4 is a library written by Andreas Schlegal that allows the programmer to easily divide up a view into multiple sections, and also provides many standard user interface features such as check boxes

or radio buttons. This is used extensively in our project, primarily to allow for a three dimensional section in which the worm is displayed and still maintain a two dimensional section for user controls to be housed. The use of ControlP5 was necessary to avoid having the user interface visuals affected by alterations to the camera in the three dimensional view. ControlP5 also provided all of the buttons that the user interacts with to choose settings (Schlegal, 2012).

The ControlP5 library as written contained a bug that caused programs to crash if the menu key is pressed. In order to make the SimWorm14 more robust, we altered the ControlP5 source to fix this issue.

5.4.3 Data Structures and Time Complexity

Every step along the way, efficiency was a top consideration when choosing how to store data. There is a large amount of data stored within the simulation, and on every time step, computations need to be made over many objects. Therefore, both spatial and temporal efficiency is key in preventing the simulation from running slowly. The time complexity of functions was minimized wherever possible.

A favorite data structure commonly used within the code was the hashmap. Hashmaps store a set of any kind of object, each of which is identified by a unique key that can also be of any data type. Hashmaps are a favorable way of storing sets because a query into a hashmap has constant time complexity.

Within the code, hashmaps were used whenever possible, not just because of their efficiency, but for the elegance of their application to the project. They were a logical choice for holding cells within the shell and genes within the cells, because each of these objects has a unique name that can serve as the key within the mapping.

5.4.4 Non-determinism

A major goal of the simulation was to set it up such that it is as self-determining as possible. That is, at every opportunity, subsequent events should be calculated rather than hard-coded. This required that we develop, as much as was possible, rules about behavior.

This non-deterministic philosophy was motivated by several factors. It would be fairly easy to observe how *C. elegans* behaves in the lab and then program a visual that progresses in the same way. However, this would defeat the purpose of creating a simulation. It would not provide any information that could not be easily observed in the lab, which is one major reason simulations exist. It would not be able to extrapolate future behavior beyond what has been observed previously. It also would be completely invariable and thus would not be able to demonstrate behavior that might occur when environment variables are altered. A major motivation for this non-deterministic philosophy was the goal to add the ability to mutate genes and watch how the development is affected.

It was not always possible to avoid hard-coding information into the simulation. For example, we wanted to match the timing of events such as cell divisions to observations made in the lab, but this timing doesn't seem to follow any mathematical pattern, so an events queue containing observed information was created. However, all such data is stored in structures that are easily altered such that a mutation that affects timing can still propagate changes over the events queue.

5.4.5 Extending the database

The project has been designed to be, in some ways, easily extensible by people without a highly technical background. There were three large sets of data involved in the simulation – these were as follows:

- The events queue, containing information about cell divisions, including when each cell divides, which axis it divides along, and what percentage of the volume is allocated to each daughter cell.
- The list of the genes present in the initial cell, p-0, and information about them, including whether they are active and their general location within the cell.
- The antecedent and consequent rules.

At its start, data points in each of these categories were placed into the appropriate data structures manually. This, while trivial for a programmer, was not something that many biologists who have interest in this simulation would be comfortable doing. So, to make adding to the simulation more accessible to laypeople, SimWorm14 was altered to read in these data points from excel spreadsheets instead. Now anyone who can enter the data into a spreadsheet in a precise fashion can test the effects of adding new genes or rules to the simulation.

5.4.6 Wildtype and Mutations

When SimWorm14 is initialized, an option is given to the user to mutate any of the *par* or *pkc-3* genes. If none of these are chosen, it will run a wildtype simulation. It will read in all of the rules from the excel spreadsheets as described in the previous section, and these will determine its behavior. Though the simulation's status at each time step will be calculated based on these rules and its status at the previous time step, there is no amount of randomness built into the wild-type simulation.

However, if the user chooses any of the cells to be mutated, several differences will arise. First, the spreadsheets will be read in as usual, but then some of the rules will be overwritten with mutation rules. Each gene that can be mutated has a list associated with it that describes the ways in which program behavior should change in the case of mutation (*par-3*, *par-6*, and *pkc-3* share a list because their mutations manifest in the same way). For example, if *par-1* is mutant, the following changes to the wildtype will be implemented:

- The protein SKN-1 is mislocalized, meaning that instead of staying confined to the posterior for the first division and then moving to the center as it would in the wildtype, in each cell it has a 90% chance of being distributed evenly throughout the cell, a 5% chance of being confined to the anterior, and a 5% chance of being confined to the posterior.
- PIE-1 degrades, meaning that it ceases to be present.
- PAR-3, MEX-3, and MEX-5 are mislocalized in much the same way as SKN-1.
- All cell divisions will be close to even, that is, each daughter cell will get about 50% of the volume. This can vary between 40-60%, and the exact value is determined randomly.
- All cells belonging to the same generation will divide at the same time.

Each of the other mutant genes has a similar list of effects associated with it.

5.5 Methods and Implementation Summary

Certain simplifications were made in order to focus on maintaining biological accuracy of the simulation. We focused on the cell divisions up to the 26-cell stage and incorporated protein interactions specific to

cell polarity. This is so we could gain useful information about cell fate in relation to mutations in the genes encoding for cell polarity proteins. We gained information for the proteins, such as interactions and cell stages for which these interactions are relevant, through literature research. The collected data was stored in a table that tracked antecedents and consequents. This table guides the protein interactions for the simulation.

The simulation was coded using Java and was aided by Processing (Fry, 2014), a software that is used to handle graphics and can act like a Java library. While creating the code, non-determinism was an essential consideration. Our team wanted to make the simulation as self-determining as possible, though some features such as cell divisions are hard-coded into the simulation.

Ultimately, the simulation was made to simulate both the wildtype and *par* or *pkc-3* mutants. While the values for the wildtype remain constant, a degree of variability was incorporated into the mutants. When a mutant is selected, certain rules are overridden to accommodate for a greater chance of mutation as well as alter the antecedents/consequents that were relevant to that gene. Each mutant has a list of effects associated with it.

6 Results

6.1 Laboratory Results

To test if the output from SimWorm14 is biologically accurate, the output from the simulation needs to be compared to real biological data. Our lab is interested in migration of neurons during the development of the nervous system. We thus created strains with transgenes that label a particular neuron, the CAN neuron, that migrates during embryogenesis. The strains also contain histone markers, to allow lineaging during embryogenesis. The initial simulation does not yet reach the developmental stage at which neuronal migration occurs; thus, comparison of the model to these strains is not yet possible, but is planned for future simulation versions.

To create this data, several strains of *C. elegans* were developed during the course of this project that contain several specific fluorescent markers (See Methods). Most of these strains contain the markers *pie-1::H2B::wcherry* and *his-72::HIS-72::wcherry* which mark the histones in the early embryo, and late embryo and larval stages, respectively (Figure 6-1). These markers are required for StarryNite to be able to track the nuclei positions of all of the cells in the different mutants. Once the nuclei positions have been tracked, AceTree will be able to create a 3D visualization of the nuclei positions in every cell stage and be able to track the lineage of each cell. In addition to the histone markers, most of the strains also contain a *ceh-10::GFP* fluorescent marker that labels the CAN neuron, which migrates during embryogenesis. This marker can also be tracked using *i*SPIM and StarryNite.

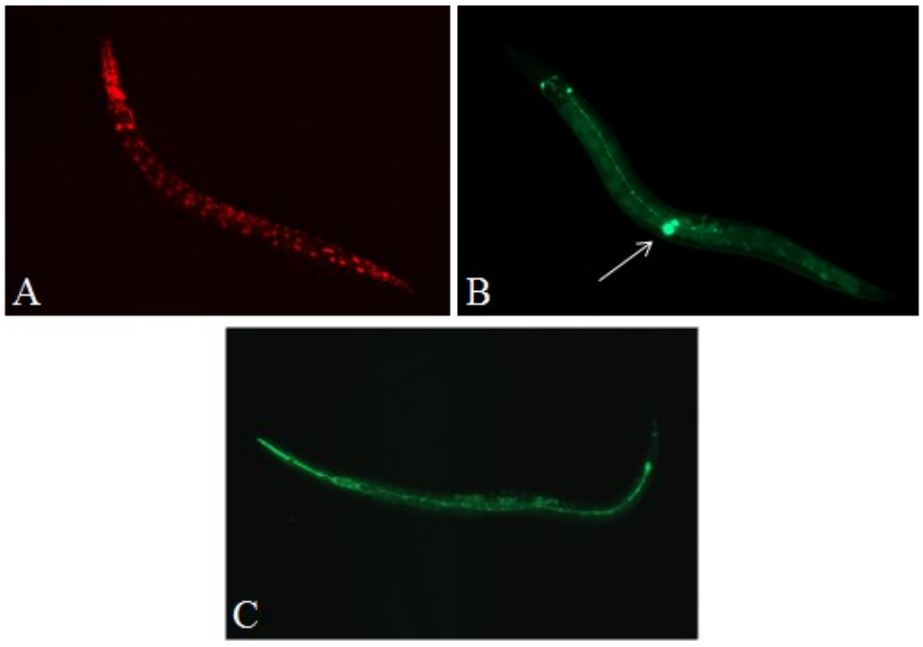


Figure 6-1: Fluorescently labeled strains constructed for simulation comparison. Photos were acquired on a compound fluorescent microscope and were taken with the anterior to the left and ventral down. (A) *Pie-1::H2B::wcherry* and *his-72::HIS-72::wcherry* were visualized in an RY1320 animal; the label shows the location of the nuclei in the worm. (B) *Ceh-10::GFP* was visualized in an RY1321 animal with a white arrow indicating the location of the CAN neuron. (C) *Dlg-1::GFP* was visualized in an RY1328 animal. All worms were at the L4 stage when the photos were taken.

The strains that contain these three markers are RY1320, RY1321, RY1323, and RY1325. In addition, RY1320 also contains the marker *pgp-12::GFP* which fluoresces in the excretory cell and was used to

help determine when *mig-10* was homozygous in the strain (*mig-10* mutants have a truncated excretory cell). The four strains that include the histone and CAN markers also each contain a mutation that affects neuronal migration during development of the nervous system (*mig-10*, *unc-34*, and *abi-1*). These strains were developed because our lab is interested in early neurodevelopment. After these strains have been imaged and analyzed, the data they provide can be used to show if the mutations created by SimWorm14 accurately portray the changes in the neuron migration in the real mutants.

There was also one strain, RY1328, which was developed that only contains the marker *dlg-1::GFP* and the *mig-10* mutation. *Dlg-1::GFP* marks the membrane of epidermal cells and will be used to create data to help identify the changes in cell shape in the *mig-10* mutant. The *dlg-1::GFP* marker may be added to other mutant strains in the future. In the embryo the green fluorescence appears between the cells and highlights the boundaries between them.

6.2 Biological Rules Developed

The biological rules in SimWorm14 were created from molecular interactions that have been recorded in multiple journal articles. In total approximately 40 antecedents and consequents were created and, out of these, 18 are currently used in SimWorm14. The antecedents and consequents that are not included in the simulation involve proteins that are in pathways that have intercellular interactions. For simplification purposes, our project only incudes intracellular interactions since it would be more complex to differentiate between intracellular interactions and intercellular interactions. In addition, normal intercellular interactions do not occur in the simulation at the moment, because cell locations are not yet accurate. SimWorm14 tracks the cellular locations of 12 proteins. The division times and percent volumes of the daughter cells up to the 26-cell stage are included as well.

In addition to the wildtype, rules for protein mislocalization, cell division timing, and cell volumes for the *par-1* through *6* and *pkc-3* mutants are included. Since mutations can have a variable effect on *C. elegans* phenotype, there is variability built into the protein localization, cell division timing, and cell volumes, as described in 5.4.6. While the real life wildtype has slight variability in phenotype as well, the simulation only includes variability for the mutant phenotype. This is because the mutant phenotype has more extreme variability than the wildtype.

6.3 SimWorm14

The following section outlines key results regarding the present simulation, SimWorm14, including details for the interface as well as output structure.

6.3.1 SimWorm14 Interface

The screen presented to the user upon first starting SimWorm14 is a simple menu in which a user chooses which genes, if any, to mutate prior to running. The mutant genes that SimWorm14 supports are *par-1, par-2, par-3, par-4, par-5, par-6* and *pkc-3* (Figure 6-2). After the user makes a selection, the main screen is drawn and remains for the rest of the simulation. Although users can select multiple mutants at a time, only single mutants have been thoroughly tested and deemed relatively accurate.

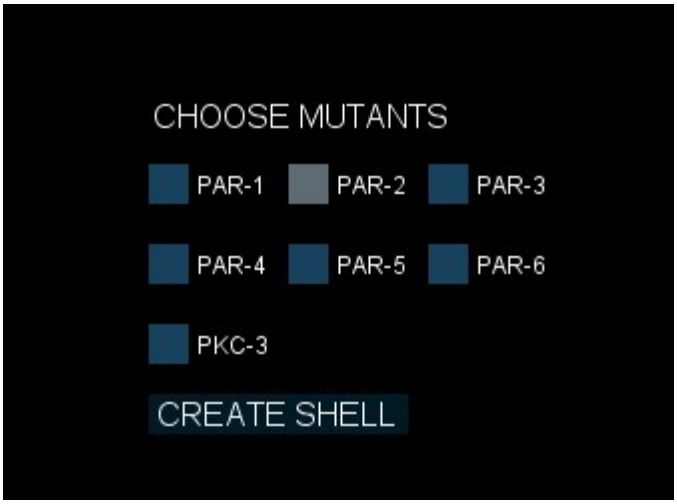


Figure 6-2: Mutant selection menu. In this menu, users choose which mutant they want to simulate. *Par-2* is selected as indicated by a different shade selection box. Users can select as many of the options as they want to mutate, or they can also choose not to select any of the mutants and therefore produce a simulation of the wildtype. Once the desired selections have been made, users press "create shell" to bring them to the simulation rendering.

The main screen is divided into two distinct sections – a three-dimensional view of the current state of the shell taking up the majority of the screen on the left (Figure 6-3), and a two-dimensional panel on the right (Figure 6-4) in which user input/output occurs.

The left side displays ellipsoids representing the location and spans of each cell within the shell. Each of these cells is color coded in a particular way as explained in section 6.3.2. In the three-dimensional view, the user can left-click and drag anywhere on the screen to rotate the camera around the cell 360 degrees, right-click and drag to zoom in or out, and middle-click and drag to pan. This functionality is made possible by the Peasycam library (Feinberg, 2013). A set of labelled coordinate axes is pictured to help the user stay oriented during rotations.

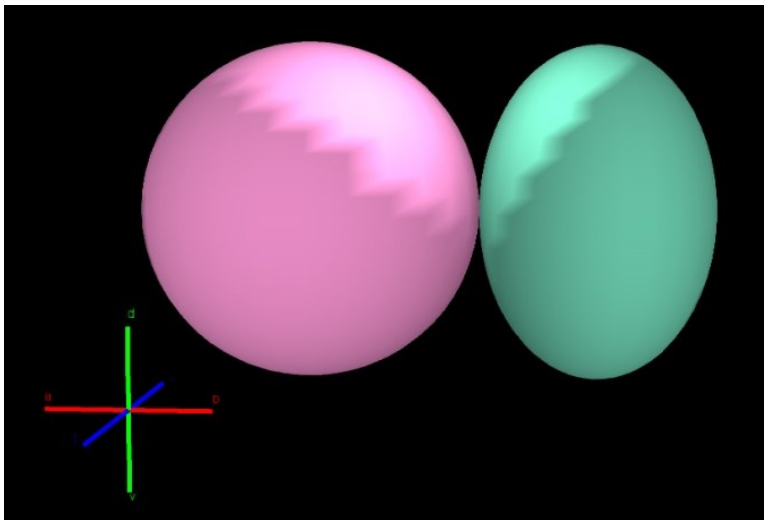


Figure 6-3: The left side of the display during the two-cell stage. Cells are represented as ellipsoids and can be viewed from any angle. Labelled coordinate axes can rotate with the rendering in order to show the user the orientation of the cells.

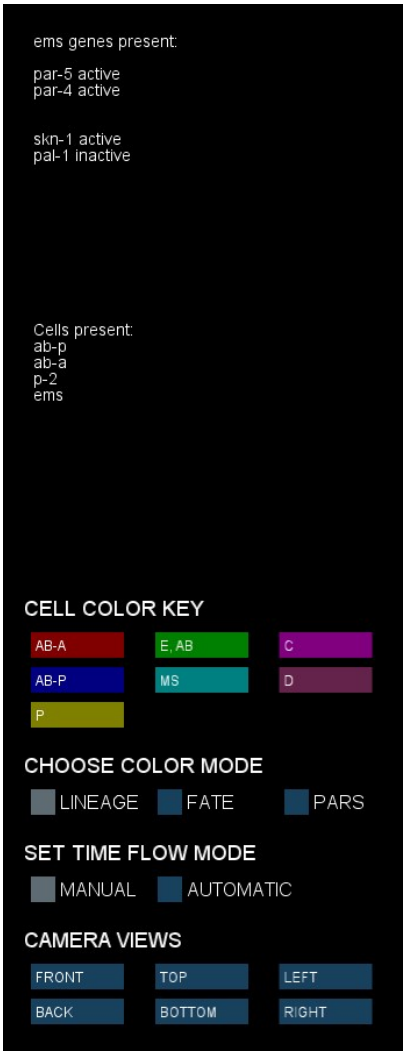


Figure 6-4: An example of what the interaction pane looks like during the simulation.

The right side contains interactive controls (Figure 6-4). The top of the screen shows a string of text that serves as the input and output. At any time, the user may type and their keystrokes will be displayed there. If the user types the name of a cell that is currently present on the screen and presses enter, the text will be replaced to show the list of proteins present in that cell and their states (active or inactive). As this list can be quite lengthy, most of the vertical space on this menu exists to accommodate this feature. However, if the list has to extend beyond the space allocated for it, the list is contained within an element that automatically gains scroll bars when they become necessary. This is the solution that currently exists to circumvent the issues we faced implementing object picking as described in section 5.4.2.

The first set of boxes below the text area serve as a color key that communicates to the user what the different colors of cells represent. The legend changes based on which color mode is selected. Below the color key are the radio buttons that the user can use to select a different color mode.

Another set of radio buttons controls the time flow mode of the simulation. If it is set to manual, the user can press right arrow to progress forward by one time step or left arrow to rewind by one time step. If it is

set to automatic, the program will measure elapsed time and periodically perform a time step automatically.

Finally, the interaction pane contains six buttons that can be used to quickly change the camera to each of the six orthogonal views – front, back, top, bottom, left, and right.

6.3.2 SimWorm14 Outputs

Beyond what is immediately visible to the user when running SimWorm14, many calculations are occurring at each time step. These functions are all documented in the javadocs (Appendix E), but what follows is a short description of how SimWorm14 runs and what it calculates.

As described in section 5.4.1, our project, as a Processing application, is events-based and thus contains a set up function and a main drawing loop. The set up function executes once at the start of the program. It simply draws the menu in which users can choose mutants for this run, then waits for the confirmation button to be clicked. When this happens, the second phase of set up occurs, which is more complex:

1. The user's choice of mutant genes are read and stored.
2. The shell constructor is called.
    a. The events queue spreadsheet is parsed and its information is stored.
    b. The first cell, p-0, is created.
        i. p-0 is given dimensions 50, 30, 30.
        ii. The genes spreadsheet is parsed and the genes present in it are assigned to p-0.
        iii. The antecedents and consequent spreadsheet is parsed. Genes have a list of relevant rules, meaning those rules for which the gene is an antecedent. Each gene in p-0 iterates through the rules and populates its own list.
        iv. For each mutant gene that the user chose, the mutant rules are applied.
        v. p-0's color is calculated according to the currently selected color mode.
        vi. p-0 is drawn to the screen.
3. The coordinate axes and interactions panel are drawn to the screen.

SimWorm14 then enters its main loop, which it will stay in for the rest of execution. This loop is waiting for one of several things to happen:

- The user clicks a mouse button.
    1 This will execute Peasycam code to rotate, zoom, or pan the camera view if it occurs in the three-dimensional view.
    2 In the two-dimensional view, mouse clicks trigger different events based on whether the object clicked was a button, and what button it was.
        a If one of the buttons for altering the camera view was clicked, the new camera location will be calculated and it will be set there.
        b If a new color mode was chosen, the new color for each cell will be calculated according to the newly selected mode's algorithm and the cells will be redrawn with their new colors.
            i For the lineage color mode, cell color is determined by the founder cell from which it is descended. These colors were chosen to match

the AceTree application, so that direct comparisons would be easier to make (Boyl, 2006).

ii For the fate color mode, cells are colored based on which fate it fulfills. A cell fulfills a fate if it contains certain combinations of active genes.

iii For the PAR color mode, cell colors are an additive combination of which PAR proteins they contain, with each PAR represented by a different color. This helps track the movements of these important proteins.

c If a new time flow mode was chosen, the corresponding rules for time flow will be set

i For the manual time flow mode, nothing happens when the program is idle. The program simply waits for the user's next action.

ii For the automatic time flow mode, when the program is idle it counts elapsed time. After a certain amount of time, a time step is triggered.

- The user presses a keyboard key.

1 If the key is enter, the string printed on the interactions pane is checked for equivalence with any of the cell names present in the shell at the time. If there is a match, that cell's genes and states are printed to the screen, otherwise, an error message is displayed.

2 If the key is right arrow, a time step occurs.

a The list of antecedent and consequent rules is updated to see if any new rules have become active.

b In each cell, for each gene, the list of relevant rules is updated, then each rule is checked to see its conditions are fulfilled. If so, a change to the gene is queued to occur after all the other genes have been checked, so as to prevent inaccuracies caused by concurrent modification.

c After all of the genes have been checked for changes, the changes are propagated.

d The events queue is checked to see if any divisions occur on this time step.

i If a division occurs, the names of the two daughter cells are calculated based on the name of the dividing cell.

ii Then the new dimensions (center point, x/y/z diameters) of the daughter cells are calculated. The daughter cells only differ from the parent cell in the axis of division.

iii The genes inherited by each daughter are calculated, based on the compartment of the parent gene in which genes were located.

iv The colors of the daughter genes are calculated based on the currently selected color mode.

v The parent cell is removed from the list of cells and the daughter cells are added to it.

e A deep clone of the shell is created and stored in a hashmap that associates it with the integer value of the current time step. This allows the state of the

shell at this time step to be saved and accessed when the simulation moves backwards in time.

f The cells are drawn to the screen.

3 If the key is left arrow, a backward time step occurs.

a The integer value of the previous time step is requested from the hashmap holding the shell clones. Its associated shell is made active and drawn to the screen.

4 If the key is anything else, that letter is added to the string printed on the interactions pane.

6.4 Molecular Reality

SimWorm14 treats each interaction as if each protein was directly interacting with the other proteins. In reality, some proteins like PIE-1 actually inhibit the transcription of mRNA of the proteins they affect, and don't actually interact with the proteins directly (Reese, 2000). For our purposes at this time in the simulation creation process, direct and indirect inhibition or activation of a protein is treated exactly the same way in our rules. This was done to keep the simulation rules as simple as possible and make progress towards visually showing how different interactions affect cell fate.

6.4.1 Gene Expression and Protein Location

SimWorm14 currently models the behavior of a small subset of the genes that are expressed at the beginning of embryogenesis (Figure 6-5); we limited ourselves to those genes whose behavior is well understood and which produce proteins that are mainly involved in intracellular interactions. Rules regarding the state of these genes are set at the beginning of SimWorm14 and dictate the way gene expression progresses over time. The cells contain anterior, posterior, and center compartments that can determine where certain proteins will be passed during the cell division; if the protein is located in the anterior compartment it will be passed on to the anterior daughter cell and if it is located in the posterior compartment it will be passed on to the posterior daughter cell. If the protein is in the center compartment it will be passed on to both daughter cells. In the biological cell, the proteins are usually inherited at different concentrations rather than all or nothing, but at this time there is not enough information to get accurate protein concentration inheritance for most of the proteins. Instead of arbitrarily assigning concentrations, the daughter cells that normally inherit low levels of protein inherit no protein in the simulation. This will need to be modified when SimWorm14 becomes more complex and realistic, but for the extent the project is right now this "all or nothing" inheritance produces relatively accurate results.

For example, in the wildtype, P-0 has all of the proteins active in order to initialize the protein interactions. The proteins do not begin interacting with each other until after the first division. When P-0 divides into AB and P-1, the AB daughter cell inherits PAR-3, 4, 5, and 6, MEX-5/6, and MEX-3 (all active). P-1 inherits PAR-1, 2, 4, and 5, and PIE-1 (all active), and also inherits SKN-1, and PAL-1 (both inactive). Because PIE-1 was inherited by P-1, SKN-1 and PAL-1, which were also inherited from P-0, are inactive (See Figure 6-5). AB then splits into ABa and ABp which inherit the same proteins and protein states as the parent. There is no change because there are no interactions that change the protein states based on those that are present. P-1 divides into P-2, which inherits all of the proteins and protein states from P-1, and EMS. EMS inherits PAR-4, 5, and 6 and SKN-1 (all active) while PAL-1 is inactive. Because PIE-1 is in the posterior compartment of P-1, only its posterior daughter cell (P-2) inherits this protein. The change in state for SKN-1 is brought by the absence of PIE-1. These interactions can clearly

be seen in Figure 6-5 and show how the protein interactions directly relate to the proteins present in each cell in SimWorm14.

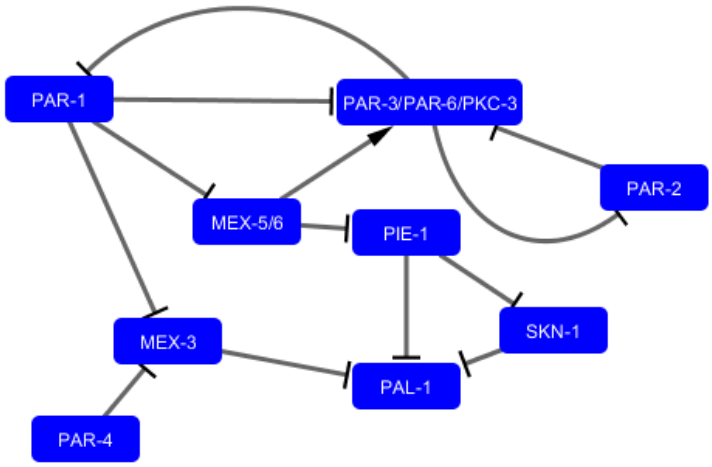


Figure 6-5: Protein map of the current protein interactions incorporated into SimWorm14 (Adapted from Shannon, 2003). These interactions are relevant for proteins that are present in the same cell. Additional rules are mapped in Appendix B.

One of the outcomes we tracked during SimWorm14 was cell fate, which is determined by which genes are expressed and consequently which proteins are present in those cells. Certain combinations of active genes in a cell indicate that that cell is expressing a particular cell fate. After programming in the known set of antecedent and consequent rules, as well as what is known about the way the proteins are inherited from parent to daughter cells during cell division, all of the cells up to the 26 cell stage, which is the scope of our project, have expressed the expected cell fate (Figure 6-6).

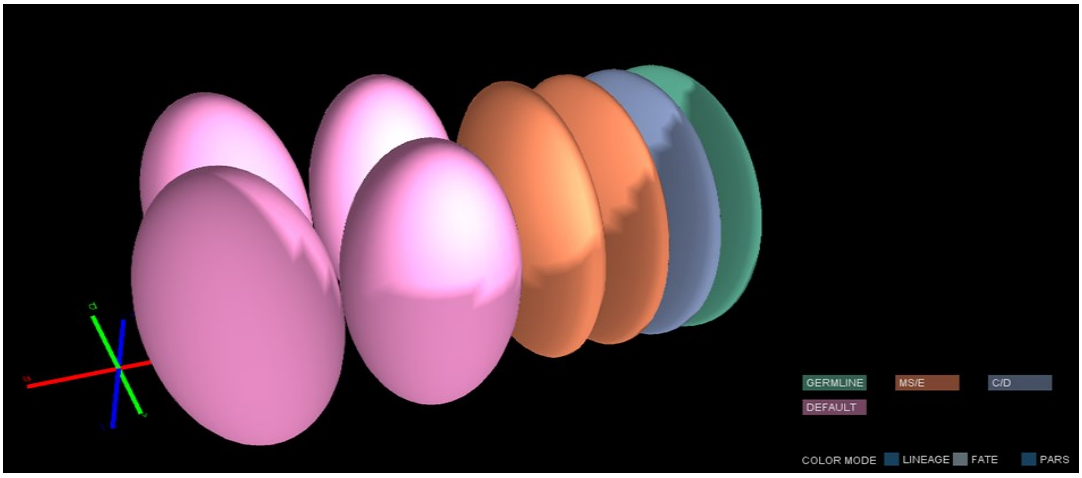


Figure 6-6: SimWorm14 under cell fate color mode at the eight cell stage. Each of the cells is expressing the biologically expected cell fate.

SimWorm14 takes four different cell fates into account at this time: germline, C/D, MS/E, and a default cell fate that correlates with the AB cell fate in the wildtype. The rules that account for these cell fates are the presence of PIE-1 for germline, SKN-1 for MS/E, PAL-1 for C/D, and none of these proteins present for the default. Although these proteins are required for these cell fates in the organism, this is a major

simplification of what actually occurs in the *C. elegans*. There are other factors that differentiate the MS and E cell fates from each other, C and D cell fates from each other, and genes that must be expressed in the AB lineage. Despite this extreme simplification, the cell fates appear to propagate accurately for the wildtype in the simulation.

6.4.2 *par* Mutants

To simulate *par* mutants, certain rules are put into place that override the wildtype rules. The first rule that is put into place is that the gene that is mutated is no longer in the simulation and any rules from the antecedents and consequents that require this gene are no longer applied to the simulation. The rules regarding the cell volume size and division timing are also overwritten and replaced with the rules that are consistent with the phenotype of the mutant. These rule changes allow for a certain degree of variability as mutants have varying effects on these properties, as described in 5.4.6. For the *par* mutants, certain proteins become mislocalized so the simulation takes this into account as well.

To see if our rules were being propagated accurately, we ran SimWorm14 multiple times with the different proteins selected for mutations (Figure 6-7). We found that the mutations were propagated accurately and had varying effects when run multiple times. The mutations were even accurate to the phenotypes described in the literature. For example, *par-1* mutations result in excess body wall and pharyngeal muscle cells which are normally produced by the MS lineage (Guo, 1995). Our simulated *par-1* mutant has excess MS/E lineage which correlates to this finding (Figure 6-8).

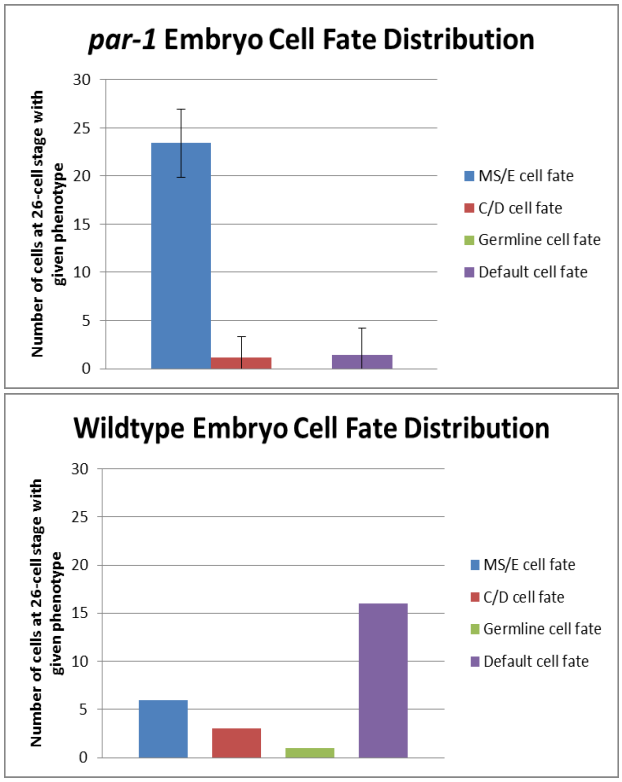


Figure 6-7: Cell fate distribution at the 26 cell stage over 1000 mutant embryos. After running a simulation of a *par-1* mutant 1000 times, it was found that the average fate distribution at the 26 cell stage is 23.427 (90.1%) MS/E fate cells, 1.141 (4.4%) C/D fate cells, 0 (0%) germline fate cells, and 1.432 (5.5%) default fate cells. The above graph shows these averages with standard deviations for the mutant and the wildtype. The wildtype has no variability, so the standard deviation is 0 for all cell fates.

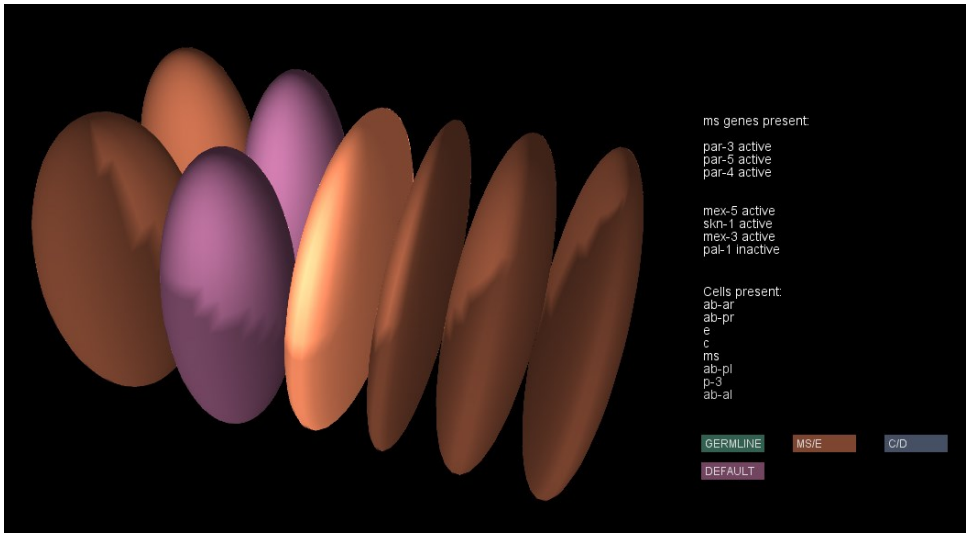


Figure 6-8: Screenshot of *par-1* mutant at the 8-cell stage. As expected based on literature findings, there is an excess of MS/E lineage cells.

This lineage change is due to the mislocalization of SKN-1 which was explained earlier results in the MS/E fate.

6.5 Visualization Results

Visualization was an important consideration for this project even though we were focusing on protein interactions. Consequently, we took necessary steps to make SimWorm14 as visually accurate as possible given the timeframe of the project and simplifications that we decided to make. The following section outlines the progress that was made in regards to cell shape and volume as well as cell movement.

6.5.1 Cell Shape and Volume

Cell shapes are difficult to simulate because they cannot be defined by a simple, three-dimensional shape. In a *C. elegans* embryo, cells are encased in a shell and take on shapes that roughly fill the volume of the shell. The cells have rounded edges but form blobby shapes as they move and divide to form the embryo. Our team made a few simplifications to make cell shape easier to manage in the timeframe we had. The first was for cell divisions to occur at specific time points and not show the division process. As shown in Figure 6-9, when cells divide, the cytoplasm of the parent cell separates into each side of the cell as the two daughter cells form. The complex cell shapes during the division process contributed to the decision to not show the cell division process.

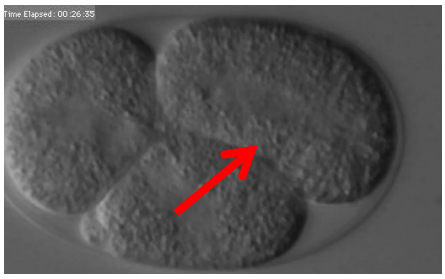


Figure 6-9: Cell division leading to the four cell stage (K. Kemphues, personal email, February 11, 2014). The red arrow points to the cell that is dividing. A notable characteristic is that the cytoplasm is being separated to either side of the cell as two daughter cells begin to form.

The second simplification we made is that cells are represented as ellipsoids, or three-dimensional ellipses, which are generated as a result of the parent cell's volume. SimWorm14 begins with a single ellipsoid to represent the first cell. As cells divide, the volumes of daughter cells are determined based on proportions of the parent cell's volume. This proportional splitting of volumes is related to a proportional splitting of the parent cell's diameter. For example, when the first cell division occurs, the split volumes for the two daughter cells should represent 60% of the parent cell's volume in one daughter and 40% of the parent cell's volume in the other daughter. Consequently, the diameters of the daughter cells will represent 60% and 40% of the parent cell's diameter, respectively, as shown in Figure 6-10.

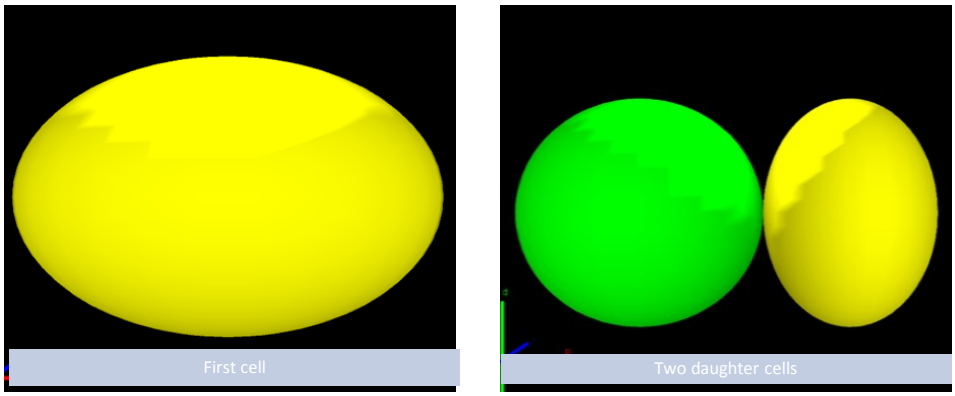


Figure 6-10: First cell (left) and the resulting two daughter cells (right). The volumes in the daughter cells are representative of 60% and 40% of the parent cell's volume. The diameters are 60% and 40% of the parent cell as well.

Because the cell's shape is predominately changing by diameter, it causes the simulation cells to form shapes that are not biologically accurate; the diameters become smaller and smaller forming pancake-shaped cells. This becomes notable at the four cell stage (Figure 6-11).

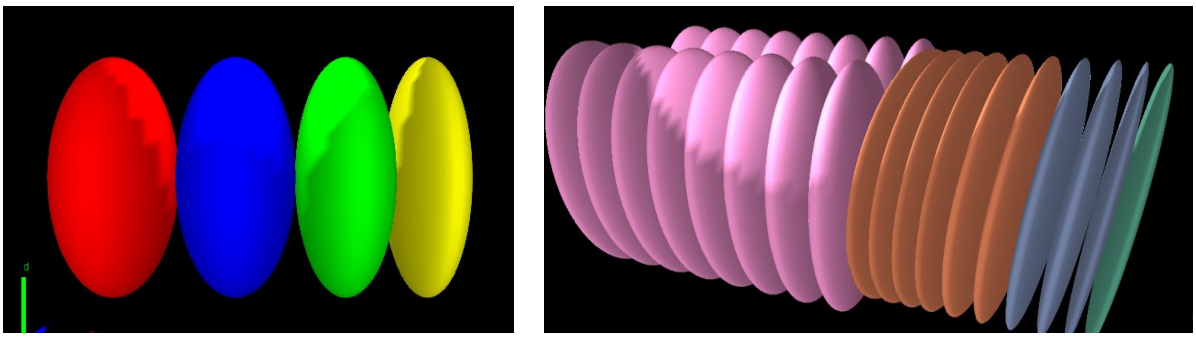


Figure 6-11: Cell shapes at the four (left) and 26-cell (right) stages. The diameters of the cells become smaller with each division while the height remains the same causing "flat" or "pancake" looking cells.

Despite the differences between SimWorm14 and biological cell shapes, the cell volumes are relatively accurate. It is challenging to assess the volumes based on visual comparisons because volume observation is largely based on shape, but given an early comparison of cell volumes (Figure 6-12) and comparing literature data to the simulation's input, we can see that volumes are relatively accurate.

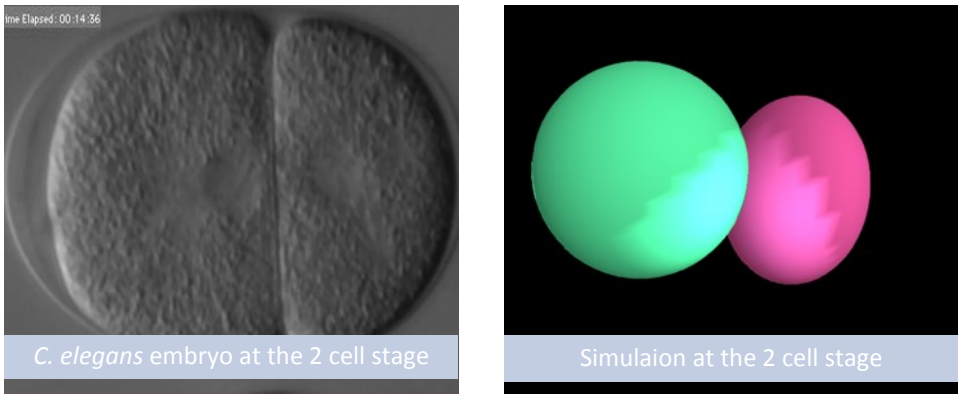


Figure 6-12: Comparison of cell volumes between the embryo and SimWorm14 (K. Kemphues, personal email, February 11, 2014). Comparing the two images reveals that the left cell has a larger volume than the right cell.

6.5.2 Cell Movement and Location

Although cell migration does not occur until gastrulation, cells still exhibit small movement within the shell as subsequent divisions cause the cell to fill the available area. This movement is complex and based on cellular collisions; the cells move as their shape changes during division. This movement is demonstrated in Figure 6-13 where the first two cells are pushed upward by a third cell.

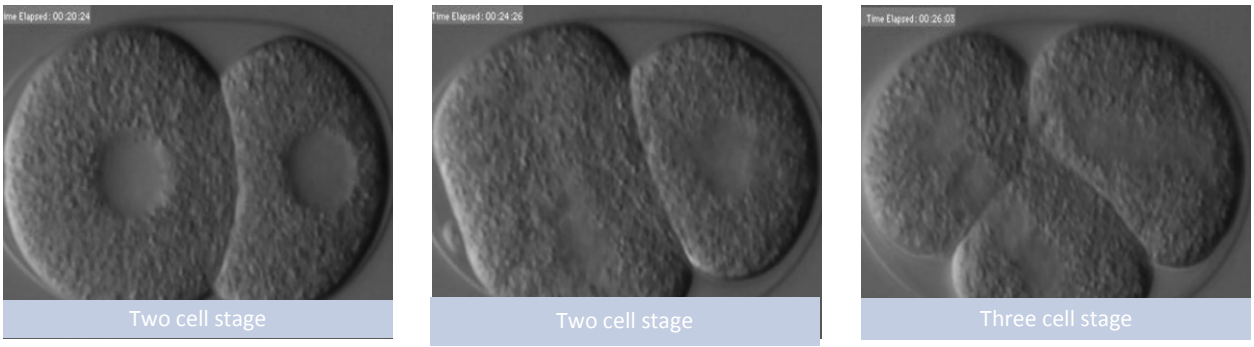


Figure 6-13: Transitioning from the two cell stage to the three cell stage (K. Kemphues, personal email, February 11, 2014). In the first image, the two cells are along the same axis. However, in the second image, as cell division occurs, the cells move within the shell. The third image shows the new cell and the original two cells in their new locations.

In order to maintain the stochastic nature of SimWorm14, our group did not want to read the cell locations directly into the simulation. However, as demonstrated above, the divisions and movements of cells are very complicated. To simplify the concept of cellular movement during division, SimWorm14 only takes into account the axis of division for a split. There are three axes: the x-axis, the y-axis, and the z-axis. Once a cell divides, the locations of the daughter cells are calculated based on the division axis. The daughter cells do not move after they are generated. Consequently, this causes some of the cell locations to be relatively inaccurate, especially as the number of divisions increase.

Figure 6-14 shows a comparison of SimWorm14 for the wildtype and AceTree (Boyl, 2006) nuclei positions of the wildtype through the eight cell stage. Note that, while AceTree shows the nuclei positions, SimWorm14 shows the entire cell. This means that the AceTree rendering shows smaller spheres that are not connected while SimWorm14 shows ellipsoids that are spatially closer.

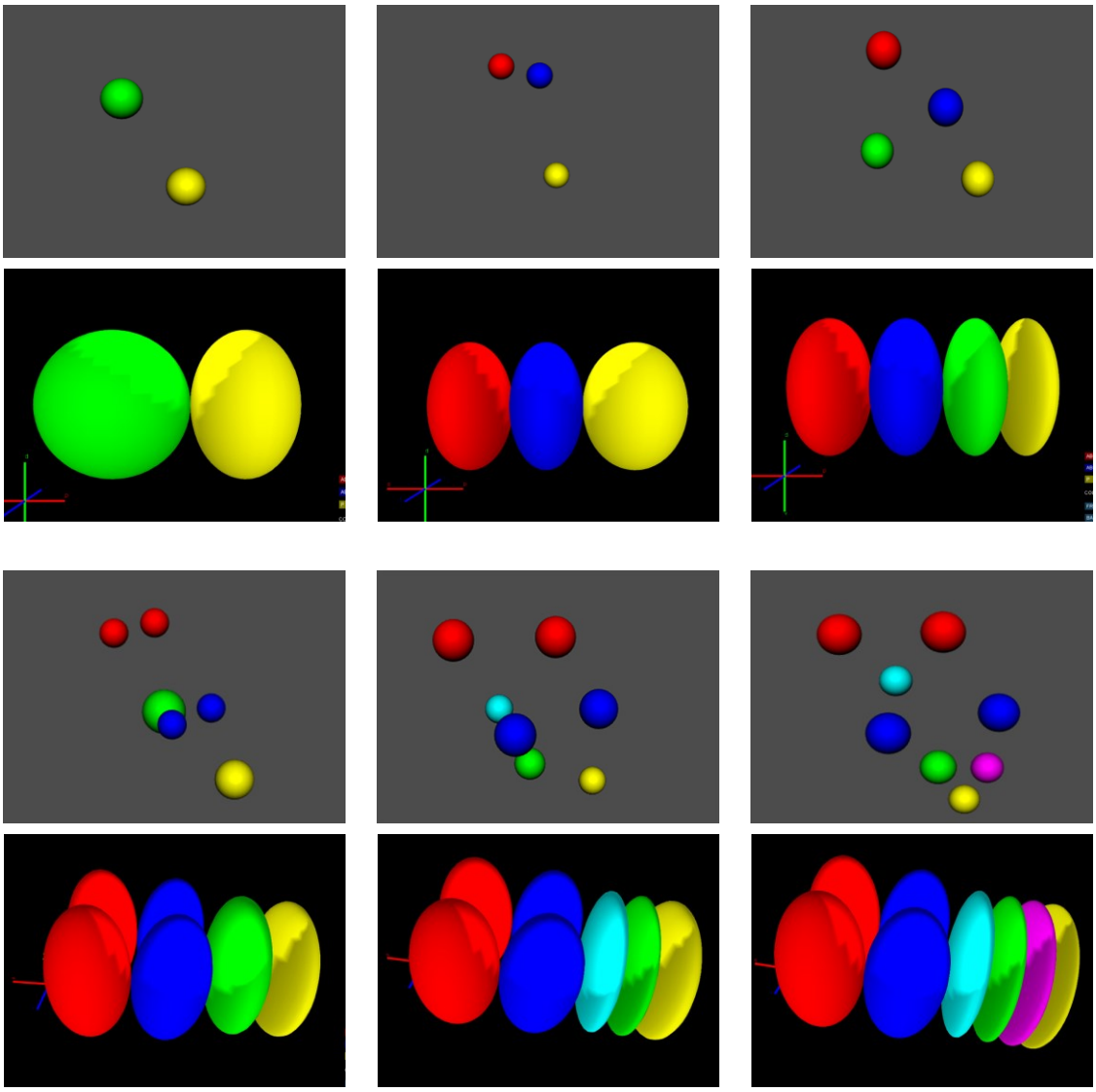


Figure 6-14: AceTree 3D rendering of nuclei compared to SimWorm14 through the eight cell stage (Adapted from Boyl, 2006). The cell coloring is indicative of cell lineage and matches between AceTree and the simulation for easy comparison.

At the two cell stage, it can be seen that there is some difference in positioning between the AceTree rendering and SimWorm14. However, we can still see the basic relationship of two cells that are next to each other. This is similar for the three cell stage. However, by the four cell stage, SimWorm14 shows the cells all along the same axis whereas the AceTree rendering shows more of a cluster. This is a great example where small movements between cell divisions affect cell locations. For example, Figure 6-15 shows the different positions of nuclei at the four cell stage as time progresses.

The six cell stage is a product of a split along the z-axis and is the first non-x-axis division. Although the cell positions are not completely accurate, SimWorm14 clearly maintains some of the relationships between cells. For example, the six cell stage is a product of ABa and ABp splitting at the same time to form adjacent daughter cells. This is clearly reflected in SimWorm14. The eight cell stage still reflects the

basic locations of the shells but has some clear differences in the orientation of cells along the x-axis. This is a result of the movement of previous cells during cell divisions.

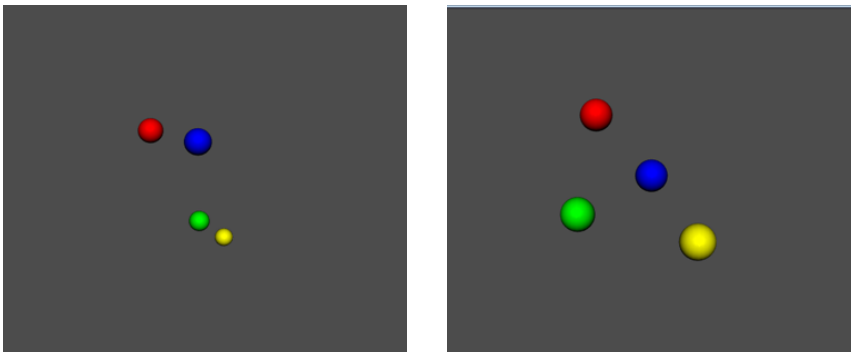


Figure 6-15: AceTree rendering of nuclei at the four cell stage (Adapted from Boyl, 2006). The left figure was taken right after cell division and the right figure was taken right before the subsequent cell division. This clearly demonstrates the small movements that occur between cell divisions.

SimWorm14 has certain strengths and weaknesses when it comes to cell movement and location. Because cells do not move between divisions, the difference in location between SimWorm14 and the AceTree renderings are augmented as the number of divisions increase. However, up until the eight cell stage, the basic relationships between cells are apparent and provide an adequate basis for making future improvements to have more complex cell movements.

6.6 Comparison to Previous Research

6.6.1 SimWorm13

This project is a continuation of previous work done at Worcester Polytechnic Institute to simulate *C. elegans*. The project completed as of last year, in 2013 (Brandon, 2013), focused on creating an accurate portrayal of *C. elegans* up to the first 100 cells. Specifically, the authors wanted to track the migration of the CAN neuron. They also focused on creating visually accurate cell shapes by using metaballs and marching cubes, two computer science concepts that help programmers make complex shapes.

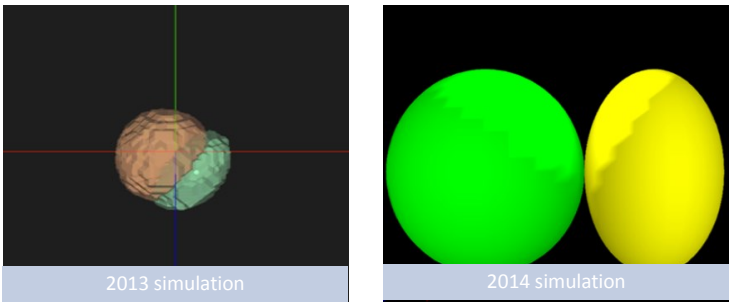


Figure 6-16: Comparison of the 2013 simulation and SimWorm14 (Adapted from Brandon, 2013). The 2013 simulation placed a heavy emphasis on visualization and used metaballs, a computer visualization technique, to simulate the complex shape of cells. SimWorm14 used Processing to create the visuals and the cells were made as ellipsoids.

This differs from our project because we focused on the early stages of embryogenesis and, while we wanted to make visually accurate cells an important consideration, we mainly focused on PAR proteins

and their effects on cell fate and polarization. Additionally, we incorporated the ability to make mutations in *par* genes so that users could test the effects of a particular mutation.

Another difference between the two projects is that we strove to make SimWorm14 as stochastic as possible. This caused the way our group handled cell divisions and movement to be very different from the previous group. In the 2013 simulation, cell positions were read in from laboratory data that designated the nuclei positions. Our group, however, did not want to explicitly tell cells where to be located in the embryo and developed an algorithm that determined cell positions based on the axis for which they divide.

Overall, the two projects aimed to simulate *C. elegans* in a way that would be useful for education and research. However, each project prioritized different proteins of interest and visualization components. They are both steps in a collaborative effort to get a more holistic view of *C. elegans* embryogenesis.

6.6.2 Other *C. elegans* Simulations

There are two main simulations that are relevant to *C. elegans*: The Perfect *C. elegans* Project (Ketano, 1998) and OpenWorm (Idili, 2011). Each of these projects are efforts of researchers to understand *C. elegans* at a deeper level by narrowly focusing on one area of interest at a time. This narrowing of scope is comparable to the development of our project since we focused on early embryogenesis and the proteins that affect cell polarity during the beginning cell stages.

The Perfect *C. elegans* Project, though interested in neural network simulation, focused mainly on visualization of the cells within a shell. In order to compute cell positions, there was some biological data given to indicate initial positions. Additional computational methods were applied to improve the biological accuracy of these cell positions such as pushing and collision forces between cells. The movement is simulated using algorithms to emulate objects in a viscous fluid.

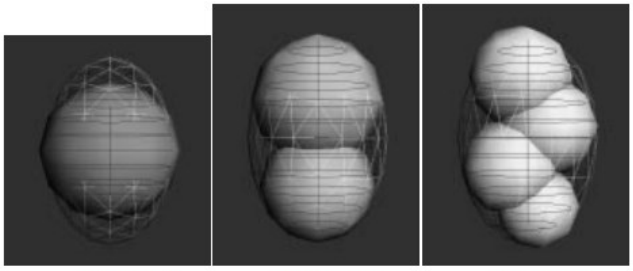


Figure 6-17: The Perfect *C. elegans* Project cells encased within the shell (Adapted from Ketano, 1998).

In comparison, SimWorm14 does not account for any biological positioning data or collision forces. Instead, we use the axis of divisions to primarily dictate the positioning of subsequent cells. It is also indicated in (Kitano, 1998) that the next steps in the project would likely be to simulate cell fate. While there are no publicly available updates on the project, it can be seen that their approach was the opposite from our own. While developers of The Perfect *C. elegans* Project focused on visualization first and planned to simulate cell fate designation, our project used proteins to simulate cell fate and would like to make future improvements to cell shape.

OpenWorm is an ongoing effort to create an open source simulation of *C. elegans*. The researchers are focusing on specific aspects of *C. elegans*, such as movement and neurology, to eventually create a biologically complete simulation of *C. elegans*. The latest development for the project involves simulating the movement of the worm. Figure 6-18 shows an image of the simulation's complex movements of *C. elegans*.

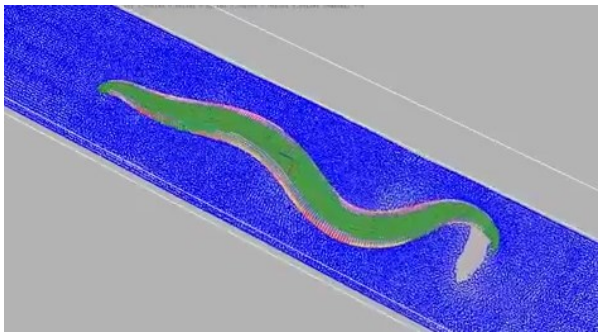


Figure 6-18: OpenWorm's simulation of *C. elegans* movement (Adapted from Idili, 2011).

While the approach of focusing on one aspect of *C. elegans* at a time is comparable to our approach in first focusing on protein interactions, the two projects are very different. OpenWorm began their simulation at a larger scale by modeling the whole worm, whereas our project starts at the cellular level. We also start with early embryogenesis while OpenWorm uses an adult worm.

Creating a simulation of *C. elegans* is a collaborative effort and, while there are several independent projects, much can be learned from each simulation. SimWorm14 aims to add to the progress of simulation by specifically tracking PAR proteins through early embryogenesis.

7 Conclusion

The goal of this project was to create a rules based simulation of early *C. elegans* embryogenesis from the first cell to the 26-cell stage. Currently, SimWorm14 is able to accurately show the cell divisions and timing up until this point. In particular, the goal was to create rules for how a subset of proteins interact and demonstrate how mutations in the genes that encode these proteins can affect cell fate. We were able to incorporate several proteins involved in embryogenesis cell polarity and create a simulation that allows a user to mutate the respective genes. The gene mutations visibly affect the mutant cell fate as can be shown through the cell fate coloring scheme. Additionally, other coloring schemes can be chosen to easily demonstrate protein distribution and lineage of each cell. SimWorm14 maintains stochastic features by utilizing probabilities to introduce random mutations in addition to the selected *par* mutation. An events queue is used to determine cell fates based on a cell's protein distribution.

In addition, we wanted to use this project to demonstrate the usefulness of simulations in the study of biological systems. SimWorm14 can be used to explore the *C. elegans* embryo using a dynamic camera that can view the embryo from any angle. Researchers can also easily check protein content within cells by typing in a cell's name which produces a list of proteins in the cell and whether they are active or inactive. Hypotheses can also be tested by testing different mutant genes against an expected phenotype.

7.1 Future Development

As with any simulation in progress, there are specific steps that should be taken to both improve the current state of SimWorm14 as well as incorporate additional features and data into the simulation. This section outlines the steps our group thinks are necessary to make in the near future to enhance the project.

The most visible inaccuracy of the current simulation is the cell shapes. These shapes are governed by complex physical rules involving the cells pushing against one another or against the shell. In the current simulation, newly divided cells simply appear side-by-side on the axis along which the division occurred, and cells are simple ellipsoid shapes. For the future, cell divisions should be handled in a more complex way that reflects the biologically accurate change in dimensions along all axes. Efforts should be made to draw the cells using metaballs, also known as "blobby objects," which are a primary method of depicting irregular three-dimensional shapes in computer graphics (Wyvill, 1990). This will allow the programmer to model the cell shapes that occur during complex interactions within the shell. Because the 2013 SimWorm project (Brandon, 2013) used metaballs to handle the complex shapes of cells, looking at this code will be useful in implementing the next stages of the SimWorm14 code. Additionally, the 2004 SimWorm project (Bogdanova, 2004) incorporated collision forces into their simulation and studying this code will be useful, in conjunction with metaballs, for creating biologically accurate cell movements.

Improving accuracy for cell shape and location will be important, because although our project incorporates intracellular gene interactions, intercellular interaction rules that exist in reality are not included at all. It is not useful to include these rules until correct cell placement is achieved because SimWorm14 does not currently accurately depict which cells are adjacent to other cells within the shell. Once biological cell position and cell boundaries are verified against SimWorm14, the interactions that occur between cells and mutants involving these interactions can be developed. Once these interactions can be simulated, more cell fates can be added to SimWorm14 and the simulation can go further in development, possibly past the 26-cell stage which is when migration of cells begins.

Our team took steps to help with future development regarding cellular interactions. There are currently 20 antecedents and consequents that have been developed and are not incorporated into SimWorm14. These rules were not included because they are involved in pathways that include cell-cell interactions. The two pathways that were studied but not included in SimWorm14 are the Notch and Wnt signaling pathways. Since these two pathways are involved in cell fate determination and are active at the beginning of embryogenesis, simulating the interactions and downstream effects from these pathways are the logical next steps in the project; these pathways and rules are included in Appendix B and C.

We also recommend that SimWorm14 remain user friendly. One update to the user interface that will be important to increasing the project's ease of use will be to troubleshoot and implement object picking. This will allow users to obtain information about cells by clicking on them in the three-dimensional view. Currently users have to type in the name of a cell in order to view its information, which is problematic because the users might not always know the name of the cell they are interested in. The right-hand panel does, however, provide a list of the names of the present cells so that users know what their options are. Also, the color scheming highlights the currently selected cell so that it is apparent which ellipsoid on the screen represents the cell about which the user has requested information.

We hope that SimWorm14 can be built upon in subsequent years so that it can be a useful research and learning tool. A complete simulation of *C. elegans* not only has the potential to aid researchers in efficiently directing their research but can also be used to teach interested individuals about the complex systems within *C. elegans*. Furthermore, we hope SimWorm14 can grow as a collaborative effort among the computational biology community and show the powerful benefits of simulation.

8 References

Aceto, D., Beers, M., & Kemphues, K. J. (2006). Interaction of PAR-6 with CDC-42 is required for maintenance but not establishment of PAR asymmetry in *C. elegans*. *Developmental Biology, 299*(2), 386-397.

Altun, Z. F. and Hall, D. H. (2006) Introduction to *C. elegans* Anatomy. Handbook of *C. elegans* Anatomy. In WormAtlas. http://www.wormatlas.org/ver1/handbook/anatomyintro/anatomyintro.htm

Ander, M. (2004). SmartCell, a framework to simulate cellular processes that combines stochastic approximation with diffusion and localisation: analysis of simplenetworks. In P. Beltrao (Ed.) (Vol. 1): Systems Biology.

Baik, J., & Rosania, G. R. (2013). Modeling and Simulation of Intracellular Drug Transport and Disposition Pathways with Virtual Cell. *J Pharm Pharmacol (Los Angel), 1*(1).

Bogdanova, N., Jajosky, J., Lloyd, N., & Stolzar, L. (2004). "Computer Simulation of *C. elegans* Embryogenesis", *Major Qualifying Project*: Worcester Polytechnic Institute.

Boyd, L., Guo, S., Levitan, D., Stinchcomb, D. T., & Kemphues, K. J. (1996). PAR-2 is asymmetrically distributed and promotes association of P granules and PAR-1 with the cortex in C. elegans embryogenesis. *Development, 122*(10), 3075-3084.

Boyl, T., Bao, Z., Murray, J., Araya, C., & Waterston, R. (2006). AceTree: a tool for visual analysis of *Caenorhabditis elegans* embryogenesis. *BMC Bioinformatics*.

Brandon, D., Cromartie, J., and Decker, W. (2013). "Modeling Development in *C. elegans*", *Major Qualifying Project*: Worcester Polytechnic Institute.

Cheeks, R. J., Canman, J. C., Gabriel, W. N., Meyer, N., Strome, S., & Goldstein, B. (2004). *C. elegans* PAR Proteins Function by Mobilizing and Stabilizing Asymmetrically Localized Protein Complexes. *Current Biology, 14*(10), 851-862.

Clavaud, Nicolas (2013). "Picking." *Picking*. Web. 13 Mar. 2014. http://n.clavaud.free.fr/processing/library/picking/

Cooke, J. G. (1999). Improvement of fishery-management advice through simulationtesting of harvest algorithms. *ICES Journal of Marine Science, 56*.

Crittenden, S. L., Rudel, D., Binder, J., Evans, T. C., & Kimble, J. (1997). Genes Required for GLP-1 Asymmetry in the Early *Caenorhabditis elegans* Embryo. *Developmental Biology, 181*(1), 36-46.

Eisenmann, David (2005). "Wnt Signaling." WormBook ed: The C. elegans Research Community. http://www.wormbook.org/chapters/www_wntsignaling/wntsignaling.html

Etemad-Moghadam, B., Guo, S., & Kemphues, K. J. (1995). Asymmetrically distributed PAR-3 protein contributes to cell polarity and spindle alignment in early *C. elegans* embryos. *Cell, 83*(5), 743-752.

Evans, T. C., & Hunter, C. P. (2005). Translational control of maternal RNAs. *WormBook* ed: The *C. elegans* Research Community, WormBook, http://www.wormbook.org.

Feinberg, Jonathon (2013). "Peasycam V200." *Peasycam*. Web. 13 Mar. 2014. http://mrfeinberg.com/peasycam/

Fisher, J., Piterman, N., Hajnal, A., & Henzinger, T. (2007). Predictive Modeling of Signaling Crosstalk during *C. elegans* Vulval Development: PLoS Computational Biology.

Fry, Ben, and Casey Reas (2014). "Processing.org." *Processing.org*. Web. 13 Mar. 2014. http://www.processing.org/

Gönczy, P., & Rose, L.S. (2005) "Asymmetric Cell Division and Axis Formation in the Embryo." WormBook ed: The C. elegans Research Community, WormBook, http://www.wormbook.org.

Guo, S., & Kemphues, K. J. (1995). Par-1, a Gene Required for Establishing Polarity in C. elegans Embryos, Encodes a Putative Ser/Thr Kinase That Is Asymmetrically Distributed. *Cell, 81*(4), 611-620.

Hao, Y., Boyd, L., & Seydoux, G. (2006). Stabilization of Cell Polarity by the *C. elegans* RING Protein PAR-2. *Developmental Cell, 10*(2), 199-208.

Hoege, C., & Hyman, A. A. (2013). Principles of PAR polarity in *Caenorhabditis elegans* embryos. *Nature Reviews Molecular Cell Biology*, *14(5)*, 315-322.

Huang, N. N., Mootz, D. E., Walhout, A. J. M., Vidal, M., & Hunter, C. P. (2002). MEX-3 interacting proteins link cell polarity to asymmetric gene expression in *Caenorhabditis elegans*. *Development, 129*(3), 747-759.

Idili, G., & Cantarelli, M. (2011-2013). OpenWorm. Retrieved 3-10-2014. http://www.openworm.org.

Kaletta, T., & Hengartner, M. O. (2006). Finding function in novel targets: *C. elegans* as a model organism. *Nat Rev Drug Discov, 5*(5), 387-398.

Kitano, H. (2002). Computational systems biology. *Nature, 420*, 206-210.

Kitano, H. (1998). The Perfect C. elegans Project: An Initial Report. In S. Hamahashi (Ed.): Artificial Life.

Law, A. M., & Kelton, D. W. (2000). *Simulation Modeling and Analysis* (3 ed.). Boston: McGraw Hill.

Loew, L. M., & Schaff, J. C. (2001). The Virtual Cell: a software environment for computational cell biology. *Trends in Biotechnology, 19*(10), 401-406.

Macara, I. G. (2004). Parsing the Polarity Code. *Nature Reviews Molecular Cell Biology, 5*(3), 220-231.

Mango, S. E., Thorpe, C. J., Martin, P. R., Chamberlain, S. H., & Bowerman, B. (1994). Two maternal genes, *apx-1* and *pie-1*, are required to distinguish the fates of equivalent blastomeres in the early *Caenorhabditis elegans* embryo. *Development, 120*(8), 2305-2315.

Mickey, K. M., Mello, C. C., Montgomery, M. K., Fire, A., & Priess, J. R. (1996). An inductive interaction in 4-cell stage *C. elegans* embryos involves APX-1 expression in the signalling cell. *Development, 122*(6), 1791-1798.

Motegi, F. & Seydoux, G. (2013).The PAR network: redundancy and robustness in a symmetry-breaking system. *Philosophical Transactions of Royal Society B* 368: 20130010. http://dx.doi.org/10.1098/rstb.2013.0010

Nance, J., Lee, J.-Y., & Goldstein, B. (2005a). Gastrulation in *C. elegans* (WormBook ed.): The C. elegans Research Community, WormBook, http://www.wormbook.org.

Nance, J. (2005b). PAR proteins and the establishment of cell polarity during *C. elegans* development. *BioEssays, 27*(2), 126-135.

Neves, A., & Priess, J. (2005). The REF-1 Family of bHLH Transcription Factors Pattern C. elegans Embryos through Notch-Dependent and Notch-Independent Pathways. *Developmental Cell, 8*(6), 867-879.

Priess, J. R. (2005). Notch signaling in the *C. elegans* embryo. WormBook ed: The C. elegans Research Community.

Reese, K. J., Dunn, M. A., Waddle, J. A., & Seydoux, G. (2000). Asymmetric Segregation of PIE-1 in *C. elegans* Is Mediated by Two Complementary Mechanisms that Act through Separate PIE-1 Protein Domains. *Molecular Cell, 6*(2), 445-455.

Riddle DL, Blumenthal T, Meyer BJ, *et al*. (1997). Editors . *C. elegans* II. 2nd edition. Cold Spring Harbor (NY): Cold Spring Harbor Laboratory Press.

Rocheleau, C., Yasuda, J., Shin, T. H., Lin, R., Sawa, H., Okano, H., et al. (1999). WRM-1 Activates the LIT-1 Protein Kinase to Transduce Anterior/Posterior Polarity Signals in *C. elegans*. *Cell, 97*(6), 717-726.

Rogers, A., Antoshechkin, I., Bieri, T., Blasiar, D., Bastiani, C., Canaran, P., ... & Sternberg, P. W. (2008). WormBase 2007. Nucleic acids research, 36(suppl 1), D612-D617.

Rose, L.S. & Kemphues K.J. (1998). Early Patterning of the *C. elegans* Embryo. Annual Review of Genetics 32: 521-545.

Schlegal, Andreas (2012). "ControlP5." *Processing GUI*. Web. 13 Mar. 2014. http://www.sojamo.de/libraries/ControlP5/

Schnabel, R., Hutter, H., Moerman, D. and Schnabel, H. (1997). Assessing normal embryogenesis in Caenorhabditis elegans using a 4D microscope: variability of development and regional specification. Dev. Biol. 184,234 -265.

Shannon, P., Markiel, A., Ozier, O., Balinga, N., Wang, J., Ramage, D., et al. (2003). Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Research*, *13(11)*, 2498-2504.

Stein, L. D., & Thierry-Mieg, J. (1998). Scriptable access to the Caenorhabditis elegans genome sequence and other ACEDB databases. Genome research, 8(12), 1308-1315.

Sulston, J. E., Schierenberg, E., White, J. G., & Thomson, J. N. (1983). The Embryonic Cell Lineage of the Nematode Caenorhabditis elegans. Developmental Biology, 100(1), 64-119.

Tabara, H., Hill, R. J., Mello, C. C., Priess, J. R., & Kohara, Y. (1999). Pos-1 encodes a cytoplasmic zinc-finger protein essential for germline specification in *C. elegans*. *Development, 126*(1), 1-11.

Watts, J. L., Morton, D. G., Bestman, J., & Kemphues, K. J. (2000). The C. elegans par-4 gene encodes a putative serine-threonine kinase required for establishing embryonic asymmetry. *Development, 127*(7), 1467-1475.

Wu, Y., Ghitani, A., Christensen, R., Santella, A., Du, Z., Rondeau, G., Bao, Z., Colón-Ramos, D., & Shroff, H. (2011). Inverted selective plane illumination microscopy (iSPIM) enables coupled cell identity lineaging and neurodevelopmental imaging in Caenorhabditis elegans. *Proceedings of the National Academy of Sciences, 108*(43), 17708-17713.

Wyvill, Geoff, and Andrew Trotman. (1990). *Ray-tracing soft objects*. Springer Japan, 1990.

9 Appendices

9.1 Appendix A: Implementation Guide

Running SimWorm14

Running the simulation requires simply double clicking the executable jar file. It must be accompanied by three .csv files in order to read in all the necessary data. These files are entitled AandC.csv, eventsQueue.csv, and genes.csv. These file names must not be altered, and all three files must be in the same directory where the executable jar is located.

Antecedents and Consequents Table

This table is designed to run the protein interactions within cells and between cells. Currently, we have worked on interactions for the wild type[1] that can have effects on cell location within the embryo, polarity and cell fate. This table is broken into several columns:

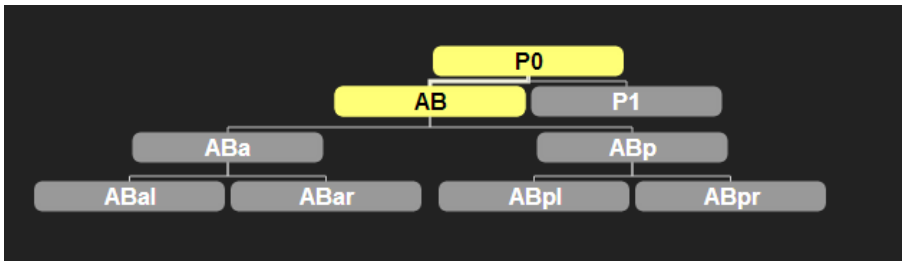
- Antecedents: the necessary conditions to have an effect on another protein which is specified in consequents
    - The 3 letter and number codes are the names of proteins
    - An up arrow indicates that the protein is active
    - The down arrow indicates that the protein is inactive
- Consequents: given that the conditions in the antecedent is true for each cell, change the protein state as specified in this column
    - This works like an if/then statement (if [antecedent], then [consequent])
- P or T: specifies if the state change is due to phosphorylation (P), transcription (T), or unknown (U)
    - Phosphorylation: time of change specified in consequent is **instant** due to an instant change in the protein conformation
    - Transcription: time of change specified in consequent is **delayed** due to a change in expression
    - Unknown: no information was found
- startStage: the minimum number of cells where that rule is applicable, expressed as cell stage
- endStage: the maximum number of cells where that rule is applicable, expressed as cell stage
    - Ex: row 2 rule is active from the embryo having 1 cell to the embryo having 26 cells, when the embryo consists of more than 26 cells we do not need to check this rule
- Interaction location: the location in the cell where this interaction is taking place in the wild type
    - Rules are only applicable on cells located in the region specified
    - **Axes**: AP, DV, LR
        - AP: Anterior (x-axis negative values) and posterior (x-axis positive values)
        - DV: Dorsal (y-axis positive values) and ventral (y-axis negative values)
        - LR: Left (z-axis positive – toward the user) and right (z-axis negative)

[1] A species that occurs under natural circumstances, not a mutant

The current working version of the antecedent and consequents table that is read in by the program is entitled AandC.csv and must be included in the directory with the executable jar in order to run the simulation.

Lineaging Table

The entirety of the *C. elegans* lineage is known. The lineage is the sequence of cell divisions and cell identities (naming is based on the cell tissue type and cell family). An example of the *C. elegans* lineage for the first few cell divisions that also shows fate (designated by color) and cell location can be found at this location: http://labs.bio.unc.edu/Goldstein/CelegansGastrulationLineage.jpg. Additionally, the figure below shows the first few divisions as a lineage tree.



You can look at subsequent divisions by visiting this website:
http://wormweb.org/celllineage#c=P0a&z=1

We have created a lineaging table that holds several pieces of information as described below:

- Parent: the name of the parent cell
- Daughter 1 & Daughter 2: each of the resulting cells after cell division of the parent
- Division of axis: the axis for which each daughter cell appears (for an axis definition, please refer to the description of interaction location under Antecedents and Consequents Table)
- Time of division: the biological data known for the cell time of division, this is calculated continuously from the start time (time of first division is 0)
- Sim Time: this is 10 minutes more than the Time of division in order to shown the first cell division in the simulation
- V_Daughter1 & V_Daughter2: Comparative volume of each daughter cell in the wild type
    - For now, the cells either split equally (50/50) or unequally (60/40)

The lineaging table that is read in by the program is entitled eventsQueue.csv and must be included in the directory with the executable jar in order to run the simulation.

Simulation Features

We understand that designing this simulation will require forethought of the types of features that will make it most useful in order to ensure the coding and implementation are created with these features in mind. Below is an outline of what we would like to have included:

- 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

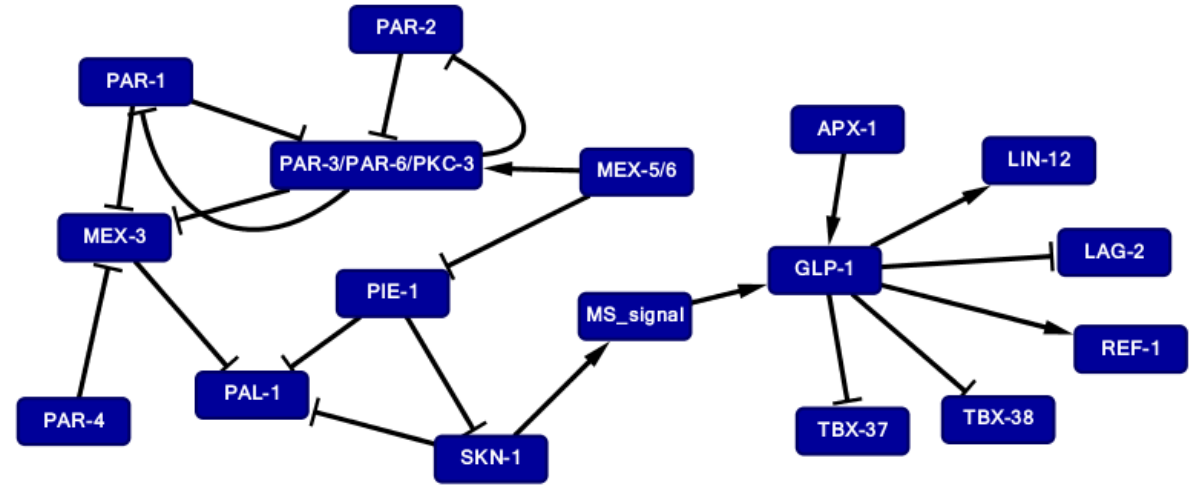
    - Be able to filter out cells – hide them from view
    - Displays should be linked (when rotating the an axis, the same cells should be highlighted)
- Capability of adjusting window proportions – having a split screen and then adjusting the proportion of the screen that shows the simulation and the part that tracks data
- Model the cells using metaballs to achieve more accurate shapes and locations

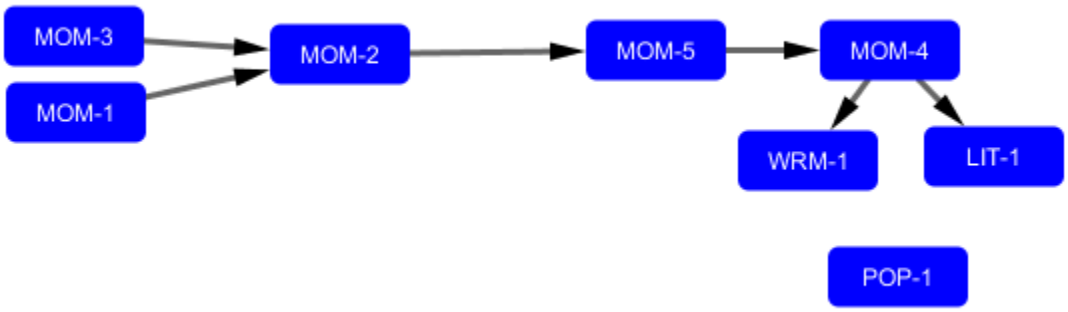
Further, the following is a list of previous goals that were accomplished successfully:

- Capability to rotate the embryo on three different planes along each axis (x-axis, y-axis, z-axis)
    - Maintains integrity of cell positioning within the embryo
- Label cells with the cell fate
    - This was indicated by color scheme.
        - For coloring schemes: http://colorbrewer2.org/
- Capability of changing the initial conditions of the genes for hypothesis testing
    - Turn on/off specific genes per user's needs to test mutant
    - This simply involves altering the data in the genes.csv file

9.2 Appendix B: Cytoscape Image of Protein Interactions

The following images depict the Notch signaling pathway with cell polarity protein interactions and the Wnt signaling pathway, respectively. They were generated using the mapping software Cytoscape (Shannon, 2003). Each node is a protein or complex of proteins and each edge indicates the type of interaction. An arrow from node A to node B indicates that protein A is activating protein B. Likewise, a "T" from node A to node B indicates that protein A is inhibiting protein B. These protein maps include all of the proteins that were researched during this project period.





9.3 Appendix C: Full Antecedents and Consequents Table

Antecedents	Consequences
pie-1 ↑, pal-1 ↑	pal-1 ↓
pie-1 ↑, skn-1 ↑	skn-1 ↓
skn-1 ↑, pal-1 ↑	pal-1 ↓
mex-3 ↑, pal-1 ↑	pal-1 ↓
mex-5/6 ↑, par-3 ↓	par-3 ↑
mex-5/6 ↑, par-6 ↓	par-6 ↑
mex-5/6 ↑, pkc-3 ↓	pkc-3 ↑
mex-5/6 ↑, pie-1 ↑	pie-1 ↓
par-2 ↑, pkc-3 ↑	pkc-3 ↓
par-2 ↑, par-6 ↑	par-6 ↓
par-2 ↑, par-3 ↑	par-3 ↓
par-1 ↑, pkc-3 ↑	pkc-3 ↓
par-1 ↑, par-6 ↑	par-6 ↓
par-1 ↑, par-3 ↑	par-3 ↓
par-3 ↑, par-6 ↑, pkc-3 ↑, par-2 ↑	par-2 ↓
par-3 ↑, par-1 ↑	par-1 ↓
par-1 ↑, mex-5/6 ↑	mex-5/6 ↓
par-1 ↑, mex-3 ↑	mex-3 ↓
par-4 ↑, mex-3 ↑	mex-3 ↓
apx-1 ↑, glp-1 ↓	glp-1 ↑
glp-1 ↑, tbx-37 ↑	tbx-37 ↓
glp-1 ↑, tbx-38 ↑	tbx-38 ↓
glp-1 ↑, ref-1 ↓	ref-1 ↑
glp-1 ↑, ref-1 ↓	ref-1 ↑
skn-1 ↑, MS signal ↓	MS signal ↑
MS signal ↑, glp-1 ↓	glp-1 ↑
glp-1 ↑, lag-2 ↑	lag-2 ↓
glp-1 ↑, lin-12 ↓	lin-12 ↑
glp-1 ↑, tbx-37 ↑, tbx-38 ↑, pha-4 ↓	pha-4 ↑
mom-3 ↑, mom-1 ↑, mom-2 ↓	mom-2 ↑
mom-5 ↑, mom-4 ↓	mom-4 ↑
mom-4 ↑, lit-1 ↓	lit-1 ↑
mom-4 ↑, wrm-1 ↓	wrm-1 ↑
mom-4 ↑, wrm-1 ↑, lit-1 ↑, pop-1 ↑	pop-1 ↓
mom-2 ↑, mom-5 ↓	mom-5 ↑
skn-1 ↑, med-2 ↓	med-2 ↑
skn-1 ↑, med-1 ↓	med-1 ↑
wrm-1 ↑, lit-1 ↑, pop-1 ↓	pop-1 ↑

9.4 Appendix D: Glossary

abstraction – A relatively user friendly tool that performs the same functions as a more difficult tool. For example, one programming language is an abstraction for another if it can achieve the same results with more readable code.

allele – One member of a pair of genes, a pair of alleles are located at the same spot on the same chromosome and code for the same trait.

apicobasal – The difference between the outside of the body (apical) and inside of the body (basal).

apoptosis – Programmed cell death.

blast cells – Immature precursor cells that have not differentiated.

centrosome – An organelle that organizes the microtubules of the cell and helps regulate cell divisions.

class – In object oriented languages, a file in which a type of objects is defined. The class typically contains the object's attributes and any functions that operate on objects of that type.

class hierarchy – A tree-like way of structuring object classes; describes the way classes relate to one another.

constant time complexity – A function whose computation time does not change based on the size of the input data.

constructor – The code that runs when a new instance of an object is created. It usually sets the values of any attributes that objects of that type contain.

cortex – A specialized layer of cytoplasm on the inner face of the plasma membrane.

cytoplasm – The material within the plasma membrane of a cell excluding the nucleus.

deep clone – A duplicate of a piece of data that is completely independent of the original, so that changing one will not affect the other.

deterministic simulation – A simulation that does not incorporate probabilities to predict an event. A deterministic simulation will always produce the same outcome given the same input.

discrete simulation – A simulation where events occur at a specific time point and changes due to an event are propagated at a specific time point.

drawing loop – In events-based languages, the code inside of the draw function is called repeatedly any time that the program is idle.

dynamic simulation – A simulation in which changes are made over time.

events queue – A repository of events which are waiting to be called upon by the program.

fluorescent marker – A protein that exhibits a bright color when exposed to a specific range of wavelengths of light.

gravid – Pregnant, carrying eggs.

GUI (graphical user interface) – The visual that a user sees when running a program. The user interacts with these visuals to trigger computations in the program. The goal is program usage to be intuitive for the user.

hard-coding – Setting program parameters in a way so that they cannot be changed without altering the code.

hashmap – A data structure that stores sets of data of the form <key, value>. The key and the value are linked such that if the programmer knows the key, he or she can easily access the corresponding value.

hermaphrodite – The sexual form of the *C. elegans* that contains both eggs and sperm allowing for self-fertilization.

heterozygous – Having two different alleles for the same trait.

histone – Highly alkaline proteins that are used for packing and organizing DNA, found in high concentrations in the nucleus of cells.

homozygous – Having two copies of the same allele for a trait.

kinase – An enzyme that transfers a phosphate group from ATP to a protein.

L1 arrest – An alternative development stage in *C.elegans* life cycle that can occur right after hatching if there is a lack of resources in the environment; the worm can survive several weeks without food at this stage, conserving energy by not developing fully.

L4 – The fourth *C. elegans* larval stage occurring starting at 40-49.5 hours after hatching when cultured at 25^{o}C, last stage before the reproductive adult stage.

linkage – The tendency for two genes that are located near each other on a chromosome to be inherited together during meiosis.

methods – Processes that consume input, perform computations, and produce output.

mislocalized – Located in the wrong cell or area of the cell.

N2M – The wildtype *C. elegans* strain that contains males.

Nematode Growth Media – Standard media used for growing *C. elegans*.

oocyte – An immature egg cell that has not been fertilized.

OpenGL – A programming language for drawing graphics.

parsing – Converting data that is human-readable into data structures that a program can understand. If the data is structured in a systematic way, parsing can be done programmatically.

phosphorylation – The addition of a phosphate group to a protein or other organic molecule.

postembryonic – After the embryonic stage.

set-up function – In events-based languages, the set up function contains the first code to be executed at the start of the program. This code initializes any data structures that need to exist for the remainder of the execution.

somatic – Cells forming the body of the organism as opposed to the germline, or sexual reproductive cells.

static simulation – A simulation in which time is not a factor.

stochastic simulation – A simulation that incorporates probabilities to predict an event. Stochastic simulations are variable and may not produce the same outcome given the same input.

Sulston naming – The cell lineage nomenclature described in Sulston, 1983.

system – The subject for which a simulation is being written and can range in size and complexity.

time complexity – The time a program takes to run as a function of the size of the input.

transcription – The transition of the genetic message from DNA to RNA.

wildtype – The typical form of an organism as it occurs in nature.

9.5 Appendix E: Javadocs

dataStructures
Enum Axes

java.lang.Object

└ java.lang.Enum<Axes>

└ **dataStructures.Axes**

All Implemented Interfaces:

java.io.Serializable, java.lang.Comparable<Axes>

---

public enum **Axes**

extends java.lang.Enum<Axes>

An enum to indicate axes in three dimensional space choices are X, Y, Z

Author:

Rachel

Enum Constant Summary	
X	
Y	
Z	

Method Summary

static Axes	**valueOf**(java.lang.String name) Returns the enum constant of this type with the specified name.
static Axes[]	**values**() Returns an array containing the constants of this enum type, in the order they are declared.

Methods inherited from class java.lang.Enum

clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf

Methods inherited from class java.lang.Object

getClass, notify, notifyAll, wait, wait, wait

Enum Constant Detail

X

public static final Axes **X**

Y

public static final Axes **Y**

Z

public static final Axes **Z**

Method Detail

values

public static Axes[] **values**()

Returns an array containing the constants of this enum type, in the order they are declared. This method may be used to iterate over the constants as follows:

```
for (Axes c : Axes.values())
    System.out.println(c);
```

Returns:

an array containing the constants of this enum type, in the order they are declared

---

valueOf

public static Axes **valueOf**(java.lang.String name)

Returns the enum constant of this type with the specified name. The string must match *exactly* an identifier used to declare an enum constant in this type. (Extraneous whitespace characters are not permitted.)

Parameters:

`name` - the name of the enum constant to be returned.

Returns:

the enum constant with the specified name

Throws:

`java.lang.IllegalArgumentException` - if this enum type has no constant with the specified name

`java.lang.NullPointerException` - if the argument is null

processing
Class BasicVisual

java.lang.Object

└ java.awt.Component

└ java.awt.Container

└ java.awt.Panel

└ java.applet.Applet

└ processing.core.PApplet

└ **processing.BasicVisual**

All Implemented Interfaces:

java.awt.event.FocusListener, java.awt.event.KeyListener, java.awt.event.MouseListener, java.awt.event.MouseMotionListener, java.awt.event.MouseWheelListener, java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.lang.Runnable, java.util.EventListener, javax.accessibility.Accessible, processing.core.PConstants

---

public class **BasicVisual**

extends processing.core.PApplet

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class processing.core.PApplet
`processing.core.PApplet.RendererChangeException`

Nested classes/interfaces inherited from class java.applet.Applet
java.applet.Applet.AccessibleApplet

Nested classes/interfaces inherited from class java.awt.Panel
java.awt.Panel.AccessibleAWTPanel

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BaselineResizeBehavior, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

(package private) controlP5.Button	**backB**
(package private) controlP5.Button	**bottomB**
(package private) peasy.PeasyCam	**camera**
(package private) controlP5.RadioButton	**chooseColorMode**

(package private) controlP5.CheckBox	**chooseMutants**
(package private) controlP5.Button	**createShell**
(package private) Shell	**displayShell**
(package private) controlP5.Button	**fateKey0**
(package private) controlP5.Button	**fateKey1**
(package private) controlP5.Button	**fateKey2**
(package private) controlP5.Button	**fateKey3**
(package private) controlP5.Button	**fateKey4**
(package private) controlP5.Button	**fateKey5**
(package private) controlP5.Button	**fateKey6**
(package private) boolean	**fateState**
(package private) controlP5.Button	**frontB**

(package private) controlP5.ControlP5	**info**
(package private) controlP5.Button	**leftB**
(package private) controlP5.Button	**lineageKey0**
(package private) controlP5.Button	**lineageKey1**
(package private) controlP5.Button	**lineageKey2**
(package private) controlP5.Button	**lineageKey3**
(package private) controlP5.Button	**lineageKey4**
(package private) controlP5.Button	**lineageKey5**
(package private) controlP5.Button	**lineageKey6**
(package private) boolean	**lineageState**
(package private) processing.core.PMatrix	**matScene**
(package private) java.util.HashMap<java.lang.String,java.lang.Boolea	**mutants**

n>	
(package private) boolean	**mutantsChosen**
(package private) controlP5.Button	**parsKey0**
(package private) controlP5.Button	**parsKey1**
(package private) controlP5.Button	**parsKey2**
(package private) controlP5.Button	**parsKey3**
(package private) controlP5.Button	**parsKey4**
(package private) controlP5.Button	**parsKey5**
(package private) boolean	**parsState**
(package private) controlP5.Button	**rightB**
(package private) controlP5.Button	**topB**
(package private) java.lang.String	**userText**
(package private) controlP5.Textarea	**userTextArea**

Fields inherited from class processing.core.PApplet

args, ARGS_BGCOLOR, ARGS_DISPLAY, ARGS_EDITOR_LOCATION, ARGS_EXTERNAL, ARGS_FULL_SCREEN, ARGS_HIDE_STOP, ARGS_LOCATION, ARGS_PRESENT, ARGS_SKETCH_FOLDER, ARGS_STOP_COLOR, DEFAULT_HEIGHT, DEFAULT_WIDTH, defaultSize, displayHeight, displayWidth, dmouseX, dmouseY, emouseX, emouseY, exitCalled, EXTERNAL_MOVE, EXTERNAL_STOP, finished, firstMouse, focused, frame, frameCount, frameRate, frameRateLastNanos, frameRatePeriod, frameRateTarget, g, height, insideDraw, javaVersion, javaVersionName, key, keyCode, keyEvent, keyPressed, loadImageFormats, looping, matchPatterns, MIN_WINDOW_HEIGHT, MIN_WINDOW_WIDTH, mouseButton, mouseEvent, mousePressed, mouseX, mouseY, online, paused, pixels, platform, pmouseX, pmouseY, recorder, redraw, requestImageMax, sketchPath, useNativeSelect, useQuartz, width

Fields inherited from class java.awt.Component

BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface processing.core.PConstants

ADD, ALPHA, ALT, AMBIENT, ARC, ARGB, ARROW, BACKSPACE, BASELINE, BEVEL, BEZIER_VERTEX, BLEND, BLUR, BOTTOM, BOX, BREAK, BURN, CENTER, CHATTER, CHORD, CLAMP, CLOSE, CODED, COMPLAINT, CONTROL, CORNER, CORNERS, CROSS, CURVE_VERTEX, CUSTOM, DARKEST, DEG_TO_RAD, DELETE, DIAMETER, DIFFERENCE, DILATE, DIRECTIONAL, DISABLE_DEPTH_MASK, DISABLE_DEPTH_SORT, DISABLE_DEPTH_TEST, DISABLE_NATIVE_FONTS, DISABLE_OPENGL_ERRORS, DISABLE_OPTIMIZED_STROKE, DISABLE_RETINA_PIXELS, DISABLE_STROKE_PERSPECTIVE, DISABLE_STROKE_PURE, DISABLE_TEXTURE_MIPMAPS, DODGE, DOWN, DXF, ELLIPSE, ENABLE_DEPTH_MASK, ENABLE_DEPTH_SORT, ENABLE_DEPTH_TEST, ENABLE_NATIVE_FONTS, ENABLE_OPENGL_ERRORS, ENABLE_OPTIMIZED_STROKE, ENABLE_RETINA_PIXELS, ENABLE_STROKE_PERSPECTIVE, ENABLE_STROKE_PURE, ENABLE_TEXTURE_MIPMAPS, ENTER, EPSILON, ERODE, ERROR_BACKGROUND_IMAGE_FORMAT, ERROR_BACKGROUND_IMAGE_SIZE, ERROR_PUSHMATRIX_OVERFLOW, ERROR_PUSHMATRIX_UNDERFLOW, ERROR_TEXTFONT_NULL_PFONT, ESC, EXCLUSION, GIF, GRAY, GROUP, HALF_PI, HAND, HARD_LIGHT, HINT_COUNT, HSB, IMAGE, INVERT, JAVA2D, JPEG, LANDSCAPE, LEFT, LIGHTEST, LINE, LINE_LOOP, LINE_STRIP, LINES, LINUX, MACOSX, MAX_FLOAT,

MAX_INT, MIN_FLOAT, MIN_INT, MITER, MODEL, MODELVIEW, MOVE, MULTIPLY, NORMAL, OPAQUE, OPEN, OPENGL, ORTHOGRAPHIC, OTHER, OVERLAY, P2D, P3D, PATH, PDF, PERSPECTIVE, PI, PIE, platformNames, POINT, POINTS, POLYGON, PORTRAIT, POSTERIZE, PROBLEM, PROJECT, PROJECTION, QUAD, QUAD_BEZIER_VERTEX, QUAD_STRIP, QUADRATIC_VERTEX, QUADS, QUARTER_PI, RAD_TO_DEG, RADIUS, RECT, REPEAT, REPLACE, RETURN, RGB, RIGHT, ROUND, SCREEN, SHAPE, SHIFT, SOFT_LIGHT, SPHERE, SPOT, SQUARE, SUBTRACT, TAB, TARGA, TAU, TEXT, THIRD_PI, THRESHOLD, TIFF, TOP, TRIANGLE, TRIANGLE_FAN, TRIANGLE_STRIP, TRIANGLES, TWO_PI, UP, VERTEX, WAIT, WHITESPACE, WINDOWS, X, Y, Z

Fields inherited from interface java.awt.image.ImageObserver

ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

BasicVisual()

Method Summary

(package private) void	**back**(float theValue)
(package private) void	**bottom**(float theValue)
(package private) void	**createShell**(float theValue)
void	**draw**()

void	**drawAxes**()
void	**drawKey**(ColorMode colorMode)
(package private) void	**front**(float theValue)
void	**gui**()
void	**keyReleased**()
(package private) void	**left**(float theValue)
static void	**main**(java.lang.String[] args)
(package private) void	**right**(float theValue)
void	**secondarySetup**()
void	**setup**()
(package private) void	**top**(float theValue)
void	**updateColorMode**()

Methods inherited from class processing.core.PApplet

abs, abs, acos, addListeners, alpha, ambient, ambient, ambient, ambientLight, ambientLight, append, append, append, append, append, append, applyMatrix, applyMatrix, applyMatrix, applyMatrix, applyMatrix, arc, arc, arraycopy, arrayCopy, arraycopy, arrayCopy, arraycopy, arrayCopy, asin, atan, atan2, background, background, background, background, background, background, background, beginCamera, beginContour, beginPGL, beginRaw, beginRaw, beginRecord, beginRecord, beginShape, beginShape, bezier, bezier, bezierDetail, bezierPoint, bezierTangent, bezierVertex, bezierVertex, binary, binary, binary, binary, blend, blend, blendColor, blendMode, blue, box, box, brightness, camera, camera, canDraw, ceil, checkExtension, clear, clip, color, color, color, color, color, color, color, color, colorMode, colorMode, colorMode, colorMode, concat, concat, concat, concat, concat, concat, concat, constrain, constrain, copy, copy, cos, createDefaultFont, createFont, createFont, createFont, createGraphics, createGraphics, createGraphics, createImage, createInput, createInput, createInputRaw, createOutput, createOutput, createPath, createPath, createReader, createReader, createReader, createShape, createShape, createShape, createShape, createWriter, createWriter, createWriter, cursor, cursor, cursor, cursor, curve, curve, curveDetail, curvePoint, curveTangent, curveTightness, curveVertex, curveVertex, dataFile, dataPath, day, debug, degrees, delay, dequeueEvents, desktopFile, desktopPath, destroy, die, die, directionalLight, displayable, dispose, dist, dist, edge, ellipse, ellipseMode, emissive, emissive, emissive, endCamera, endContour, endPGL, endRaw, endRecord, endShape, endShape, exec, exit, exp, expand, expand, expand, expand, expand, expand, expand, expand, expand, expand, expand, expand, expand, expand, expand, expand, expand, expand, fill, fill, fill, fill, fill, fill, filter, filter, filter, filter, floor, flush, focusGained, focusGained, focusLost, focusLost, frameRate, frustum, get, get, get, getCache, getExtension, getMatrix, getMatrix, getMatrix, green, handleDraw, handleKeyEvent, handleMethods, handleMethods, handleMouseEvent, hex, hex, hex, hex, hint, hour, hue, image, image, image, imageMode, init, insertFrame, isGL, join, join, keyPressed, keyPressed, keyPressed, keyReleased, keyReleased, keyTyped, keyTyped, keyTyped, lerp, lerpColor, lerpColor, lightFalloff, lights, lightSpecular, line, line, link, link, loadBytes, loadBytes, loadBytes, loadFont, loadImage, loadImage, loadImageIO, loadImageMT, loadImageTGA, loadJSONArray, loadJSONArray, loadJSONObject, loadJSONObject, loadPixels, loadShader, loadShader, loadShape, loadShape, loadStrings, loadStrings, loadStrings, loadStrings, loadTable, loadTable, loadXML, loadXML, log, loop, mag, mag, main, main, makeGraphics, map, mask, match, matchAll, max, max, max, max, max, max, method, millis, min, min, min, min, min, min, minute, modelX, modelY, modelZ, month, mouseClicked, mouseClicked, mouseClicked, mouseDragged, mouseDragged, mouseDragged, mouseEntered, mouseEntered, mouseEntered, mouseExited, mouseExited, mouseExited, mouseMoved, mouseMoved, mouseMoved, mousePressed, mousePressed, mousePressed, mouseReleased, mouseReleased, mouseReleased, mouseWheel, mouseWheel, mouseWheelMoved, nativeKeyEvent, nativeMouseEvent, nf, nf, nf, nf, nfc, nfc, nfc, nfc, nfp,

```
nfp, nfp, nfp, nfs, nfs, nfs, nfs, noClip, noCursor, noFill, noise, noise,
noise, noiseDetail, noiseDetail, noiseSeed, noLights, noLoop, norm, normal,
noSmooth, noStroke, noTexture, noTint, open, open, openStream, orientation,
ortho, ortho, ortho, paint, param, parseBoolean, parseBoolean, parseBoolean,
parseBoolean, parseByte, parseByte, parseByte, parseByte, parseByte,
parseByte, parseByte, parseByte, parseByte, parseChar, parseChar, parseChar,
parseChar, parseFloat, parseFloat, parseFloat, parseFloat, parseFloat,
parseFloat, parseInt, parseInt, parseInt, parseInt, parseInt, parseInt,
parseInt, parseInt, parseInt, parseInt, parseInt, parseInt, parseJSONArray,
parseJSONObject, parseXML, parseXML, pause, perspective, perspective, point,
point, pointLight, popMatrix, popStyle, postEvent, pow, print, print, print,
print, print, print, print, print, print, printArray, printCamera, println,
println, println, println, println, println, println, println, println,
println, println, printMatrix, printProjection, pushMatrix, pushStyle, quad,
quadraticVertex, quadraticVertex, radians, random, random, randomGaussian,
randomSeed, rect, rect, rect, rectMode, red, redraw, registerDispose,
registerDraw, registerKeyEvent, registerMethod, registerMouseEvent,
registerPost, registerPre, registerSize, removeCache, removeListeners,
render, reportDeprecation, requestImage, requestImage, resetMatrix,
resetShader, resetShader, resizeRenderer, resume, reverse, reverse, reverse,
reverse, reverse, reverse, reverse, rotate, rotate, rotateX, rotateY,
rotateZ, round, run, runSketch, runSketch, runSketch, saturation, save,
saveBytes, saveBytes, saveBytes, saveFile, saveFrame, saveFrame,
saveJSONArray, saveJSONArray, saveJSONObject, saveJSONObject, savePath,
saveStream, saveStream, saveStream, saveStream, saveStream, saveStrings,
saveStrings, saveStrings, saveTable, saveTable, saveXML, saveXML, scale,
scale, scale, screenX, screenX, screenY, screenY, screenZ, second,
selectFolder, selectFolder, selectFolder, selectFolder, selectImpl,
selectInput, selectInput, selectInput, selectInput, selectOutput,
selectOutput, selectOutput, selectOutput, set, set, setCache, setIconImage,
setMatrix, setMatrix, setMatrix, setupExternalMessages,
setupFrameResizeListener, shader, shader, shape, shape, shape, shapeMode,
shearX, shearY, shininess, shorten, shorten, shorten, shorten, shorten,
shorten, shorten, showDepthWarning, showDepthWarningXYZ, showMethodWarning,
showMissingWarning, showVariationWarning, sin, size, size, size, sketchFile,
sketchFullScreen, sketchHeight, sketchPath, sketchQuality, sketchRenderer,
sketchWidth, smooth, smooth, sort, sort, sort, sort, sort, sort, sort, sort,
sort, sort, specular, specular, specular, sphere, sphereDetail, sphereDetail,
splice, splice, splice, splice, splice, splice, splice, splice, splice,
splice, splice, splice, splice, split, split, splitTokens, splitTokens,
spotLight, sq, sqrt, start, status, stop, str, str, str, str, str, str, str,
str, str, str, stroke, stroke, stroke, stroke, stroke, stroke, strokeCap,
strokeJoin, strokeWeight, style, subset, subset, subset, subset, subset,
subset, subset, subset, subset, subset, subset, subset, subset, subset, tan,
text, text, text, text, text, text, text, text, text, text, text, textAlign,
textAlign, textAscent, textDescent, textFont, textFont, textLeading,
textMode, textSize, texture, textureMode, textureWrap, textWidth, textWidth,
textWidth, thread, tint, tint, tint, tint, tint, tint, translate, translate,
triangle, trim, trim, unbinary, unhex, unregisterDispose, unregisterDraw,
```

unregisterKeyEvent, unregisterMethod, unregisterMouseEvent, unregisterPost, unregisterPre, unregisterSize, update, updateListeners, updatePixels, updatePixels, urlDecode, urlEncode, vertex, vertex, vertex, vertex, vertex, year

Methods inherited from class java.applet.Applet

getAccessibleContext, getAppletContext, getAppletInfo, getAudioClip, getAudioClip, getCodeBase, getDocumentBase, getImage, getImage, getLocale, getParameter, getParameterInfo, isActive, newAudioClip, play, play, resize, resize, setStub, showStatus

Methods inherited from class java.awt.Panel

addNotify

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, addImpl, addPropertyChangeListener, addPropertyChangeListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getAlignmentX, getAlignmentY, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalKeys, getFocusTraversalPolicy, getInsets, getLayout, getListeners, getMaximumSize, getMinimumSize, getMousePosition, getPreferredSize, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, paramString, preferredSize, print, printComponents, processContainerEvent, processEvent, remove, remove, removeAll, removeContainerListener, removeNotify, setComponentZOrder, setFocusCycleRoot, setFocusTraversalKeys, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setFont, setLayout, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener,
addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener,
addKeyListener, addMouseListener, addMouseMotionListener,
addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents,
contains, contains, createImage, createImage, createVolatileImage,
createVolatileImage, disable, disableEvents, dispatchEvent, enable, enable,
enableEvents, enableInputMethods, firePropertyChange, firePropertyChange,
firePropertyChange, firePropertyChange, firePropertyChange,
firePropertyChange, firePropertyChange, firePropertyChange,
firePropertyChange, getBackground, getBaseline, getBaselineResizeBehavior,
getBounds, getBounds, getColorModel, getComponentListeners,
getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor,
getFocusListeners, getFocusTraversalKeysEnabled, getFont, getFontMetrics,
getForeground, getGraphics, getGraphicsConfiguration, getHeight,
getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint,
getInputContext, getInputMethodListeners, getInputMethodRequests,
getKeyListeners, getLocation, getLocation, getLocationOnScreen,
getMouseListeners, getMouseMotionListeners, getMousePosition,
getMouseWheelListeners, getName, getParent, getPeer,
getPropertyChangeListeners, getPropertyChangeListeners, getSize, getSize,
getToolkit, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent,
hasFocus, hide, imageUpdate, inside, isBackgroundSet, isCursorSet,
isDisplayable, isDoubleBuffered, isEnabled, isFocusable, isFocusOwner,
isFocusTraversable, isFontSet, isForegroundSet, isLightweight,
isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isShowing,
isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus,
mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move,
nextFocus, paintAll, postEvent, prepareImage, prepareImage, printAll,
processComponentEvent, processFocusEvent, processHierarchyBoundsEvent,
processHierarchyEvent, processInputMethodEvent, processKeyEvent,
processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, remove,
removeComponentListener, removeFocusListener, removeHierarchyBoundsListener,
removeHierarchyListener, removeInputMethodListener, removeKeyListener,
removeMouseListener, removeMouseMotionListener, removeMouseWheelListener,
removePropertyChangeListener, removePropertyChangeListener, repaint, repaint,
repaint, repaint, requestFocus, requestFocus, requestFocusInWindow,
requestFocusInWindow, reshape, setBackground, setBounds, setBounds,
setComponentOrientation, setCursor, setDropTarget, setEnabled, setFocusable,
setFocusTraversalKeysEnabled, setForeground, setIgnoreRepaint, setLocale,
setLocation, setLocation, setMaximumSize, setMinimumSize, setName,
setPreferredSize, setSize, setSize, setVisible, show, show, size, toString,
transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait,

```
wait
```

Field Detail

displayShell

Shell **displayShell**

---

userText

java.lang.String **userText**

---

camera

peasy.PeasyCam **camera**

---

matScene

processing.core.PMatrix **matScene**

---

info

controlP5.ControlP5 **info**

---

userTextArea

controlP5.Textarea **userTextArea**

---

frontB

controlP5.Button **frontB**

---

backB

controlP5.Button **backB**

---

topB

controlP5.Button **topB**

---

bottomB

controlP5.Button **bottomB**

---

leftB

controlP5.Button **leftB**

---

rightB

controlP5.Button **rightB**

---

mutantsChosen

boolean **mutantsChosen**

---

mutants

java.util.HashMap<java.lang.String,java.lang.Boolean> **mutants**

---

chooseMutants

controlP5.CheckBox **chooseMutants**

---

createShell

controlP5.Button **createShell**

---

chooseColorMode

controlP5.RadioButton **chooseColorMode**

---

lineageState

boolean **lineageState**

---

fateState

boolean **fateState**

---

parsState

boolean **parsState**

---

fateKey0

controlP5.Button **fateKey0**

---

fateKey1

controlP5.Button **fateKey1**

---

fateKey2

controlP5.Button **fateKey2**

---

fateKey3

controlP5.Button **fateKey3**

---

fateKey4

controlP5.Button **fateKey4**

---

fateKey5

controlP5.Button **fateKey5**

---

fateKey6

controlP5.Button **fateKey6**

---

parsKey0

controlP5.Button **parsKey0**

---

parsKey1

controlP5.Button **parsKey1**

---

parsKey2

controlP5.Button **parsKey2**

---

parsKey3

controlP5.Button **parsKey3**

---

parsKey4

controlP5.Button **parsKey4**

---

parsKey5

controlP5.Button **parsKey5**

---

lineageKey0

controlP5.Button **lineageKey0**

---

lineageKey1

controlP5.Button **lineageKey1**

---

lineageKey2

controlP5.Button **lineageKey2**

---

lineageKey3

controlP5.Button **lineageKey3**

---

lineageKey4

controlP5.Button **lineageKey4**

---

lineageKey5

controlP5.Button **lineageKey5**

---

lineageKey6

controlP5.Button **lineageKey6**

Constructor Detail

BasicVisual

public **BasicVisual**()

Method Detail

setup

public void **setup**()

Overrides:

`setup` in class `processing.core.PApplet`

---

secondarySetup

public void **secondarySetup**()

---

draw

public void **draw**()

Overrides:

`draw` in class `processing.core.PApplet`

---

createShell

void **createShell**(float theValue)

---

front

void **front**(float theValue)

---

back

void **back**(float theValue)

---

top

void **top**(float theValue)

---

bottom

void **bottom**(float theValue)

---

left

void **left**(float theValue)

---

right

void **right**(float theValue)

---

updateColorMode

public void **updateColorMode**()

---

keyReleased

public void **keyReleased**()

Overrides:

`keyReleased` in class `processing.core.PApplet`

---

drawKey

public void **drawKey**(ColorMode colorMode)

---

drawAxes

public void **drawAxes**()

---

gui

public void **gui**()

---

main

public static void **main**(java.lang.String[] args)

dataStructures
Class Cell

java.lang.Object

└ **dataStructures.Cell**

---

public class **Cell**

extends java.lang.Object

Field Summary

private Coordinates	**center**
private RGB	**color**
private DivisionData	**divide**
private int	**generation**
private java.util.HashMap<java.lang.String,Gene>	**genes**
private Coordinates	**lengths**
private java.lang.String	**name**
private java.lang.String	**parent**

private java.util.HashMap<java.lang.String,Gene>	**recentlyChanged**
(package private) Coordinates	**sphereLocation**
(package private) BasicVisual	**window**

Constructor Summary

Cell(BasicVisual window,
java.lang.String name, Coordinates center, Coordinates lengths,
java.lang.String parent,
java.util.HashMap<java.lang.String,Gene> genes, RGB color, DivisionData divide, int generation)

Constructor for a cell object

Method Summary

java.util.HashMap<java.lang.String,Gene>	**applyCons**() Checks for fulfilled antecedents and applies their consequences.
void	**drawCell**() Draws the cell to the PApplet
Coordinates	**getCenter**()
DivisionData	**getDivide**()

int	**getGeneration**()
java.util.HashMap<java.lang.String,Gene>	**getGenes**()
java.lang.String	**getInfo**() Returns the string that is printed to the screen when information about the cell is requested by the user
Coordinates	**getLengths**()
java.lang.String	**getName**()
java.lang.String	**getParent**()
java.util.HashMap<java.lang.String,Gene>	**getRecentlyChanged**()
Coordinates	**getSphereLocation**()
void	**setColor**(RGB color)
Cell	**timeLapse**(int stage) Per cell effects that occur on a timestep.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

window

BasicVisual **window**

---

name

private java.lang.String **name**

---

center

private Coordinates **center**

---

lengths

private Coordinates **lengths**

---

parent

private java.lang.String **parent**

---

genes

private java.util.HashMap<java.lang.String,Gene> **genes**

---

recentlyChanged

private java.util.HashMap<java.lang.String,Gene> **recentlyChanged**

---

color

private RGB **color**

---

divide

private DivisionData **divide**

---

generation

private int **generation**

---

sphereLocation

Coordinates **sphereLocation**

Constructor Detail

Cell

public **Cell**(BasicVisual window,

java.lang.String name,

Coordinates center,

Coordinates lengths,

java.lang.String parent,

java.util.HashMap<java.lang.String,Gene> genes,

RGB color,

DivisionData divide,

int generation)

Constructor for a cell object

Parameters:

`window` - The PApplet where the cell will be displayed

`name` - The name of the cell

`center` - The coordinates of the center point of the cell

`lengths` - The length of the cell on each axis

`parent` - The name of this cell's parent (the cell that divides to create this cell)

`genes` - The list of genes present in this cell

`color` - The color the cell should be rendered in

`divide` - The data that will be required to calculate this cell's division

`generation` - Generation that the cell belongs to (p-0 is 0th generation, ab and p-1 are first generation, etc)

Method Detail

getName

public java.lang.String **getName**()

---

getCenter

public Coordinates **getCenter**()

---

getLengths

public Coordinates **getLengths**()

---

getParent

public java.lang.String **getParent**()

---

getGenes

public java.util.HashMap<java.lang.String,Gene> **getGenes**()

---

getRecentlyChanged

public java.util.HashMap<java.lang.String,Gene> **getRecentlyChanged**()

---

getSphereLocation

public Coordinates **getSphereLocation**()

---

getDivide

public DivisionData **getDivide**()

---

getGeneration

public int **getGeneration**()

---

setColor

public void **setColor**(RGB color)

---

applyCons

public java.util.HashMap<java.lang.String,Gene> **applyCons**()

Checks for fulfilled antecedents and applies their consequences. Cascading effects handled on next timestep.

Returns:

The updated list of genes - cell's genelist should be set equal to this result after this method is called.

---

timeLapse

public Cell **timeLapse**(int stage)

Per cell effects that occur on a timestep. Updates the relevantCons list and calls applyCons.

Parameters:

`stage` - The number of cells present in the shell right now

Returns:

The updated cell

---

drawCell

public void **drawCell**()

Draws the cell to the PApplet

---

getInfo

public java.lang.String **getInfo**()

Returns the string that is printed to the screen when information about the cell is requested by the user

Returns:

String containing list of genes and their states

dataStructures
Class CellChangesData

java.lang.Object

└ **dataStructures.CellChangesData**

---

public class **CellChangesData**

extends java.lang.Object

Field Summary

java.util.List<Cell>	**cellsAdded**
java.util.List<java.lang.String>	**cellsRemoved**

Constructor Summary

CellChangesData(java.util.ArrayList<java.lang.String> cellsRemoved, java.util.ArrayList<Cell> cellsAdded)

Constructor for cellChangesData - holds information after a cell division about which cells are now gone and which ones will be added

Method Summary

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString,

```
wait, wait, wait
```

Field Detail

cellsRemoved

public java.util.List<java.lang.String> **cellsRemoved**

---

cellsAdded

public java.util.List<Cell> **cellsAdded**

Constructor Detail

CellChangesData

public **CellChangesData**(java.util.ArrayList<java.lang.String> cellsRemoved,

java.util.ArrayList<Cell> cellsAdded)

Constructor for cellChangesData - holds information after a cell division about which cells are now gone and which ones will be added

Parameters:

`cellsRemoved` - Cells that divided and are now gone

`cellsAdded` - Cells that were created during division

dataStructures
Enum ColorMode

java.lang.Object

└ java.lang.Enum<ColorMode>

└ **dataStructures.ColorMode**

All Implemented Interfaces:

java.io.Serializable, java.lang.Comparable<ColorMode>

---

public enum **ColorMode**

extends java.lang.Enum<ColorMode>

Enum Constant Summary

FATE
LINEAGE
PARS

Method Summary

static ColorMode	**valueOf**(java.lang.String name) Returns the enum constant of this type with the specified name.
static ColorMode[]	**values**() Returns an array containing the constants of this enum

	type, in the order they are declared.

Methods inherited from class java.lang.Enum

`clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf`

Methods inherited from class java.lang.Object

`getClass, notify, notifyAll, wait, wait, wait`

Enum Constant Detail

FATE

public static final ColorMode **FATE**

LINEAGE

public static final ColorMode **LINEAGE**

PARS

public static final ColorMode **PARS**

Method Detail

values

public static ColorMode[] **values**()

Returns an array containing the constants of this enum type, in the order they are declared. This method may be used to iterate over the constants as follows:

for (ColorMode c : ColorMode.values())

System.out.println(c);

Returns:

an array containing the constants of this enum type, in the order they are declared

---

valueOf

public static ColorMode **valueOf**(java.lang.String name)

Returns the enum constant of this type with the specified name. The string must match *exactly* an identifier used to declare an enum constant in this type. (Extraneous whitespace characters are not permitted.)

Parameters:

`name` - the name of the enum constant to be returned.

Returns:

the enum constant with the specified name

Throws:

`java.lang.IllegalArgumentException` - if this enum type has no constant with the specified name

`java.lang.NullPointerException` - if the argument is null

dataStructures
Enum Compartment

java.lang.Object

└ java.lang.Enum<Compartment>

└ **dataStructures.Compartment**

All Implemented Interfaces:

java.io.Serializable, java.lang.Comparable<Compartment>

---

public enum **Compartment**

extends java.lang.Enum<Compartment>

Enum Constant Summary

ANTERIOR
DORSAL
LEFT
POSTERIOR
RIGHT
VENTRAL
XCENTER

YCENTER
ZCENTER

Method Summary

static Compartment	**valueOf**(java.lang.String name) Returns the enum constant of this type with the specified name.
static Compartment[]	**values**() Returns an array containing the constants of this enum type, in the order they are declared.

Methods inherited from class java.lang.Enum

clone, compareTo, equals, finalize, getDeclaringClass, hashCode, name, ordinal, toString, valueOf

Methods inherited from class java.lang.Object

getClass, notify, notifyAll, wait, wait, wait

Enum Constant Detail

public static final Compartment **ANTERIOR**

XCENTER

public static final Compartment **XCENTER**

POSTERIOR

public static final Compartment **POSTERIOR**

DORSAL

public static final Compartment **DORSAL**

YCENTER

public static final Compartment **YCENTER**

VENTRAL

public static final Compartment **VENTRAL**

LEFT

public static final Compartment **LEFT**

ZCENTER

public static final Compartment **ZCENTER**

RIGHT

public static final Compartment **RIGHT**

Method Detail

values

public static Compartment[] **values**()

Returns an array containing the constants of this enum type, in the order they are declared. This method may be used to iterate over the constants as follows:

```
for (Compartment c : Compartment.values())
   System.out.println(c);
```

Returns:

an array containing the constants of this enum type, in the order they are declared

---

valueOf

public static Compartment **valueOf**(java.lang.String name)

Returns the enum constant of this type with the specified name. The string must match *exactly* an identifier used to declare an enum constant in this type. (Extraneous whitespace characters are not permitted.)

Parameters:

`name` - the name of the enum constant to be returned.

Returns:

the enum constant with the specified name

Throws:

`java.lang.IllegalArgumentException` - if this enum type has no constant with the specified name

`java.lang.NullPointerException` - if the argument is null

dataStructures
Class Consequence

java.lang.Object

└ **dataStructures.Consequence**

---

public class **Consequence**

extends java.lang.Object

Field Summary

private Gene[]	**antecedents**
private Gene	**consequence**
private int	**endStage**
private int	**startStage**

Constructor Summary

Consequence(Gene[] antecedents, Gene consequence, int startStage, int endStage) Constructor for a consequence object

Method Summary

Gene[]	**getAntecedents**()
Gene	**getConsequence**()
int	**getEndStage**()
int	**getStartStage**()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

antecedents

private Gene[] **antecedents**

---

consequence

private Gene **consequence**

---

startStage

private int **startStage**

---

endStage

private int **endStage**

Constructor Detail

Consequence

public **Consequence**(Gene[] antecedents,

Gene consequence,

int startStage,

int endStage)

Constructor for a consequence object

Parameters:

`antecedents` - All of the gene names and their states (see simple gene constructor) required for this consequence to occur

`consequence` - The gene and its state that will be set if the antecedents are fulfilled

`startStage` - The cell stage (number of cells present) at which this rule starts being considered

`endStage` - The cell stage at which this rule stops being considered

Method Detail

getAntecedents

public Gene[] **getAntecedents**()

---

getConsequence

public Gene **getConsequence**()

---

getStartStage

public int **getStartStage**()

---

getEndStage

public int **getEndStage**()

dataStructures
Class ConsList

java.lang.Object

└ **dataStructures.ConsList**

---

public class **ConsList**

extends java.lang.Object

Field Summary

java.util.List<Consequence>	**AandC**
java.util.List<Consequence>	**startLate**

Constructor Summary

ConsList()
Constructor for a ConsList object Just populates AandC and startLate from the CSV

Method Summary

void	**readAandCInfo**(java.lang.String file) Parses a CSV to create the antecedent and consequence rules

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

AandC

public java.util.List<Consequence> **AandC**

---

startLate

public java.util.List<Consequence> **startLate**

Constructor Detail

ConsList

public **ConsList**()

Constructor for a ConsList object Just populates AandC and startLate from the CSV

Method Detail

readAandCInfo

public void **readAandCInfo**(java.lang.String file)

Parses a CSV to create the antecedent and consequence rules

Parameters:

`file` - The name of the CSV file as a string

dataStructures
Class Coordinates

java.lang.Object

└ **dataStructures.Coordinates**

---

public class **Coordinates**

extends java.lang.Object

Field Summary

private Compartment	**AP**
private Compartment	**DV**
private Compartment	**LR**
private float	**x**
private float	**y**
private float	**z**

Constructor Summary

Coordinates(Compartment AP, Compartment DV, Compartment LR)

Constructor that indicates compartment
Coordinates(float x, float y, float z) Constructor that indicates location

Method Summary

Compartment	**getAP**()
Compartment	**getDV**()
Compartment	**getLR**()
float	**getSmallest**()
float	**getX**()
float	**getY**()
float	**getZ**()
Coordinates	**lengthsToScale**()

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString,
wait, wait, wait
```

Field Detail

x

private float **x**

---

y

private float **y**

---

z

private float **z**

AP

private Compartment **AP**

DV

private Compartment **DV**

LR

private Compartment **LR**

Constructor Detail

Coordinates

public **Coordinates**(float x,

float y,

float z)

Constructor that indicates location

Parameters:

`x` - The x coordinate of the location

`y` - The y coordinate

`z` - The z coordinate

Coordinates

public **Coordinates**(Compartment AP,

Compartment DV,

Compartment LR)

Constructor that indicates compartment

Parameters:

`AP` - The compartment on the anterior-posterior axis

`DV` - The compartment on the dorsal-ventral axis

`LR` - The compartment on the left-right axis

Method Detail

getX

public float **getX**()

---

getY

public float **getY**()

---

getZ

public float **getZ**()

---

getAP

public Compartment **getAP**()

---

getDV

public Compartment **getDV**()

---

getLR

public Compartment **getLR**()

---

getSmallest

public float **getSmallest**()

---

lengthsToScale

public Coordinates **lengthsToScale**()

dataStructures
Class DivisionData

java.lang.Object

└ **dataStructures.DivisionData**

---

public class **DivisionData**

extends java.lang.Object

Field Summary

private Axes	**axis**
private double	**d1Percentage**
private int	**generation**
private java.lang.String	**parent**
private int	**time**

Constructor Summary

DivisionData(java.lang.String parent, double d1Percentage, Axes axis, int time, int generation)

The constructor for a DivisionData object, which contains all the information that the cellDivision function needs to know in order to compute a division

Method Summary

Axes	**getAxis**()
double	**getD1Percentage**()
int	**getGeneration**()
java.lang.String	**getParent**()
int	**getTime**()
DivisionData	**setD1Percentage**(double d1Percentage)
DivisionData	**setTime**(int time)

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

parent

private java.lang.String **parent**

---

d1Percentage

private double **d1Percentage**

---

axis

private Axes **axis**

---

time

private int **time**

---

generation

private int **generation**

Constructor Detail

DivisionData

public **DivisionData**(java.lang.String parent,

double d1Percentage,

Axes axis,

int time,

int generation)

The constructor for a DivisionData object, which contains all the information that the cellDivision function needs to know in order to compute a division

Parameters:

`parent` - The name of the cell being divided

`d1Percentage` - The percentage of the volume that goes to d1 (between 0 and 1)

`axis` - The axis along which the cell is dividing

`time` - The time at which the division occurs

`generation` - The generation of the dividing cell

Method Detail

getParent

public java.lang.String **getParent**()

---

getD1Percentage

public double **getD1Percentage**()

---

getAxis

public Axes **getAxis**()

---

getTime

public int **getTime**()

---

setD1Percentage

public DivisionData **setD1Percentage**(double d1Percentage)

---

setTime

public DivisionData **setTime**(int time)

---

getGeneration

public int **getGeneration**()

dataStructures
Class Gene

java.lang.Object

└ **dataStructures.Gene**

---

public class **Gene**

extends java.lang.Object

Field Summary

(package private) LocationData	**changes**
(package private) Coordinates	**location**
private java.lang.String	**name**
private java.util.List<Consequence>	**relevantCons**
private GeneState	**state**

Constructor Summary

Gene(java.lang.String name, GeneState state)

Cnstructor for a "simple gene" which only has name and state - used to avoid storing excess info in antecedents and consequences which only need name and state

Gene(java.lang.String name, GeneState state, Coordinates location) Constructor for genes that don't change compartment
Gene(java.lang.String name, GeneState state, Coordinates location, LocationData changes) Constructor for genes that change compartment during the course of development

Method Summary

LocationData	**getChanges**()
Coordinates	**getLocation**()
java.lang.String	**getName**()
java.util.List<Consequence>	**getRelevantCons**()
GeneState	**getState**()
Gene	**populateCons**() Populates relevantCons Must be called before any gene's list of relevant consequences should be used.
Gene	**setLocation**(Coordinates location)
Gene	**setState**(GeneState state)

Gene	**updateCons**(int time) Updates the relevantCons list to remove consequences whose time period is over, or add new ones whose time periods have begun Should be called every time step TODO inefficient now that we're reading from CSV and might not be working exactly right - see testApplyingConsequences

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

name

private java.lang.String **name**

---

state

private GeneState **state**

---

relevantCons

private java.util.List<Consequence> **relevantCons**

---

location

Coordinates **location**

---

changes

LocationData **changes**

Constructor Detail

Gene

public **Gene**(java.lang.String name,

GeneState state,

Coordinates location)

Constructor for genes that don't change compartment

Parameters:

`name` - Name of the gene

`state` - Active/inactive state of the gene (also can be unknown)

`location` - Compartment in which the gene is located

Gene

public **Gene**(java.lang.String name,

GeneState state,

Coordinates location,

LocationData changes)

Constructor for genes that change compartment during the course of development

Parameters:

`name` - Name of the gene

`state` - Active/inactive state of the gene

`location` - Compartment in which the gene is located

`changes` - Info on compartment that the gene moves to and what time the move occurs

---

Gene

public **Gene**(java.lang.String name,

GeneState state)

Cnstructor for a "simple gene" which only has name and state - used to avoid storing excess info in antecedents and consequences which only need name and state

Parameters:

`name` - Name of the gene

`state` - Active/inactive state of the gene

Method Detail

populateCons

public Gene **populateCons**()

Populates relevantCons Must be called before any gene's list of relevant consequences should be used. Otherwise this list will be null. Only the gene instances that are in a cell's gene list generally need to call this

Returns:

The gene with its populated list

---

updateCons

public Gene **updateCons**(int time)

Updates the relevantCons list to remove consequences whose time period is over, or add new ones whose time periods have begun Should be called every time step TODO inefficient now that we're reading from CSV and might not be working exactly right - see testApplyingConsequences

Parameters:

`time` - The cell stage that the simulation is currently at (number of cells present)

Returns:

The gene with its relevantCons list updated

---

getName

public java.lang.String **getName**()

---

getState

public GeneState **getState**()

---

getRelevantCons

public java.util.List<Consequence> **getRelevantCons**()

---

getLocation

public Coordinates **getLocation**()

---

getChanges

public LocationData **getChanges**()

---

setState

public Gene **setState**(GeneState state)

---

setLocation

public Gene **setLocation**(Coordinates location)

dataStructures
Class GeneState

java.lang.Object

└ **dataStructures.GeneState**

---

public class **GeneState**

extends java.lang.Object

Field Summary

private double	**firstUsed**
private boolean	**on**
private boolean	**unknown**

Constructor Summary

GeneState()
Constructor for a geneState that is unknown indefinitely

GeneState(boolean on)
Constructor for a geneState whose state is known

GeneState(double firstUsed)
Constructor for a geneState whose state is unknown, but becomes known at a certain time

Method Summary

double	**getFirstUsed**()
boolean	**isOn**()
boolean	**isUnknown**()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

on

private boolean **on**

---

unknown

private boolean **unknown**

---

firstUsed

private double **firstUsed**

Constructor Detail

GeneState

public **GeneState**(boolean on)

Constructor for a geneState whose state is known

Parameters:

`on` - True if the gene is active

GeneState

public **GeneState**(double firstUsed)

Constructor for a geneState whose state is unknown, but becomes known at a certain time

Parameters:

`firstUsed` - The time at which the state becomes known

GeneState

public **GeneState**()

Constructor for a geneState that is unknown indefinitely

Method Detail

isOn

public boolean **isOn**()

---

isUnknown

public boolean **isUnknown**()

---

getFirstUsed

public double **getFirstUsed**()

dataStructures
Class LocationData

java.lang.Object

└ **dataStructures.LocationData**

---

public class **LocationData**

extends java.lang.Object

Field Summary

(package private) java.lang.String	**changeDivision**
(package private) Coordinates	**changedLocation**
(package private) Coordinates	**initialLocation**

Constructor Summary

LocationData(Coordinates initialLocation, Coordinates changedLocation, java.lang.String changeAfterDivision) Constructor for a locationData object - holds information about genes that change location

Method Summary

java.lang.String	**getChangeDivision**()

Coordinates	**getChangedLocation**()
Coordinates	**getInitialLocation**()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

initialLocation

Coordinates **initialLocation**

---

changedLocation

Coordinates **changedLocation**

---

changeDivision

java.lang.String **changeDivision**

Constructor Detail

LocationData

public **LocationData**(Coordinates initialLocation,

Coordinates changedLocation,

java.lang.String changeAfterDivision)

Constructor for a locationData object - holds information about genes that change location

Parameters:

`initialLocation` - The location of the gene at the time of creation of the shell

`changedLocation` - The location that it changes to

`changeAfterDivision` - The name of the cell who division triggers the change in location

Method Detail

getInitialLocation

public Coordinates **getInitialLocation**()

---

getChangedLocation

public Coordinates **getChangedLocation**()

---

getChangeDivision

public java.lang.String **getChangeDivision**()

test
Class MethodsTests

java.lang.Object

└ **test.MethodsTests**

---

public class **MethodsTests**

extends java.lang.Object

Field Summary

(package private) Shell	**testShell**
(package private) BasicVisual	**testVis**

Constructor Summary

MethodsTests()

Method Summary

void	**enumTests**()
void	**firstDivision**()

void	**instancesTest**()
void	**moreInstancesTests**()
void	**mutationsTest**()
void	**par1MutantTest**()
void	**par2MutantTest**()
void	**par3MutantTest**()
void	**par4MutantTest**()
void	**par5MutantTest**()
void	**perCellMutationsTest**()
void	**perShellMutationTest1**()
void	**perShellMutationTest2**()
void	**randomTests**()

void	**runEventsQueue**()
void	**testApplyingConsequences**()
void	**testGeneInitiation**()
void	**testHashMapInstances**()
void	**testInheritance**()
void	**testInstantiation**()
void	**testReadingCSV**()
void	**testTimeLapse**()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

testVis

BasicVisual **testVis**

---

testShell

Shell **testShell**

Constructor Detail

MethodsTests

public **MethodsTests**()

Method Detail

firstDivision

public void **firstDivision**()

---

runEventsQueue

public void **runEventsQueue**()

---

testGeneInitiation

public void **testGeneInitiation**()

---

testApplyingConsequences

public void **testApplyingConsequences**()

---

testTimeLapse

public void **testTimeLapse**()

---

enumTests

public void **enumTests**()

---

randomTests

public void **randomTests**()

---

testInheritance

public void **testInheritance**()

---

instancesTest

public void **instancesTest**()

---

mutationsTest

public void **mutationsTest**()

---

perShellMutationTest1

public void **perShellMutationTest1**()

---

perShellMutationTest2

public void **perShellMutationTest2**()

---

perCellMutationsTest

public void **perCellMutationsTest**()

---

moreInstancesTests

public void **moreInstancesTests**()

---

par1MutantTest

public void **par1MutantTest**()

---

par2MutantTest

public void **par2MutantTest**()

---

par3MutantTest

public void **par3MutantTest**()

---

par4MutantTest

public void **par4MutantTest**()

---

par5MutantTest

public void **par5MutantTest**()

---

testInstantiation

public void **testInstantiation**()

---

testReadingCSV

public void **testReadingCSV**()

---

testHashMapInstances

public void **testHashMapInstances**()

dataStructures
Class RGB

java.lang.Object

└ **dataStructures.RGB**

---

public class **RGB**

extends java.lang.Object

Field Summary

private int	**blue**
private int	**green**
private int	**red**

Constructor Summary

RGB(int red, int green, int blue) Constructor for an RGB object

Method Summary

int	**getBlue**()

int	**getGreen**()
int	**getRed**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

red

private int **red**

---

green

private int **green**

---

blue

private int **blue**

Constructor Detail

RGB

public **RGB**(int red,

int green,

int blue)

Constructor for an RGB object

Parameters:

`red` - The 0-255 value of the red channel

`green` - The 0-255 value of the green channel

`blue` - The 0-255 value of the blue channel

Method Detail

getRed

public int **getRed**()

---

getGreen

public int **getGreen**()

---

getBlue

public int **getBlue**()

dataStructures
Class Shell

java.lang.Object

└ **dataStructures.Shell**

---

public class **Shell**

extends java.lang.Object

Field Summary

private java.util.HashMap<java.lang.String,Cell>	**cells**
ColorMode	**colorMode**
private Coordinates	**dimensions**
private java.util.HashMap<java.lang.String,DivisionData>	**divisions**
java.util.HashMap<java.lang.String,java.lang.Boolean>	**mutants**
float	**mutationProb**
(package private) int	**simTime**
java.util.HashMap<java.lang.String,Gene>	**startGenes**

(package private) BasicVisual	**window**

Constructor Summary

Shell(BasicVisual window, java.util.HashMap<java.lang.String,java.lang.Boolean> mutants) Constructor for a cell - initializes everything

Method Summary

Cell	**calcMutation**(Cell c) **Deprecated.**
RGB	**cellColorFate**(java.util.HashMap<java.lang.String,Gene> genes) color codes based on cell fate, which is determined by the states of various genes
RGB	**cellColorLineage**(java.lang.String cellName) color codes based on lineage, which can be determined from the cell name
RGB	**cellColorPars**(java.util.HashMap<java.lang.String,Gene> genes) Color codes cells based on what par proteins they contain
CellChangesData	**cellDivision**(DivisionData data) simulates division of cell by calculating new names, centers, dimensions, and gene states of daughter cells daughter1 is always the more

	anterior, dorsal, or right child
java.util.HashMap<java.lang.String,Gene>	**childGenes**(java.lang.String parent, Axes axis, boolean daughter1) calculates the genes that a child will contain; to be called during cell division
void	**drawAllCells**() Draws all cells present in the shell to the screen
java.util.HashMap<java.lang.String,Cell>	**getCells**()
Coordinates	**getDimensions**()
java.util.HashMap<java.lang.String,DivisionData>	**getDivisions**()
java.lang.String	**nameCalc**(java.lang.String parent, Axes axis, boolean d1) Determines the name of a daughter cell based on what the parent cell is and what axis it's dividing along
java.util.HashMap<java.lang.String,Gene>	**par1Mutations**(java.util.HashMap<java.lang.String,Gene> genes) Calculates mutations that occur in a cell due to par1 being mutant
java.util.HashMap<java.lang.String,Gene>	**par2Mutations**(java.util.HashMap<java.lang.String,Gene> genes) Calculates mutations that occur in a cell due to par2 being mutant
java.util.HashMap<java.lang.Stri	**par3Mutations**(java.util.HashMap<java.lang.String,Gene> genes)

ng,Gene>	Calculates mutations that occur in a cell due to par3, par6, or pkc-3 being mutant (these mutants all behave the same way)
java.util.HashMap<java.lang.String,Gene>	**par4Mutations**(java.util.HashMap<java.lang.String,Gene> genes) Calculates mutations that occur in a cell due to par4 being mutant
java.util.HashMap<java.lang.String,Gene>	**par5Mutations**(java.util.HashMap<java.lang.String,Gene> genes) Calculates mutations that occur in a cell due to par5 being mutant
java.util.HashMap<java.lang.String,Gene>	**perCellMutations**(java.util.HashMap<java.lang.String,Gene> genes) Calculates mutations for each cell
void	**perShellMutations**() Calculates mutations for the overall shell
java.util.HashMap<java.lang.String,DivisionData>	**readEventsQueue**(java.lang.String file) Reads info about the events queue from CSV
java.util.HashMap<java.lang.String,Gene>	**readGeneInfo**(java.lang.String file) Populates the initial gene list from a CSV
void	**timeStep**() runs cell timeStep on each cell and then checks for cell divisions
void	**updateColorMode**() Recolors all cells to match a new color mode

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

window

BasicVisual **window**

---

cells

private java.util.HashMap<java.lang.String,Cell> **cells**

---

dimensions

private Coordinates **dimensions**

---

divisions

private java.util.HashMap<java.lang.String,DivisionData> **divisions**

---

simTime

int **simTime**

---

mutationProb

public float **mutationProb**

---

startGenes

public java.util.HashMap<java.lang.String,Gene> **startGenes**

---

mutants

public java.util.HashMap<java.lang.String,java.lang.Boolean> **mutants**

---

colorMode

public ColorMode **colorMode**

Constructor Detail

Shell

public **Shell**(BasicVisual window,

java.util.HashMap<java.lang.String,java.lang.Boolean> mutants)

Constructor for a cell - initializes everything

Parameters:

`window` - The PApplet in which the shell will be drawn

`mutants` - The user's choice for which genes should be mutated in this shell

Method Detail

getCells

public java.util.HashMap<java.lang.String,Cell> **getCells**()

---

getDimensions

public Coordinates **getDimensions**()

---

getDivisions

public java.util.HashMap<java.lang.String,DivisionData> **getDivisions**()

---

nameCalc

public java.lang.String **nameCalc**(java.lang.String parent,

Axes axis,

boolean d1)

Determines the name of a daughter cell based on what the parent cell is and what axis it's dividing along

Parameters:

`parent` - The name of the cell that is dividing

`axis` - The axis along which the cell is dividing

`d1` - Indicates whether we are calculating the name of daughter1 or daughter2

Returns:

The name that the corresponding daughter cell should have

---

childGenes

public java.util.HashMap<java.lang.String,Gene> **childGenes**(java.lang.String parent,

Axes axis,

boolean daughter1)

calculates the genes that a child will contain; to be called during cell division

Parameters:

`parent` - the parent that is dividing

`axis` - the axis along which the division is occurring

`daughter1` - true if we are calculating genes for daughter1, false if we're calculating for daughter2

Returns:

the genes that the child will contain

---

cellDivision

public CellChangesData **cellDivision**(DivisionData data)

simulates division of cell by calculating new names, centers, dimensions, and gene states of daughter cells daughter1 is always the more anterior, dorsal, or right child

Parameters:

`data` - The divisionData for the cell that is dividing; contains name, axis, percentages, etc.

Returns:

Data on which cells are now gone or new cells that were created

---

cellColorPars

public RGB **cellColorPars**(java.util.HashMap<java.lang.String,Gene> genes)

Color codes cells based on what par proteins they contain

Parameters:

`genes` - The geneslist of the cell to be colored

Returns:

The RGB value of the cell's color

---

cellColorLineage

public RGB **cellColorLineage**(java.lang.String cellName)

color codes based on lineage, which can be determined from the cell name

Parameters:

`genes` - The name of the cell to be colored

Returns:

The RGB value of the cell's color

---

cellColorFate

public RGB **cellColorFate**(java.util.HashMap<java.lang.String,Gene> genes)

color codes based on cell fate, which is determined by the states of various genes

Parameters:

`genes` - The geneslist of the cell to be colored

Returns:

The RGB value of the cell's color

---

updateColorMode

public void **updateColorMode**()

Recolors all cells to match a new color mode

---

drawAllCells

public void **drawAllCells**()

Draws all cells present in the shell to the screen

---

timeStep

public void **timeStep**()

runs cell timeStep on each cell and then checks for cell divisions

---

calcMutation

@Deprecated

public Cell **calcMutation**(Cell c)

Deprecated.

---

perCellMutations

public java.util.HashMap<java.lang.String,Gene>
perCellMutations(java.util.HashMap<java.lang.String,Gene> genes)

Calculates mutations for each cell

Parameters:

`genes` - The genes that the cell contains

Returns:

The updated list of genes with mutations calculated

---

perShellMutations

public void **perShellMutations**()

Calculates mutations for the overall shell

---

par1Mutations

public java.util.HashMap<java.lang.String,Gene>
par1Mutations(java.util.HashMap<java.lang.String,Gene> genes)

Calculates mutations that occur in a cell due to par1 being mutant

Parameters:

`genes` - the cell's genes

Returns:

the updated genes with effects from mutation

---

par2Mutations

public java.util.HashMap<java.lang.String,Gene>
par2Mutations(java.util.HashMap<java.lang.String,Gene> genes)

Calculates mutations that occur in a cell due to par2 being mutant

Parameters:

`genes` - the cell's genes

Returns:

the updated genes with effects from mutation

---

par3Mutations

public java.util.HashMap<java.lang.String,Gene>
par3Mutations(java.util.HashMap<java.lang.String,Gene> genes)

Calculates mutations that occur in a cell due to par3, par6, or pkc-3 being mutant (these mutants all behave the same way)

Parameters:

`genes` - the cell's genes

Returns:

the updated genes with effects from mutation

---

par4Mutations

public java.util.HashMap<java.lang.String,Gene>
par4Mutations(java.util.HashMap<java.lang.String,Gene> genes)

Calculates mutations that occur in a cell due to par4 being mutant

Parameters:

`genes` - the cell's genes

Returns:

the updated genes with effects from mutation

---

par5Mutations

public java.util.HashMap<java.lang.String,Gene>
par5Mutations(java.util.HashMap<java.lang.String,Gene> genes)

Calculates mutations that occur in a cell due to par5 being mutant

Parameters:

`genes` - the cell's genes

Returns:

the updated genes with effects from mutation

---

readGeneInfo

public java.util.HashMap<java.lang.String,Gene> **readGeneInfo**(java.lang.String file)

Populates the initial gene list from a CSV

Parameters:

`file` - the name of the CSV as a string

Returns:

The genes list as populated

---

readEventsQueue

public java.util.HashMap<java.lang.String,DivisionData> **readEventsQueue**(java.lang.String file)

Reads info about the events queue from CSV

Parameters:

`file` - the name of the CSV as a string

Returns:

The events queue as populated