

Simulation of Chicken-limb Growth with Irregular Domain Shape

Kedar Aras, Trevor Cickovski, David Cieslak, Chengbang Huang

Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556, United States

karas@nd.edu, tcickovs@nd.edu, dcieslak@nd.edu, chuang1@nd.edu

Abstract—Morphogenesis governs the formation of patterns in embryonic cell development, which eventually differentiate into bone and organs. Patterning instabilities that occur during morphogenesis can be described by computational models, such as the Cellular Potts Model (CPM). Multiscale, experimentally-motivated simulations have successfully used the CPM to reproduce morphological phenomena in the cellular slime mold *Dictyostelium discoideum* [6], [7], vertebrate neurulation [8] and convergent extension [9]. This model has been incorporated into the three-dimensional framework CompuCell3D which runs simulations of morphogenesis on rectangular grids. While these grids validate accuracy of the model, they are not biologically realistic. Models dealing with organogenesis, e.g., the 2D continuum model of chicken limb development in Hentschel et al. [5], treat both cells and morphogens as continuous fields. To improve this, we extend CompuCell3D to accommodate irregularly shaped grids and modify the CPM implementation to discard pixels outside this grid. We use software design methodologies to keep the extensibility of the framework. We implement domain growth on irregular grids by both extending a density-dependent growth algorithm implemented in CompuCell3D, and by devising an algorithm based on a dynamic sequence of irregular grids. Finally, we validate our algorithms and design through a three-dimensional avian (chicken) limb bud simulation.

Index Terms—Morphogenesis, Cellular Potts Model (CPM), boundary condition, irregular shape

I. INTRODUCTION

Morphogenesis is a set of processes which describe pattern formation in embryonic bone and organ development. Some of these processes generate instabilities that can be modeled mathematically. One such known mathematical model is the Cellular Potts Model [1]. This model represents cells on a mathematical lattice, with each lattice site containing an integer *index*, and each unique simulated cell contains its own *index*. The central algorithm consists of a method that randomly selects a pixel in the lattice and a neighbor and proposes an index 'flip',

changing the index at the selected pixel to that of the neighbor. If it is determined that the total system energy decreases with this flip it is executed, otherwise it is executed with Monte Carlo probability.

CompuCell3D runs simulations of morphogenesis in three dimensions on rectangular grids, using a combination of the CPM coupled with PDE solvers for establishing chemical gradients. The current version of CompuCell3D works only on rectangular grids, which are biologically unrealistic. We present an approach to running the CPM on CompuCell3D on an irregular shaped grid. First, we present an algorithm for detecting, given a lattice point, whether or not it is contained within some particular irregular shape, and present a carefully generated example shape of the avian limb.

We also implement domain growth for the irregular shape, using two separate models. The current design of CompuCell3D uses a density-dependent algorithm for domain growth. System density is calculated as a measure of the number of pixels containing a cell over the total number of pixels multiplied by 100 for a percentage. Each time the system density exceeds a user-specified direction, the CPM lattice grows by a user-specified number of rows in the positive *z* direction. We extend this model to the irregular shape by restricting density calculation to the area within the irregular shape. Our second model uses a dynamic data structure along with a simulation of the migration of cells towards the evolving structure.

As CompuCell3D was built for extensibility, we must add our new features to the framework in such a way that this is not broken. We thus describe our software design methodologies for incorporation of the irregular shape algorithm(s). Finally, we validate our algorithms and methodologies through an example simulation of the avian (chicken) limb, which was the first example simulation of CompuCell3D. We

ensure that cells are restricted both in initialization and in the CPM algorithm to the irregular shape that we described. We run CompuCell3D and visualize the results using Ogle [10].

II. ALGORITHM TO STORE LIMB SURFACE

CompuCell3D handles regular domains, a rectangle for 2D and a box for 3D. In reality, the chicken limb does not look like a a rectangle (2D) or a box (3D) and has an irregular shape, see Figure 2. The irregular shape is obtained from experimental

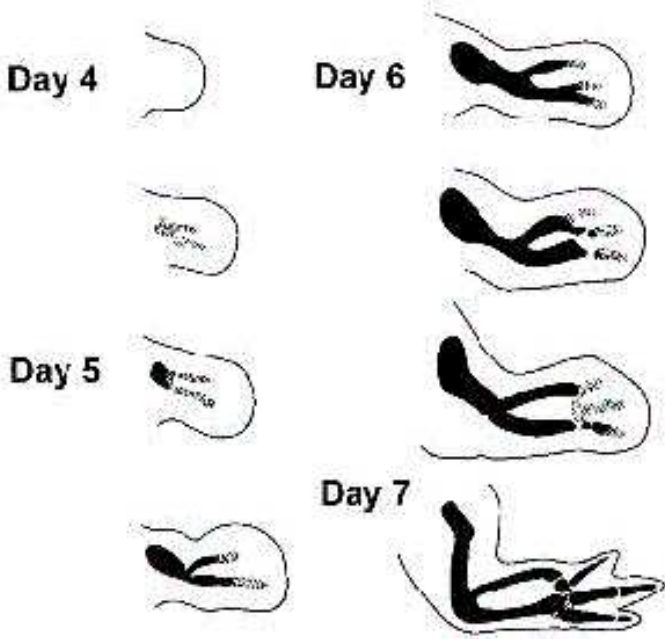


Fig. 1. Images of real chicken limb taken from experiment.

images or constructed by our own data. Since the actual limb shape from experiment is not available yet, for this project, we construct a similar irregular shape for chicken limb simulation by putting different geometric segments together. Figure 2 is the shape constructed, and it contains five pieces.

1) Bottom:

$$\frac{x^2}{a_1^2} + \frac{y^2}{b_1^2} = \frac{H_1 - z}{H_1}.$$

2) Middle:

$$\frac{x^2}{a_2^2} + \frac{y^2 - s_1 - (z - h_1) \tan(\theta_1)}{b_2^2} = \frac{H_2 \cos(\theta_1) - (z - h_1)}{H_2 \cos(\theta_1)}.$$

3) Toe 1:

$$\frac{x^2}{a_3^2} + \frac{y^2 - s_1 - s_2 - (z - h_1 - h_2) \tan(\theta_3)}{b_3^2} = \frac{H_3 \cos(\theta_3) - (z - h_1 - h_2)}{H_3 \cos(\theta_3)},$$

4) Toe 2:

$$\frac{x^2}{a_3^2} + \frac{y^2 - s_1 - s_2}{b_3^2} = \frac{H_3 - (z - h_1 - h_2)}{H_3},$$

5) Toe 3:

$$\frac{x^2}{a_3^2} + \frac{y^2 - s_1 - s_2 - (z - h_1 - h_2) \tan(-\theta_3)}{b_3^2} = \frac{H_3 \cos(-\theta_3) - (z - h_1 - h_2)}{H_3 \cos(-\theta_3)}.$$

x , y and z are unknown, and all others are parameters. The parameters must be defined properly so that each part is in the proper position.

After the irregular shape is obtained, we need to find a proper way to store the limb surface, a way that can be implemented easily and is also memory-efficient. For the Potts Model, we choose a pixel first, then decide what to do next. If the pixel is inside the limb, the regular cellular automaton is carried on; otherwise, we just simply discard the pixel we chose. Therefore, the way we store limb surface should be efficient for both cases of whether a randomly chosen pixel is inside the limb or outside of it.

We use a 2D array of vectors to store the limb surface. When we put the irregular limb shape into a regular grid which is big enough to accommodate the whole limb, the irregular shape will intersect with the grid. We use the closest grid points to approximate the intersections. We project all the intersections vertically to the xy -plane. For a vertical line (in the grid), it may intersect with the irregular shape several times, or may not intersect at all. We store the z -coordinates of all the intersections for one vertical line in sorted order in one vector at the array position given by the x and y coordinates of the vertical line. If there is no intersection, then the vector is empty.

Jordan Curve Theorem 2.1: A simple closed curve in a plane separates the plane into two regions of which it is the common boundary.

One region is said to be inside the simple closed curve, and the other region outside. To go from a

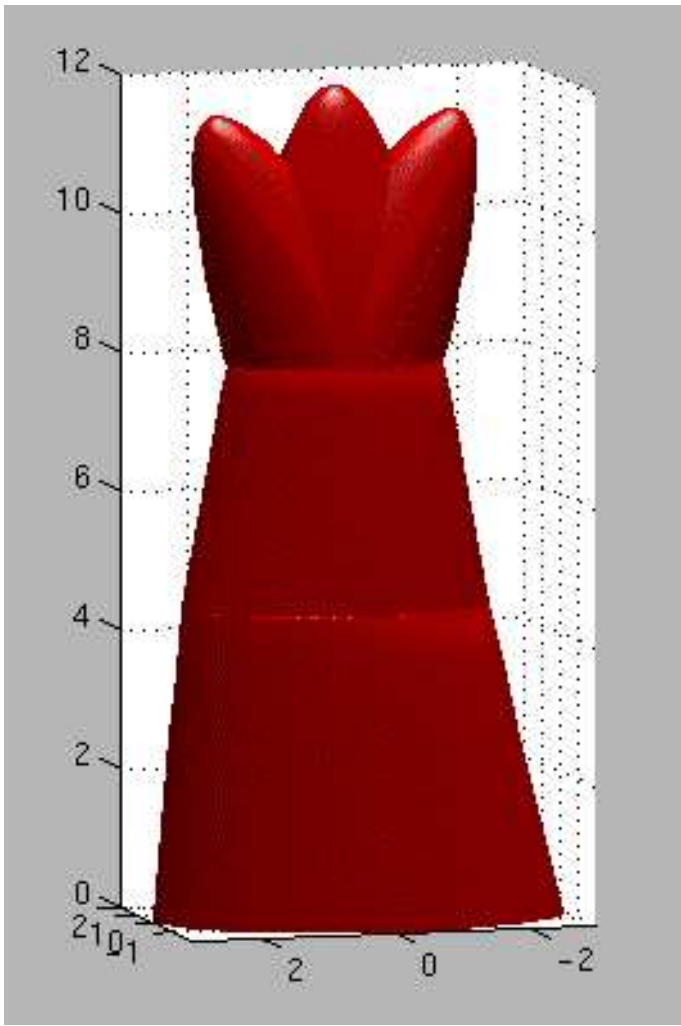


Fig. 2. The irregular shape generated by hand.

point on one side of the curve to a point on the other side, one must necessarily cross the curve. This seemingly trivial and self-evident theorem is actually difficult to prove mathematically.

This theorem can be extended to the 3D case, i.e. a simple closed surface separates the 3D space into two regions of which it is the common boundary.

Lemma 2.2: Suppose a line intersects with a simple closed surface in 3D space, then the number of intersections is even, if the tangent points (degenerate case of intersection) are not counted.

Proof: Without loss of generality, assume the line is the x-axis. Call the intersections p_1, p_2, \dots, p_n in increasing order (none of them are tangent points), see Figure 3.

If we walk along the x-axis from p_0 to p_{n+1} , every time we hit a intersection point, we either walk inside the surface or outside the surface. Since p_0 and p_{n+1} are both outside the surface, if we walk inside the

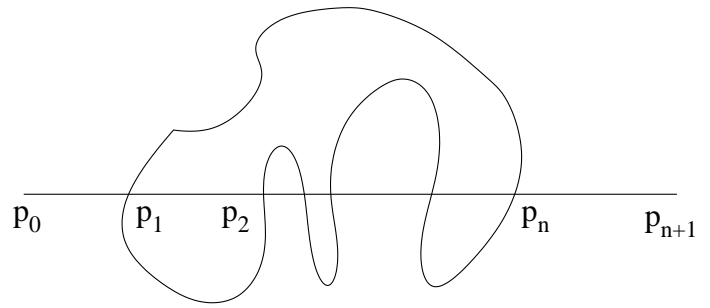


Fig. 3. x-axis intersects with a close surface.

surface, we have to walk outside later. “Inside” and “outside” are pairwise in this way, and so are the intersections. Therefore, the number of intersections n is even. Q.E.D.

Every time a pixel is picked, we go to the corresponding vector according to the x and y-coordinates of the picked pixel. If the vector is empty, then the picked pixel is outside of the limb; otherwise, we find the hypothetical location of the z-coordinate of the picked pixel in the corresponding vector, which is sorted. There are three possible cases:

- 1) The inserted z-coordinate is equal to an entry in the vector, then the picked pixel is on the boundary of the limb.
- 2) The inserted z-coordinate divides the vector into two parts of *even* sizes, then the picked pixel is *outside* the limb.
- 3) The inserted z-coordinate divides the vector into two parts of *odd* sizes, then the picked pixel is *inside* the limb.

Figure 4 is an 2D example. We use a 10×10 grid to accommodate an irregular surface. We can use an array of vectors to store the surface. The array is of size 10. The actual array of vectors is shown below the line.

III. GROWTH PROBLEM HANDLING

During the growth of a chickenlimb, it is possible that the limb shape at stage 1 S_1 is not contained completely by the next limb shape at stage 2 S_2 , see Figure 5. We cannot discard those cells outside S_2 and inside S_1 . We need to find a way to “push” those cells into S_2 .

Dillon and Othmer [4] suggested that the limb boundary gives a force on the fluid inside the limb. We can use this force to make the cells outside S_2 tend towards S_2 . The algorithm we propose is as follows. After a certain number of Potts steps are

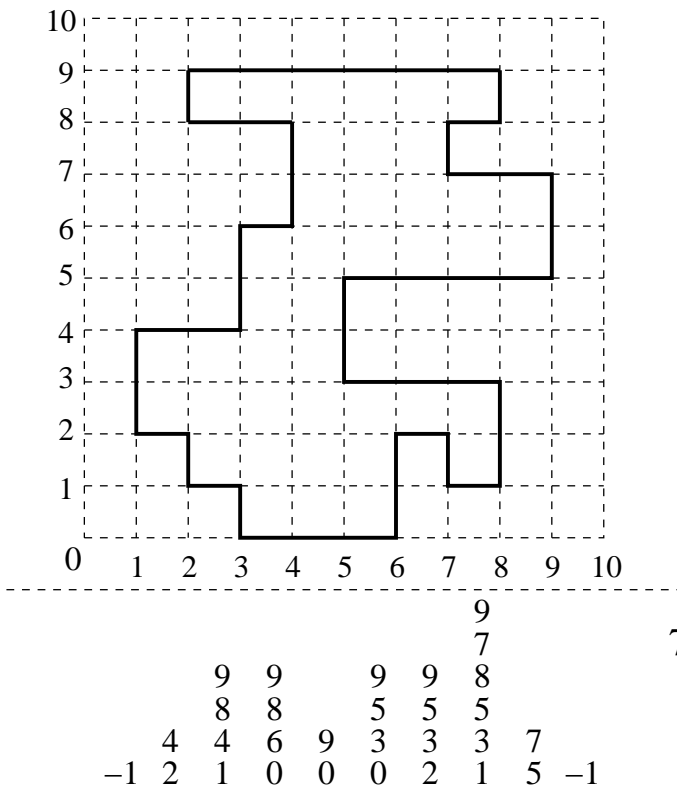


Fig. 4. An example to explain how to use an array of vectors to store the limb surface.

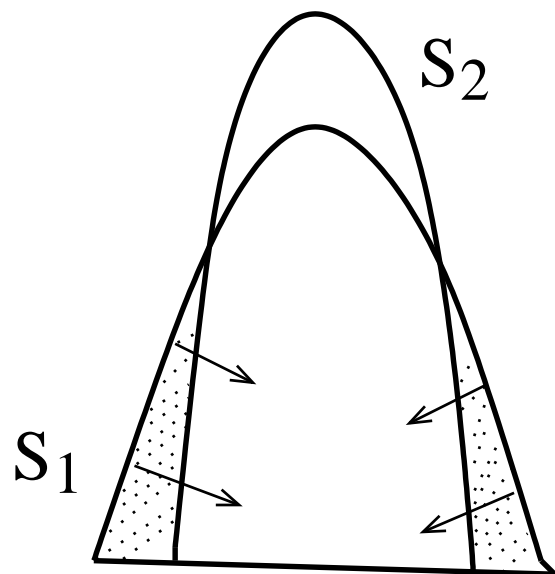
executed on the first shape S_1 , the new shape S_2 is introduced. If S_1 is not contained completely by S_2 , we run the simulation inside $S_1 \cup S_2$ (the union of the two domains). The difference is that when a boundary point p of S_1 is chosen and p is outside S_2 , we simply remove p from the domain (boundary pushes p inward), instead of doing a Monte Carlo step with p as selected pixel. If p is a cell pixel, the cell volume changes after p is removed. The removing of p will take effect when other pixels of the cell are selected later by making contribution to the volume energy. When p is removed, there are certain things we need to take care of:

- 1) Update the data structure for S_1 , the 2D array of vectors.
- 2) Update the volume for the corresponding cell, if p is a cell pixel.

IV. IMPLEMENTATION

A. The Problem

To extend CompuCell3D for compatibility with irregular shapes, the challenges from a design perspective with respect to extensibility are manifold. For example, assuming a one-to-one mapping between



7 Fig. 5. S_1 and S_2 are two limb bud shapes, and S_2 is right after S_1 . S_2 does not contain S_1 completely. The force from the boundary of S_1 pushes the cells towards S_2 .

irregular shape data structure and algorithm, we need to avoid requiring modification of the core functionality of CompuCell3D for addition (or removal) of a data structure/algorithm pair. CompuCell3D already has a method for extensibility through plugins and the configuration file. We cannot use plugins because they must inherit from one of several predefined interfaces, none of which are applicable to the task at hand. But we can use the configuration file, and in our implementation we build on our pre-implemented boundary conditions and strategies. The new irregular shape feature fits well into this design, since the job of boundary strategies is to check given a point, whether or not it is in the grid which is the same job that our algorithm must tackle. We then abstract algorithms so that extensions to the framework are simpler.

B. Design Patterns Used

We use the following design patterns to achieve code reusability, flexibility and modularity, all from [11].

- **Bridge Pattern:** The Bridge pattern decouples an abstraction from its implementation so that the two can vary independently. The bridge uses encapsulation, aggregation and inheritance to separate responsibilities into different classes. The bridge pattern is useful when not only the class itself varies often but also what the class

does. The class itself is the abstraction and what the class does is the implementation.

- **Factory Pattern:** The Factory pattern helps to model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate. It is responsible for manufacturing objects. It helps to instantiate the appropriate subclass by creating the right object from a group of related classes. The pattern promotes loose coupling by eliminating the need to bind application-specific classes into the code.
- **Singleton Pattern:** The Singleton pattern is designed to restrict instantiation of a class to one (or a few) objects. This is useful when exactly one object is needed to coordinate actions across the system. The pattern is implemented by creating a class with a method that creates a new instance of the object if one does not exist. If one does exist, it returns a reference to the object that already exists. To make sure that the object cannot be instantiated any other way the constructor is made either private or protected.
- **Strategy Pattern:** The Strategy pattern allows algorithms to be selected on-the-fly at runtime depending on conditions. It is used in situations where algorithms can be dynamically swapped in an application. Strategy is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. Strategy lets the algorithms vary independently from clients that use them. An abstract `Strategy` class is defined that the custom strategy classes can implement. Each strategy object implements the algorithm that makes it unique. The client instantiates the `Strategy` class and passes it as an argument when it calls the constructor of the context class. The client is then able to call methods on the context object and any strategy-specific methods as necessary. Because the context creates an instance of the abstract `Strategy`, the client is effectively using polymorphism to call strategy-specific methods on the context object.

Figure 6 depicts the architecture used to extend CompuCell3D to support irregular shapes.

We use the following classes:

- **Algorithm:** Defines a common interface for regular and irregular shape algorithms. The interface declares a virtual operation to determine if a selected point's neighbor lies within the grid dimensions. This approach allows each algorithm to define its own data structure for point validation and the details are abstracted from the rest of the CompuCell3D framework.
- **RegularShapeAlgorithm:** Implements the `Algorithm` interface for regular shapes. Essentially, the point is checked to see if it lies within the specified dimensions for the x, y and z axes of the grid.
- **IrregularShapeAlgorithm:** Algorithm implementation for irregular shapes. The data structure (a 2-dimensional array of vectors is used for simplicity and efficiency) is populated with points from an input file specified as a configuration parameter.
- **AlgorithmFactory:** Responsible for creating appropriate algorithm instances. The type of algorithm to be used is specified as a parameter in the configuration file. Thus the CompuCell3D framework can use an algorithm without having to actually specify the type.
- **Boundary:** Declares a common interface for applying boundary conditions to a given simulation. The interface has a virtual operation to apply a boundary condition to a point. The operation returns true if the condition is applied successfully. Boundary conditions are applied independently on each of the three axes for a given simulation.
- **NoFluxBoundary:** Implements the `NoFlux` boundary condition. The condition enforces the grid dimensions to be absolute and finite. In essence, it rejects all the points that lie outside the boundary. This translates into returning false every time the condition is applied to a point that lies outside the dimensions. `NoFluxBoundary` is the default boundary condition in CompuCell3D unless specified otherwise in the configuration file. `NoFlux` boundary is the only condition supported for irregular shapes at this time.
- **BoundaryFactory:** Responsible for creation of boundary condition instances. The configuration file specifies the conditions to be applied to each axis. This abstraction allows for different boundary conditions to be plugged in or out without affecting the underlying CompuCell3D framework.

- **BoundaryStrategy:** BoundaryStrategy is implemented as a singleton object and therefore only one global instance is available at any point during the simulation. It encapsulates the selection of algorithms and appropriate boundary conditions for use by the CompuCell3D framework. By virtue of being a singleton, it ensures that there is only one instance of the algorithm and boundary condition available for execution. The parameters from the configuration file are passed to the factories to create appropriate algorithm and boundary condition instances. BoundaryStrategy is invoked every time a neighbor needs to be retrieved for a given point. BoundaryStrategy goes about the selection process as follows:

- Retrieve a neighbor for the point.
- Invoke the algorithm to determine if the point lies inside the grid dimensions.
- If the point lies inside, it is returned.
- If the point lies outside, the boundary conditions are applied.
- If the boundary condition is applied successfully, the resulting point is returned as a neighbor.
- If the boundary conditions cannot be applied successfully, the point is discarded and a new neighbor is selected. The process is repeated until a valid neighbor is found and returned.

A snippet from the CompuCell3D configuration file illustrating an irregular shape simulation is shown in Program 1.

The configuration file is read as follows:

- **Dimensions:** Superimposed rectangular grid dimensions. (61*41*121)
- **Steps:** Iteration count for the simulation. (10)
- **Temperature:** A simulation constant. (2)
- **Flip2DimRatio:** Number of flip attempts per simulation step. (61*41*121*2 = 605242)
- **Shape:** Type of simulation shape (Irregular). If not specified, “Regular” is used as the default value.
- **Algorithm:** Type of algorithm to use (Chengbang). The CompuCell3D framework currently supports only two algorithms. If not specified, “Default” is used as a base value. The Default value is applicable to regular shapes and Chengbang is applicable to irregular shapes.

```

<Potts>
  <Dimensions x='61' y='41' z='121' />
  <Steps> 10 </Steps>
  <Temperature> 2 </Temperature>
  <Flip2DimRatio> 2 </Flip2DimRatio>
  <Shape Algorithm='Chengbang'
    File='IrregularShape.dat' />
    Irregular
  </Shape>
  <Boundary_x> NoFlux </Boundary_x>
  <Boundary_y> NoFlux </Boundary_y>
  <Boundary_z> NoFlux </Boundary_z>
</Potts>

```

Program 1. Potts parameters for the irregular shape. We have entitled our algorithm Chengbang, reading the irregular shape from the file IrregularShape.dat. NoFlux boundary conditions must be used on each axis because wraparound is not well-defined for irregular shapes and is needed for Periodic conditions. The superimposed rectangular grid dimensions are 61x41x121, and the simulation is run for 10 steps at a temperature of 2. 61x41x121x2 = 605242 flip attempts are made at every simulation step.

- **File:** Data points for the shape (IrregularShape.dat). Regular shapes do not require a file to be specified. For irregular shapes however, an input file containing the data points used by the algorithm is required.
- **Boundary_x:** Boundary condition to be used for the x-axis (NoFlux). Also the default value.
- **Boundary_y:** Boundary condition to be used for the y-axis (NoFlux). Also the default value.
- **Boundary_z:** Boundary condition to be used for the z-axis (NoFlux). Also the default value.

At the beginning of the simulation, the configuration file is read and the shape, algorithm, file and boundary condition parameters are passed on to the BoundaryStrategy to be used during its instantiation. As the BoundaryStrategy gets instantiated, it in turn invokes the AlgorithmFactory and BoundaryFactory and passes on the algorithm and boundary condition parameters respectively to create appropriate instances. The shape parameter is used by BoundaryStrategy to set a flag internally. This is later used to choose the appropriate code path to execute in the getNeighbor() function.

Program 1 shows pseudocode of the central Metropolis algorithm within CompuCell3D that is executed at each simulation step, with a slight modification of discarding pixels outside the irregular shape. Random points are chosen within the user-specified x, y, and z dimensions.

Note that this Metropolis algorithm also needs to find neighbors to the selected point. However, this

```

1) for i := 1 to flipattempts do
  a) pick a random point P(x, y, z);
  a) while P is not valid do
    i) pick a random point P(x, y, z);
    A) pick a random neighbor to P and
       attempt flip of P;

```

Algorithm 1. Pseudocode for the new central CPM Metropolis algorithm in CompuCell3D. There is now an extra loop which repeatedly chooses random pixels until one is found that is valid, in the case of an irregular shape, if the pixel is in the shape.

did not need changing because there was already a check in the algorithm for neighbor calculation for validity with respect to boundary conditions. Since our design just created a new algorithm for validity, any calculated neighbors would now invoke our algorithm instead of the previous default algorithm. There previously was no check for validity of the CPM selected pixel because with the default validity algorithm the selected pixel would always be fine since it is randomly chosen within the rectangular lattice dimensions. But with irregular shapes we now need this check.

C. Benefits of Our Approach

Our approach benefits the framework and most importantly the end user by focusing on extensibility. To change the irregular shape and algorithm applied to a particular simulation, or even to revert back to rectangular grids, the user need only modify a configuration file tag and does not need to adjust any CompuCell3D source. Factories [11] handle instantiating the appropriate boundary strategy and algorithm based on the user-specified values.

Since our design abstracts algorithm/data structure pairs, the addition of a new pair is also not difficult. Each algorithm/data structure pair would entail the addition of one new C++ class and appropriate implementation of a method to check if a particular point is in the grid, and a one-line addition to the algorithm factory implementation, associating the appropriate configuration file tag with its algorithm object. This yields the decoupling of algorithms/data structures from the core implementation of CompuCell3D that we desired.

V. RESULTS

Figure 7 shows the results of our validation simulation run on a 2.4 GHz Intel Xeon processor with

1 GB of RAM. The simulation took no more than a couple of minutes. There are roughly 590 cells in this simulation. The total grid dimensions were 61x41x121.

Mesenchymal cells in the avian limb shape undergo the CPM Metropolis algorithm. Lattice initialization was specified as uniform. Initialized cells were correspondingly restricted to a uniform distribution within the avian limb shape. CPM contact and volume energies are used, without any superimposed chemical fields. Mitosis is turned on. The CPM algorithm was able to successfully discard selected CPM and neighboring pixels that did not belong to the shape. In addition we see that cells do not migrate outside of the irregular shape, demonstrating that the neighbor validity check works since this implies no flip attempts are made to neighboring pixels outside the irregular shape.

We implement an extension of a constrained domain growth model that has been used in the validation of CompuCell3D [3]. This model superimposes a growth factor FGF linearly distributed in the positive z direction, which governs the division of the lattice into zones. At the very head there is an Apical Zone where no pattern formation can occur, followed by an active zone where cells are currently condensing into patterns, and a frozen zone where patterns have completely formed. Growth in this model is driven by density. Once the density crosses a threshold, cellular mitosis is turned off for a number of steps and the grid grows. Mitosis is turned back on after a certain amount of time. We apply this model to our irregular shaped domain. Although there is no condensation, we still show the domain growing in the positive z direction in the results shown in Figure 8. The apical zone is shown by red cells, and all other cells are blue.

Finally, Figure 9 shows the results of running CompuCell3D on a changing irregular shape. For this case we implement two irregular shapes. We impose shape 1, a short and wider shape from steps 0 through 50. At step 50 we impose the second irregular shape and force cells towards this new shape through the afore mentioned algorithm, by modifying the boundary of shape 1 and contributions to the volume energy. We visualize at steps 50, 74 and 150. There were 281 cells in this simulation and this remained consistent throughout.

VI. CONCLUSIONS AND FUTURE WORK

Our work has extended CompuCell3D so that its central CPM algorithm can be applied to both rectangular and irregular lattices. The new CPM algorithm successfully discards pixels that are outside any specified irregular shapes. We were able to keep the extensibility of the CompuCell3D framework, and also used an algorithm that saved storing the entire irregular shape. We implemented an extension of the domain growth algorithm in CompuCell3D for irregular shapes,

We see two immediate explorations that can be undertaken. The first is to implement a PDE reaction-diffusion model which is used in three of the four example simulations provided with the CompuCell3D framework. The two avian limb example simulations each use a slight modification of the Hentschel-Glimm [2], [3] equations, and these would likely be applicable here. With the avian limb bud in a rectangular domain, the chemical gradient provided by RD equations specified patterns to which mesenchymal cells would chemotax (or haptotax) and respond accordingly. With our new model, cartilage and bone patterning is determined by a manually encoded irregular shape, requiring a reevaluation of the biological role of superimposed chemical fields.

We also will attempt to improve our design in terms of performance. For example, currently if a pixel is chosen in the CPM that is outside of the irregular shape it is immediately discarded and repeated attempts are made until one is chosen within the shape, resulting in a performance penalty. The same situation occurs when neighbors are calculated. In this particular situation, since the irregular avian limb shape occupies a large percentage of the superimposed 61x41x121 rectangular grid, the performance hit was low, but if this percentage is small there will be a large number of discarded pixels and neighbors, making this penalty high. We plan to look into potential ways to modify the pixel selection algorithm to be smarter in terms of what it chooses, instead of choosing completely randomly based on the full dimensions. The lazy evaluation of the CompuCell3D NeighborFinder [3], although resulting in enormous speed and memory consumption improvements, actually provides a hindrance to this goal in terms of neighbors, because it calculates neighbors with respect to the origin and then transposes, having no knowledge of the CPM

selected pixel, much less of the irregular shape. One way of providing this is through a memoization or cache of information on selected CPM or neighbor pixels and whether or not they have been inside or outside the grid on past applications of our algorithm, saving reapplication of the same algorithm and reducing the performance penalty. This cache would need to be flushed upon every change in the data structure representing our irregular shape. If we implement this we will want to change the core CPM functionality as minimally as possible.

REFERENCES

- [1] F. Graner and J. A. Glazier, Simulation of biological cell sorting using a two-dimensional extended potts model, *Phys. Rev. Lett.*, 69:2013–2016, 1992.
- [2] H. G. E. Hentschel, T. Glimm, J. A. Glazier and S. A. Newman, Dynamical mechanisms for skeletal pattern formation in the vertebrate limb, *Proc R Soc Lond B Biol Sci*, 271:1713–1722, 2004.
- [3] T. Cickovski, C. Huang, R. Chaturvedi, T. Glimm, H. Hentschel, M. Alber, J. A. Glazier, S. A. Newman, and J. A. Izaguirre, A framework for three-dimensional simulation of morphogenesis, *IEEE/ACM Trans. Comp. Biol. and Bioinformatics*, 2004. Submitted, preprint at http://www.nd.edu/~tcickovs/bare_jrnl.pdf.
- [4] R. Dillon and H. Othmer, A Mathematical Model for Outgrowth and Spatial Patterning of the Vertebrate Limb Bud, *J. theor. Biol.*, 197:295-330, 1999.
- [5] H. G. E. Hentschel, T. Glimm, J. A. Glazier and S. A. Newman, Dynamical mechanisms for skeletal pattern formation in the vertebrate limb, *Proc. R. Soc. Lond: Bio. Sciences*, 271:1713-1722, 2004.
- [6] A. F. M. Mare, P. Hogeweg, How amoeboids self-organize into a fruiting body: multicellular coordination in Dictyostelium discoideum, in *Proc. Natl. Acad. Sci. U.S.A.*, 98:3879-3883, 2001.
- [7] A. F. M. Mare, P. Hogeweg, Modelling Dictyostelium discoideum morphogenesis: the culmination, *Bull. Math. Biol.*, 64:327-353, 2002.
- [8] M. Kerszberg, J.-P. Changeux, A simple model of neurulation, *Bioessay*, 20:758-770, 1998.
- [9] M. Zajac, G. L. Jones, and J. A. Glazier, Simulating convergent extension by way of anisotropic differential adhesion, *J. Theoret. Bio.*, 222/2:247-59, 2003.
- [10] OGLE, Ogle large-scale scientific data visualizer, <http://www.cora.nwra.com/Ogle>.
- [11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.



Springs, Michigan.

Kedar Aras is a graduate student in the department of computer science and engineering at the University of Notre Dame, Notre Dame, Indiana, directed by Dr. Izaguirre. He is also a Research Analyst at Whirlpool Corporation, Benton Harbor, Michigan. His current research is in the area of software engineering and computational biology. Aras has a B.S. in computer science from Andrews University, Berrien



Dame.

Trevor Cickovski is in his third year of graduate study in the department of computer science and engineering at the University of Notre Dame, Notre Dame, Indiana, directed by Dr. Izaguirre. His current research interests include domain-specific language development, stochastic simulations of biocomplexity and software engineering. Cickovski has a B.S. in computer science from the University of Notre



Dave Cieslak is a graduate student in the department of computer science and engineering at the University of Notre Dame, Notre Dame, Indiana. Dave has a B.S. in computer science from Notre Dame.



Chengbang Huang is a Ph.D. student in the department of computer science and engineering at the University of Notre Dame, Notre Dame, Indiana, directed by Dr. Izaguirre. His current research interest is to use a multi-model framework to simulate avian limb growth. Huang has an M.S. in computer science from the University of Notre Dame.

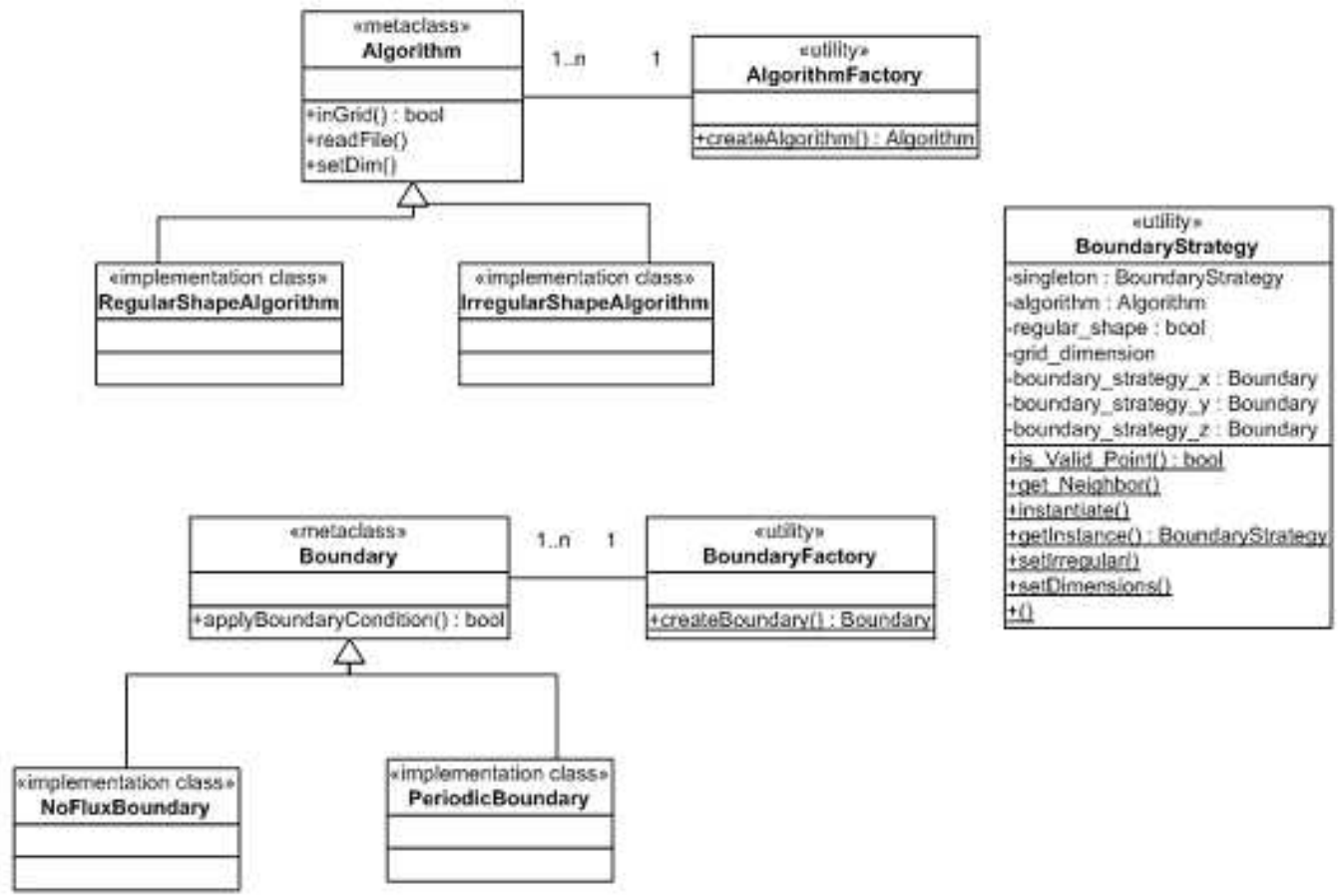


Fig. 6. UML diagram for our class architecture.

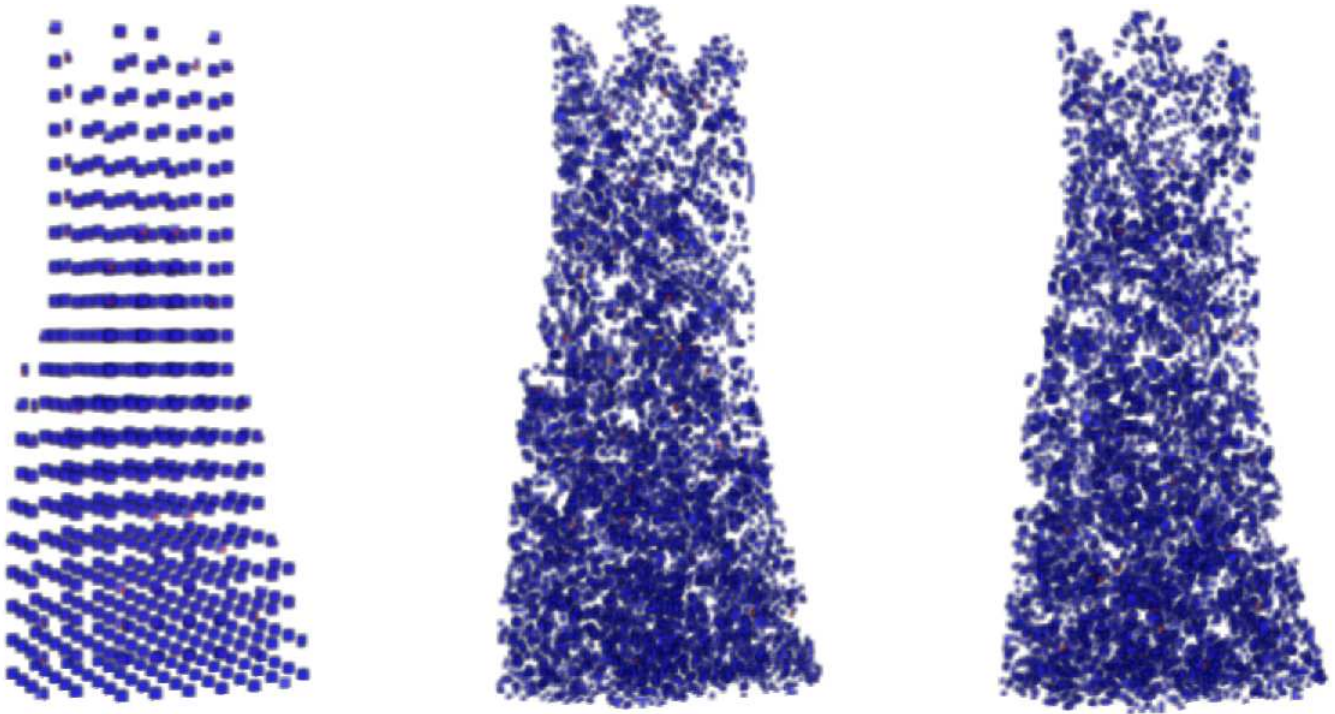


Fig. 7. A CompuCell3D run of the avian limb bud simulation with the irregular shape. Screenshots were taken at steps 0, 6 and 10, at a z-angle of 200 degrees. Cells are initially uniformly distributed within the shape.

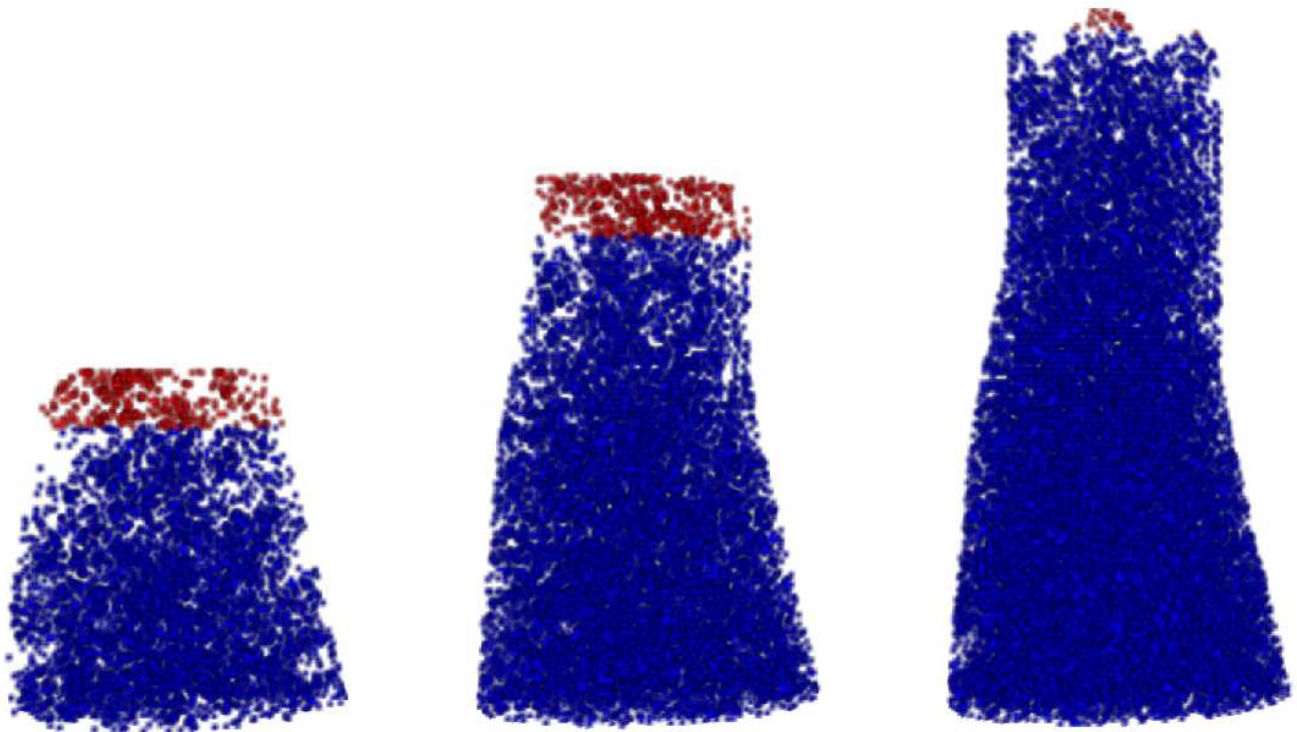


Fig. 8. Simulation of the avian limb bud with the constrained growth implementation in the CompuCell3D framework with an irregular shape. The domain grows upward in the same fashion once the density, now calculated by only counting pixels within the irregular shape, crosses a threshold. The condensing patterns in the CompuCell3D validation simulation [3] followed an Apical Zone where no condensation could occur. This region for the new domain has been colored in red. All other cells are blue. Rotation is once again at a z angle of 200 degrees. A total of 1750 steps were run.

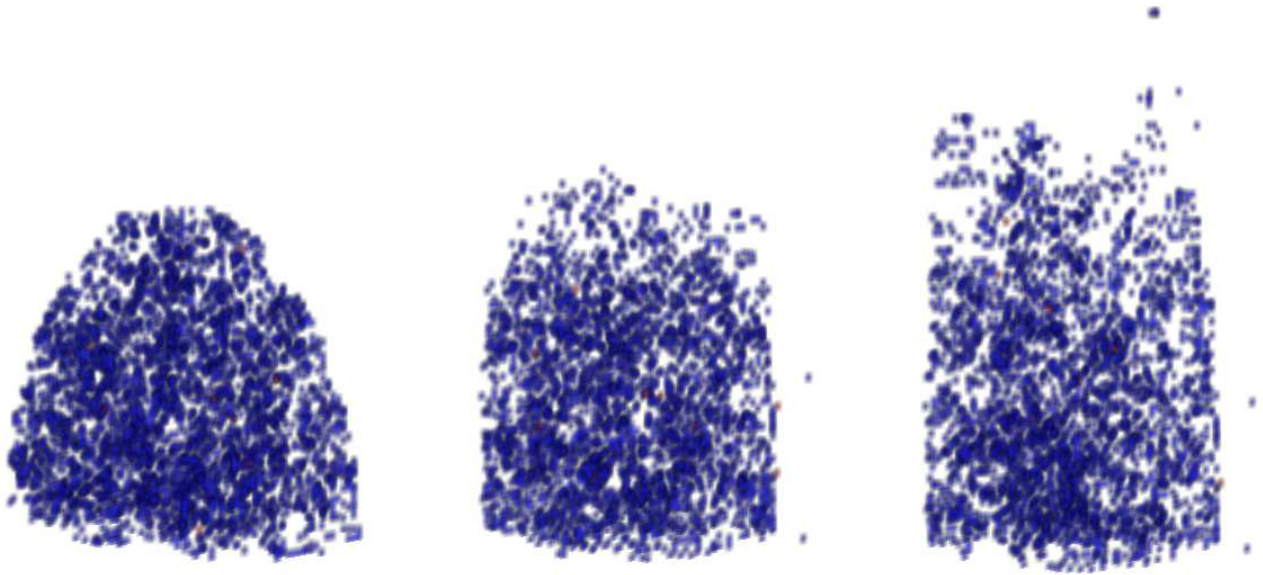


Fig. 9. A CompuCell3D run of the avian limb bud simulation with the changing data structure. There is one change in the structure. We impose the first irregular shape in steps 0-50. We impose a second, longer and narrower shape from steps 50-150. Results were visualized at steps 50, 74 and 150. Note the change in the cell formation by step 74. No cells were lost during this transition to the new shape.