Citation: Artificial Life, 2(1), 37–77, 1995

The Cell Programming Language

Pankaj Agarwal[1]
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street, New York 10012

Keywords: Modeling, Simulation, General model of development, Cell interaction, Pattern formation, Morphogenesis, Programming Language

Abstract

In developmental biology, modeling and simulation play an important role in understanding cellular behavior. This article includes a review of the general models of development that are based on cellular interaction. Inspired by these models, we suggest a simple language, the *Cell Programming Language* (CPL), to write computer programs to describe this developmental behavior. These programs are an estimate of the minimal information needed to model realistically such developmental systems. Using these programs, it is possible to simulate and visualize cell behavior. We have employed CPL to model the following: aggregation in cellular slime mold in response to a chemotactic agent, cellular segregation and engulfment due to differential adhesion, and precartilage formation in vertebrate limbs. We believe CPL is a useful tool for developing, understanding, and checking biological models.

1 Introduction

Developmental biology is the study of the process by which a single cell develops into an adult organism. A basic tenet of developmental biology, and for that matter of all science, is that structure and simplicity prevail at some level of organization. Though various phenomena may appear exceedingly complex, at some level of organization structure can be discovered. The processes involved in embryonic development are extremely complex and involve a great number of physical and chemical transformations that are not well understood. The developmental processes and stages are strikingly similar in the entire animal kingdoms. This similarity provides hope of deciphering the developmental plan. We have all wondered how a single small cell gives rise to a complex body plan with a myriad of tissues (neurons, muscle, cartilage, skin, etc.) and organs (heart, stomach, kidney, etc.), all organized intricately. An early school of thought believed that the sperm contained a miniature human fetus inside its head, and growth into an adult was purely by expansion. We now realize that the cell contains only a blueprint (genome) of the body plan. This blueprint unfolds with time, and in conjunction with the environment, physical laws, and chemical laws, gives rise to an adult organism.

All the cells of an organism are created by repeated division from a single ancestor cell. While they divide they retain the genetic information, but they come to occupy different regions. The cells determine their own function based on the genetic plan and their environment.

Cells are about halfway in the biological hierarchy (from molecules to organisms), and constitute an important structure in the understanding of development. Much has been learned about the biochemistry of cells. Organisms are comprised of cells, and all cells are basically similar. Schleiden and Schwann are credited with the first sound formulation of cell theory. They believed the cell to be a bag filled with chemical substances that interact according to the laws of physics and chemistry. Today, the cell is known to be a highly complex, but structured, entity. Cells contain cytoplasm and a nucleus. Inside the cytoplasm are membrane-bounded

[1]Currently at the Institute for Biomedical Computing, 700 South Euclid Avenue, St. Louis, MO 63110; email: *agarwal@ibc.wustl.edu*

organelles that include the mitochondria. The nucleus carries the genetic material in the form of chromosomes. The chromosomes contain DNA (deoxyribonucleic acid). The DNA macromolecules are composed of four types of nucleotides: adenine (A), cytosine (C), guanine (G), and thymine (T). DNA is a double stranded structure with each strand composed of a sequence of these nucleotides joined together by phosphodiesterase molecules. Triplets of these nucleotides specify an amino acid. The 64 (4^3) possible triplets specify just 20 different amino acids (some amino acids are specified by more than one triplet). These amino acids combine to form proteins that are half the dry weight of a cell. The amino acid sequence determines both the three-dimensional structure and function of the protein. Much of the cell biochemistry revolves around proteins. The molecular mechanisms of development are only now being discovered and are responsible for much excitement in the field.

Summaries of developmental biology can be found in any introductory text on biology and in detail in Gilbert's text on developmental biology [9]. We focus on a few developmental processes in Section 3 (in which the constituent cells play major roles) to provide motivation.

The next section discusses some models of development that have withstood the test of time and that have influenced our work. Section 3 provides short descriptions of some developmental phenomena. Models and simulations for some of these are presented in Section 7. An overview of the Cell Programming Language (CPL) is presented in Section 4. The language is explained in detail in Sections 5 and 6, which may be skipped for a preliminary reading of the article without loss of continuity.

2 Generalized Models of Development

This section includes a discussion of Turing's morphogen model, which attempts to explain pattern formation; a mechanical model proposed by Odell et al., which examines cell sheet buckling; Gordon's general model; and cellular automata and Lindenmayer systems, which are theoretical models of parallel computation. Furthermore, we briefly describe two recent models: the connectionist model, which is used to predict the segmentation genes in the *Drosophila*; and a model that aims to grow artificial neural networks. Prusinkiewicz [25] has reviewed models of morphogenesis with a significant visual component, while this article reviews the models for development with significant functionality, and in addition, presents a simple language to describe a range of developmental behavior.

2.1 Turing's Morphogen Model

Turing, in a seminal paper published in 1952, proposed a dynamic mathematical developmental model [36]. He suggested that reaction-diffusion systems obeying physical laws could lead to various stable patterns. A system of chemical substances, called morphogens, reacting together and diffusing through the tissue may account for morphogenesis. Even if the morphogens have uniform concentrations, minor random fluctuation could lead to instabilities. One of Turing's findings was that under certain conditions periodic patterns arise; high and low peaks of morphogen concentration occur, forming, for example, a series of stripes of regular width. The wavelength of these periodic patterns is determined by the diffusion constants and rates of reactions of the morphogens. Turing briefly considered real systems in terms of his model, including the *hydra* and the whorls of leaves of certain plants.

Let us consider the simplest Turing system with two morphogens called the activator (A) and the inhibitor (I). The rate of change of concentration of these chemicals with respect to time is given by the following equations:

$$\frac{\delta A}{\delta t} = k_a A - k_i I + \mu \frac{\delta^2 A}{\delta s^2}$$

$$\frac{\delta I}{\delta t} = k_a A - k_i I + K\mu \frac{\delta^2 I}{\delta s^2}$$

The $\frac{\delta^2}{\delta s^2}$ terms represent the diffusion in space. Consider the case, where the following items are true:

- k_a is positive, thus an increase in the concentration of the activator A leads to an increase in the concentration of both A and I.

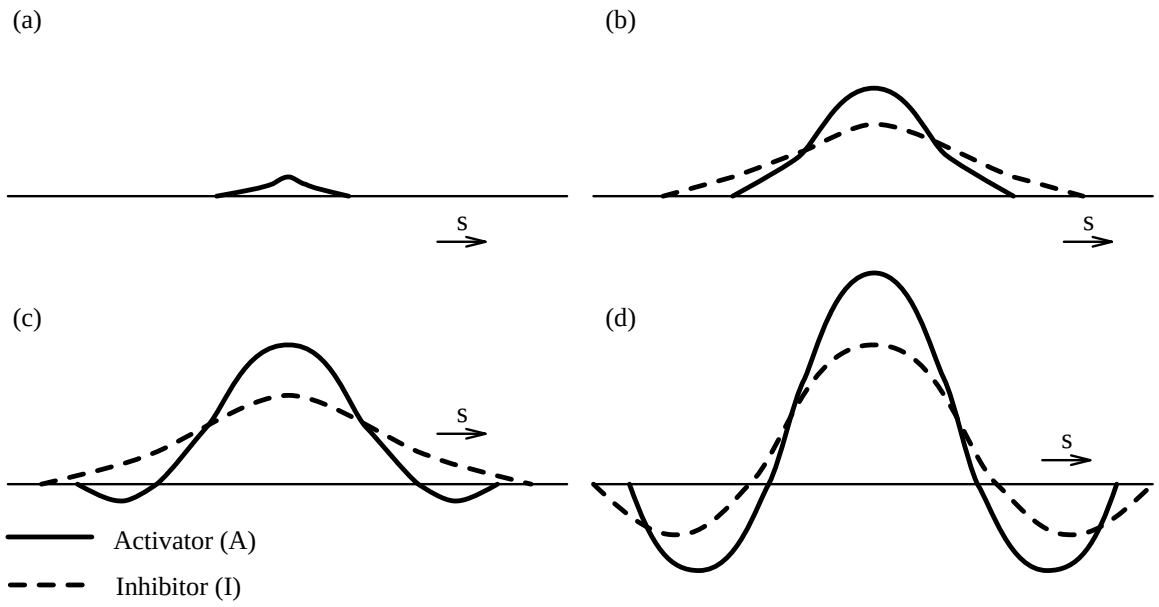


Figure 1: Regions of high and low activator concentration arise from a small perturbation in the initial uniform activator level. Each of the figures shows the activator and inhibitor values along the one-dimensional system. The subfigures a,b,c,d show the progression in time.

- k_i is positive, thus an increase in the concentration of I leads to a decrease in the concentrations of both A and I.
- $K > 1$, thus I diffuses faster than A.

For illustration, let us consider a one-dimensional system with both A and I present in uniform concentrations. Figure 1 illustrates the results of an upward perturbation in the activator value at a single point. This may occur due to random noise. This increase in the activator A leads to a greater increase in A, and an increase in I. But, because I diffuses faster than A it spreads out further as is shown in Figure 1b. Thus, the inhibitor diffuses to points in space where the activator value has not yet risen. This increase in inhibitor leads to the activator level falling below the equilibrium (Figure 1c). Therefore, regions with high and low values of the activator are established throughout the system (Figure 1d). The reactions themselves predict these extremes to be unbounded, but other physical constraints impose saturation values. Non-linear systems may also bound the extremes. In a two-dimensional system this activator-inhibitor system can lead to circular standing waves.

Meinhardt has explored in great detail various morphogen systems and the resulting stable patterns [17]. Such reaction-diffusion (activator-inhibitor) systems account for a significant share of the proposed models in developmental biology. Ouyang and Swinney have discovered that stable patterns can arise in chemical systems from reaction-diffusion [24]. The patterns form spontaneously by varying a control parameter. They observed both hexagonal and striped patterns.

We have used an activator-inhibitor system for modeling precartilage formation in vertebrate limbs (Section 7.2.5).

2.2 Mechanical Model

Odell et al. presented a mechanical model to explain the folding of embryonic epithelium [23]. This was based on hypothesized properties of the cytoskeleton. They observed that if a cell that was part of a layer (or a ring) contracted, it would stretch the other cells in the layer. Thus, stretching cells beyond a point induced a contraction resulting in a smaller than original apical (outer) surface with the volume remaining

constant. This cascade effect of reduction in apical surface would cause a buckling in the cell sheet producing an invagination, which resembles gastrulation in *Sea Urchin*, ventral furrow formation in *Drosophila*, and neuralation in amphibians. Odell et al. emphasized the importance of not assigning each cell an autonomous program of shape change:

> In this study we want to minimize both the number of complex instructions (e.g. morphogens and clocks) as well as the genetic programming required to generate morphogenetic patterns. It is, of course, conceivable that each individual cell in the blastula is genetically programmed to execute a sequence of instructions directing each movement that the cell performs as well as its precise timing. However, we regard such a view as evolutionarily implausible. [23, page 454]

2.3 Gordon's Model

Gordon [11], in 1966, suggested a general model for development. According to Gordon, an organism may be regarded as an ensemble of cells, each cell capable of making decisions based on its own state and the environment. The environment would include the configuration of cells around it, and the chemical and electrical messages (surface interactions, hormones, nerve impulses, etc.) it receives. The internal state of a cell could include its state of differentiation, a limited memory, and an internal clock. A cell could take the following actions: do nothing; reset its internal clock; differentiate; communicate with other cells; divide; expand or shrink; fuse with a neighbor; move; and die.

The interactions between cells are probabilistic due to incomplete or inaccurate information about its own state and the environment. Genetic control of development is indirect in this model. The genes presumably determine the next state function, albeit implicitly.

2.4 Cellular Automata and Lindenmayer Systems

A cellular automaton is a theoretical model of a parallel computer. It is an interconnection of identical cells. Each cell computes an output from local inputs. These inputs are received from a finite set of cells forming its neighborhood and possibly from an external source. Each cell houses a finite state machine, which is formally denoted by the 4-tuple (Q, Σ, δ, q_0). Q is a finite set of states, Σ is a finite input alphabet, δ is the transition function mapping $Q \times \Sigma$ to Q, and q_0 in Q is the initial state. The concept is the same as that of finite automata in computational theory, except that there is no final state.

A cellular automaton that allows cells to divide into daughter cells and allows the disappearance, or death, of cells is known as a dynamic cellular automaton or a *Lindenmayer* system. In other words, a Lindenmayer system is an array of interconnected automata, along with the provisions:

- The input to an automaton are the states of its neighboring automata. Automata on the border of the array may also have external (environmental) inputs. Therefore, $\Sigma \subset Q^k$, where k is the maximum number of neighbors providing input to the cell.
- The automaton array is not limited to its starting size. It can expand and contract by the reproduction or death of automata. This is modeled by modifying the transition function from $\delta : Q \times \Sigma \rightarrow Q$ to $\delta : Q \times \Sigma \rightarrow (\epsilon, Q, Q * Q)$, where ϵ corresponds to the disappearance of an automaton and $Q * Q$ to the duplication of one.

This system is of interest to theoretical biologists as a model for the growth and development of organisms. Lindenmayer used cellular automata to grow branching and non-branching filaments, which exhibit various developmental patterns, such as a constant apical pattern, non-dividing apical zone, and banded patterns [15]. He employed a one-dimensional cellular array with two neighbors for every automaton, and all the automata were identical. The state transition table for a Lindenmayer system that produces branching filaments is shown in Figure 2. In this example, the only input for each automaton is its own state.[2]

[2]This example is taken from [15]. The Figures 2, 3, and 4 have been redrawn but have the same content as the original paper.

Present state	1	2	3	4	5	6	7	8	9
Next state	2∗3	2	2∗4	2∗5	6∗5	7	8	9∗[3]	9

Figure 2: State transition table for the Lindenmayer system. ∗ indicates division. [ ] indicates a new branch.

Time	Filament
1	1
2	23
3	224
4	2225
5	22265
6	222765
7	2228765
8	2229[3]8765
9	2229[24]9[3]8765
10	2229[225]9[24]9[3]8765
11	2229[2265]9[225]9[24]9[3]8765
12	2229[22765]9[2265]9[225]9[24]9[3]8765
13	2229[228765]9[22765]9[2265]9[225]9[24]9[3]8765
14	2229[229[3]8765]9[228765]9[22765]9[2265]9[225]9[24]9[3]8765
15	2229[229[24]9[3]8765][229[3]8765]9[228765]9[22765]9[2265]9[225]9[24]9[3]8765

Figure 3: A sample calculation for 15 time steps of the automaton. A diagrammatic representation is provided in Figure 4.

Figure 3 displays a sample calculation for 15 time steps of the automaton. Figure 4 shows a diagrammatic representation of the final state of the filament. This models the developmental pattern for a particular red algae, *Callithamnion roseum*, and has the following features:

- the main filament has at its base three cells that do not bear branches;
- each successive cell above these on the main filament bears one branch;
- in all stages four cells below the tip of the main filament have no branches;
- each primary or higher order branch repeats the pattern of the main filament.

Toffoli and Margolus have described various application of cellular automata in modeling [34]. Prusinkiewicz has used Lindenmayer systems to design plants with intricate patterns [26].

2.5 Connectionist Model

Mjolsness, Sharp, and Reinitz have provided a "systematic method for discovering and expressing correlations in experimental data on gene expression and other developmental processes" [18]. This *connectionist* model uses grammar rules to model state changes, and a fixed type of differential equation to model behavior within the state. The differential equations are of the form:

$$\frac{\mathrm{d}v_i^a}{\mathrm{d}t} = g_a\left(\sum_b T^{ab} v_i^b + h^a\right) - \lambda_a v_i^a$$

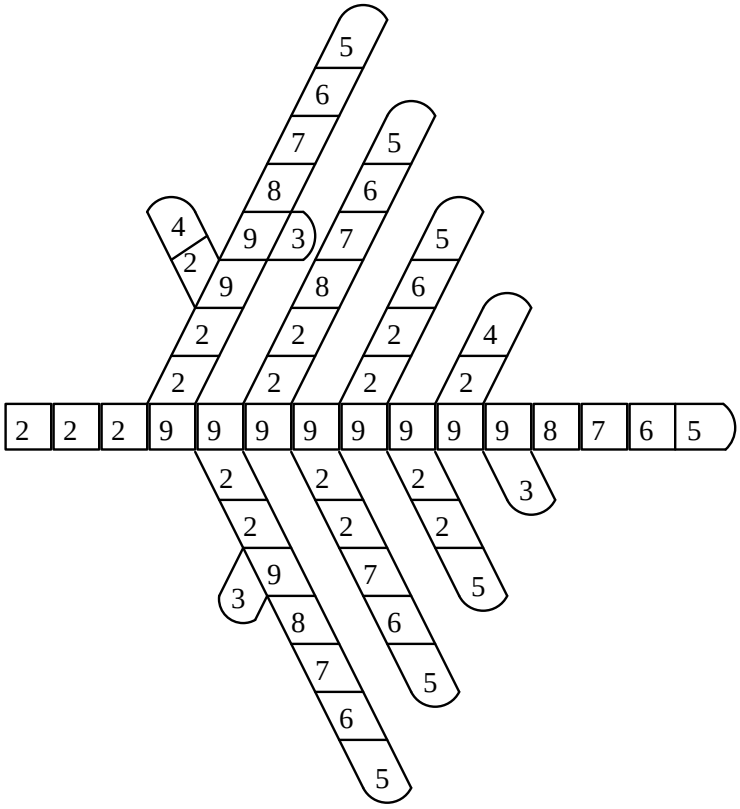


Figure 4: A branching filament obtained by using a cellular automaton after 15 time steps.

These equations describe the time dependence of the various protein (a) concentrations. v_i^a are the protein concentrations for object i (which may be a cell, nucleus, or synapse), g_a are sigmoidal threshold functions, T^{ab} are connectionist threshold elements, and h^a are the threshold terms. Both T^{ab} and h^a are adjusted (trained) to fit the experimental data. This allows them to compute the coupling between the various genes (which produce the proteins), and thus predict the concentration profiles for similar systems (mutants). They have tested the system on the segmentation genes in the *Drosophila* blastoderm.

2.6 A Model for Growing Artificial Neural Networks

Fleischer and Barr have designed a simulation system for studying multicellular pattern formation [7]. This system has been designed to simulate evolution. They eventually hope to create artificial neural networks to solve problems. The cells have a genome represented by a set of differential equations. The differential equations code for the cell's chemical, mechanical, genetic, and electrical activity. This representation eases evolution, because cells can modify genetic information (i.e. exchange some of their differential equations) by either mutation or during cell meiosis. The choice of differential equation to model cell activity permits fine control of their behavior; however, the computational cost associated with solving them restricts the simulations to a few hundred cells. The cells are modeled as two-dimensional disks with potentially different radii. They have conducted simulations which displayed cell migration, cell differentiation, gradient following, clustering, lateral inhibition, and neurite outgrowth.

3 Biological Phenomena Modeled

This section discusses some developmental behavior for which relatively simple models are available or can be designed.

3.1 Slime Mold Aggregation

Dictyostelium discoidea[3] is a free-living amoeba and is considered by some to be the hydrogen atom of developmental biology. *Dictyostelium* is a bridge between unicellular and multicellular organisms, because *Dictyostelium* cells spend portions of their life in each mode. In their unicellular existence, they eat bacteria and reproduce by binary fission. Exhaustion of food supply causes tens of thousands of these amoebae to join together, to form moving streams of cells that converge into conical mounds. These conical aggregates modify their shape to form a slug (worm-like structure). The slug migrates in search of better environmental conditions, where the cells in the slug differentiate into stalk cells and spore cells, which together form a fruiting body. The spore cells disperse, each one becoming a new amoeba. This is a simple, yet intriguing, life cycle and a rich source for models in development.

The cellular aggregation is randomly initiated. Cells do not move directly towards these random centers; rather, they join with each other to form streams; the streams converge into larger streams, and eventually all streams merge in the center. This motion is attributed to chemotaxis[4], the chemical involved is cyclic adenosine monophosphate (cAMP). There is no *dominant* cell or predetermined center; whichever amoebae happen to secrete cAMP first become aggregation centers. Other amoebae respond to the cAMP by initiating movement towards the cAMP source and by releasing cAMP of their own.[5]

3.2 Limb Skeleton Formation

The organization of the bones in the limbs is regarded as a classical example of development. What kind of process would create an increasing number of progressively smaller bones in the limb along the proximo-distal axis (from the shoulder to the fingers): a single humerus; the radius and the ulna; and the five metacarpals-digits? This basic theme plays out in almost all vertebrates with minor differences. In addition, variations are observed often enough in the number of digits to conclude that the number of these bones is not pre-ordained, but formed by some process at a certain stage of development.

3.3 Sponge Reconstitution

Sponges are simple protozoa that possess a remarkable property. Wilson [40] in 1907 observed that if a sponge is dissociated into its individual cells by passing through a sieve, the cells reaggregate to form a *functional* sponge. This reconstitution is species-specific; if cells from sponges of different species are mixed together, each of the reformed sponges contains cells only from its own species. This helped prove that cells can recognize other cells of their own kind. Later experiments by Moscona, Holtfreter, and Steinberg have established that cells from other organisms also recognize cells of their own kind, and tend to segregate when mixed with other cells [19]. These experiments established that cells have an identity, and it became possible to visualize development in terms of the constituent cells.

Steinberg [31] observed that cell aggregates from embryonic tissue have a tendency to round up *in vitro*. In fact, there are quantitative differences in this rounding up; limb bud tissue rounds up more than heart tissue, which in turn rounds up more than liver tissue. If limb bud cells and heart cells are intermixed, the limb bud cells migrate centrally. If heart cells and liver cells are intermixed, the heart cells migrate centrally (i.e. the liver cells envelop the heart cells). A transitivity in this central migration is observed; therefore, if limb bud cells and liver cells are intermixed, then the limb bud cells migrate centrally.

[3] Often referred to as the cellular slime mold.

[4] Chemotactic movement is caused by the diffusion of chemical substances through a medium. Cells may detect and move along such chemical gradients.

[5] This summary has been paraphrased from Gilbert [9].

Cell segregation, rounding up, central migration, and transitivity of migration are properties observed in non-living liquids as well. Liquids tend to round up in the absence of external forces. Some liquids, when intermixed, segregate. This segregation may involve one of the liquids being suspended as a droplet inside the other (central migration). In liquids this behavior is attributed to surface tension forces, which arise from differences in adhesion between molecules of different liquids. This led Steinberg to postulate the *differential adhesion hypothesis*, which states that cells adhere differently to one another. Accordingly, the arrangement of embryonic tissues *in vitro* can be explained by the forces generated by the surface tension due to differential adhesion [29, 30]. Nowadays, it is widely accepted that cell adhesions play a major role in the process of development [9].

4 Overview of CPL

This section contains a brief description of the design and form of the *Cell Programming Language (CPL)*. The design was motivated by studying models of developmental behavior (Section 3), and was build upon some of the ideas of Turing, Odell et al., Gordon, and Lindenmayer (Section 2).

4.1 Brief Description of CPL

A genome is the program for the development of an organism. The genome, in conjunction with the environment, determines the behavior of each cell of the organism. The program for each cell (written in CPL) plays the role of its genome.

The program for an individual cell consists of a set of states. In each state, rules are specified that determine the cell properties (i.e. shape, motility, concentrations of various molecular species, etc.). Different states of the same cell signify different phases in the cell's life. Each cell has a tissue type associated with it. Cells of the same tissue type execute the same CPL program.

We use the discrete time simulation model. At every time step, each cell executes all the instructions in its present state sequentially. All cells are assumed to be executing in parallel, with synchronization performed after every time step.

The cells are two-dimensional. Each cell has a physical location comprising a collection of discrete connected points. This physical presence imparts to the cells the attributes of area, perimeter, and neighbors (other cells). The neighbor attribute forms the basis for all intercellular communication.

The language contains features for specifying:

- the location, area, and shape of the cells;
- the concentrations of various chemicals in each cell, the equations of their catalysis, and diffusion;
- the direction and speed of cell motion;
- the rates of cell growth and division;
- cell differentiation: the evolution of cell behavior during its lifetime.

We use "cell" to mean a biological cell, and a "point" refers to a lattice point. In cellular automata literature, there is often no distinction between the two, but this distinction is critical in CPL, because cells can move around and occupy different lattice points over time.

4.2 Physical Representation of Cells

Physically, an actual cell is a solid that may be approximated by a polygonal structure with a specified area. It is generally many-sided and not necessarily convex. A cell changes shape, grows in area, divides into two, and moves, depending on its own state and the environment. The chosen model should permit all these operations, and above all be flexible to handle additions to the set of operations.

Operator	Use/Meaning
!, &&, ||	logical not, and, or
^	exponentiation
*, /, %	multiplication, division, modulo
+, −	addition, subtraction
==, != , >, <, >=, <=	relational operators
=	assignment operator

Figure 5: CPL operators

A majority of the models that have been designed for cells so far, including the one CPL uses, model the cell as two-dimensional. One such model, used by a number of researchers, treats the cell as a rigid body of fixed size and shape [19]. This model does not permit cells with either variable sizes or different shapes. An extension of this model in which each cell is modeled as an aggregate of a large number of discrete rigid objects overcomes these deficiencies, as it permits cells to have arbitrary shape and size. This is the most general discrete representation. In addition, if all cells have the same shape and size, then each cell can be represented by a single point, providing a compact representation.

In CPL, we represent each cell by a collection of discrete connected points. These points can be regarded as the points in a hexagonal lattice.[6] Each cell can occupy one or more of the lattice points. All the lattice points occupied by one cell should be connected, i.e. a cell cannot be disjoint.

The physical representation of cells is mainly a technical issue and has not for the most part influenced the design of CPL. The representation of cells could be modified without significant alterations to CPL. In particular, we could build three-dimensional models by representing each cell as a collection of discrete connected points in three-dimensional space. The representation of cells is primarily influenced by the computational frontier.

The next two sections discuss the syntax, semantics, and some of the implementation issues for the instructions in CPL.

5 CPL Syntax and Semantics

CPL reserved words appear in this paper in the `typewriter type style`. Names between "<" and ">" represent templates. <*expression*> refers to any expression; <*integer-expression*> refers to any expression evaluating to an integer; <*direction-expression*> refers to any expression evaluating to a direction; and <*instruction*> refers to any single instruction, or a block of instructions enclosed in curly braces. CPL uses curly braces in the style of the programming language C++ to mark the beginning and end of blocks, and semicolons to separate instructions [32].

In CPL, all non-reserved words (with the first character being a letter and the following characters alphanumeric or underscores) are *valid identifiers* (used for biochemical, tissue, state, and variable names). Distinctions are made between upper and lower case letters, and all the reserved words are in lower case.

A BNF grammar for CPL is available [1].

5.1 Expressions

CPL expressions mostly follow the rules of C or C++ [32], except ^ is used for exponentiation. In particular, expressions evaluating to zero are regarded as false and those not evaluating to zero as true.

CPL expressions use a subset of the language C's operators. CPL operators are listed in Figure 5.

[6] The hexagonal structure in which each lattice point has six neighbors is better defined than the four- or eight-neighbor lattice structure, as the latter violate the Jordan Curve Theorem. The theorem states that a simple closed curve should separate the image into two simple connected regions [12, page 67].

The possible operands in these expressions may be:

- constants: integer, real, and direction {of the form point(integer,integer)};
- variable names;
- biochemical names;
- cell attributes: `perimeter`, `area`, `cell_number`, `location`;
- neighbor attributes: `neighbor.direction`, `neighbor.contact_length`, `neighbor.perimeter`, `neighbor.area`, `neighbor.cell_number`, `neighbor.state` and `neighbor.tissue_type`;
- simulation attributes: `time`, `time_interval`, `steps` (`steps` = `time/time_interval`); and
- functions: `sqrt` :real→real, `int`:real→int, and `random`:(integer×integer) →integer.

The non-trivial operands are discussed in Section 5.6.

5.2 Instructions

Examples of CPL instruction sequences are highlighted by placing them in rectangular boxes. A complete CPL program is presented in the appendix.

The programs written for cells are called *tissue definitions* because the same program is used by all the cells of the same kind or tissue. In addition, there are *cell definitions*, which contain information about the starting location (and chemical concentrations) of the initial cells.

The repertoire of instructions that the cell may choose to execute is listed in this section, along with descriptions of the syntax and semantics of each of them.

- **assignment:** The assignment instruction is used to assign new values to variables. CPL has the simple assignment statement,

  ```
  clock = 1
  ```

  The positive and negative accumulator assignments (as in the language "C") are also permitted.

  ```
  clock += 1
  ```

  ```
  clock -= 1
  ```

  In addition, derivatives of biochemicals may be specified, and that assignment takes the form:

  ```
  deriv BcOne = k * BcTwo * BcThree
  ```

  `deriv` is a reserved word and indicates that the expression on the right hand side (RHS) is *added* onto the previous BcOne value (and not assigned). The RHS is implicitly multiplied by the size of the time step[7]

[7] The time step is explained in Section 5.3

in the simulation. Moreover, the addition to biochemical values is only effected at the end of the time period.[8] The above equation assumes the following reaction:

$$BcTwo + BcThree \rightarrow BcOne$$

with the rate of production of BcOne (d $BcOne/\mathrm{d}t$) being given by

$$\frac{deriv\ BcOne}{\Delta t} = k * BcTwo * BcThree$$

$$deriv\ BcOne = (k * BcTwo * BcThree) * \Delta t$$

The biochemical assignment statement (`deriv`) in CPL is similar to the above equation, though the multiplication of the RHS by Δt is implicit. Δt should be small for the equation to be valid.

The biochemical concentrations can be set to a specified value by using the straightforward assignment (without using the `deriv` reserved word). In that case, there is no implicit multiplication of the RHS by the time interval.

```
BcOne = 1.0
```

This sets the concentration of BcOne to 1.0 at the end of the next time step. This statement is primarily used to set initial concentrations.

The assignment statement has the following possible forms:

`deriv` <biochemical> = <expression>
<biochemical> = <expression>
<variable> = <expression>
<variable> += <expression>
<variable> −= <expression>

- **move:** The move instruction causes the cell to move by exchanging the location of the cell with that of a neighboring cell in the specified direction. The move is well-defined if the two cells exchanged are of equal size; however, the move is not as well-defined when cells of unequal sizes are involved.[9]

  `move` <direction-expression>

```
move direction1
```

A `move` changes the neighborhood and location of both the cells involved. This may require recomputation of values for some of the cell variables, particularly the biochemical gradients.

The direction specified is relative to the location of the cell.

- **goto:** A `goto` instruction specifies a state switch. A cell executes all the instructions in its present state at each time unit; the `goto` provides a mechanism for switching this set of instructions. Typically, gotos

[8] This is the only instruction, whose execution is, in essence, delayed to the end of the time step.
[9] In the unequal area case, the `move` conserves the areas of all the cells involved; however, their shape may not be conserved.

would be used to cycle between a set of states (such as *waitForSignal, readyToSignal*, and *signal*).[10] These states consist of a set of instructions specified by the user. There are no predefined states.

`goto` <state-name>

```
goto waitForSignal
```

The execution of the `goto` ceases execution of the rest of the program for that time step. In other words, the `goto` instruction, if executed, is the last instruction executed for a cell at any time step. At the next time step, instructions belonging to the new state are executed.

- **differentiate_to:** This instruction is a form of the `goto` instruction. The `differentiate_to` instruction can be used to specify the type of tissue the cell should differentiate into. This is employed to specify a major change, often irreversible, in the cell's life history. From the programmer's perspective, it is possible to eliminate one of the two instructions: `goto` or `differentiate_to`; however, they serve different biological purposes (cycling between a set of states and irreversible change).

  `differentiate_to` <tissue-name>

```
differentiate_to epithelial
```

  The `differentiate_to` instruction changes the CPL program that the cell executes.

  The execution of the `differentiate_to` ceases execution of the rest of the specific cell's program for that time step. At the next time step, the program for the new tissue is executed.

- **for_each_neighbor_do:** This permits the execution of an instruction (or a block of instructions) using the parameters of each of the neighboring cells in sequence. This instruction takes another instruction as its argument. Some extra read-only variables are available in the scope of the `for_each_neighbor_do`, specifically, the area of the neighbor; the direction to the neighbor; contact length with the neighbor; the tissue type and state of the neighbor; the biochemical concentrations inside the neighbor; and the variable values inside the neighbor.

  The `for_each_neighbor_do` executes the block of instructions with each neighbor in turn. It randomly picks up the first neighbor and then cycles through the remaining neighbors by going around the cell boundary. The `exit` instruction may be used to exit the loop prematurely. Nested `for_each_neighbor_do`'s are not allowed.

  `for_each_neighbor_do` <instruction>

```
for_each_neighbor_do
  deriv BcOne = (neighbor.BcOne − BcOne)/Drate;
```

The above instruction sequence illustrates a possible representation of diffusion. An amount proportional to the difference in the concentration of BcOne between the cell and the neighbor is added to the concentration of BcOne.[11] In a similar fashion, diffusion for the other biochemicals in the cell could be represented. The

[10] These states are employed in the cellular slime mold example in the appendix.

[11] The user should ensure that the cells do not cheat in diffusing biochemicals. If a cell receives some amount of a biochemical because of diffusion, another cell must lose an equal amount of it; therefore, the language user must write code ensuring the conservation of biochemicals during diffusion. The language facilitates this because all the cells access the same biochemical values at any time. It is impossible to change the biochemical value inside a cell until the end of a time step.

variable *Drate* determines the rate of diffusion. A larger *Drate* would indicate slower diffusion.

```
dirBcOne = point(0,0);
for_each_neighbor_do {
  deriv BcOne = (neighbor.BcOne - BcOne)/Drate;
  dirBcOne += neighbor.direction * (neighbor.BcOne - BcOne)/Drate;
}
```

The above instruction sequence demonstrates the accumulation of the direction of the biochemical diffusion, which is the sum of the direction to its neighbors weighted by the amount of the biochemical diffusing from the respective neighbor.

- **if-then-else:** The if-then-else instruction provides conditional execution of instructions. The conditions can depend on any of the cell attributes, including the neighbor attributes (only when `if-then-else` is used inside a `for_each_neighbor_do` or `with_neighbor_in_direction`).

  `if` <expression> <instruction>
  `if` <expression> <instruction> `else` <instruction>

  If the expression evaluates to a non-zero value, the condition is taken to be true.

```
if (area > 100) divide horizontal
```

- **exit:** The `exit` may only be used inside a `for_each_neighbor_do`, and the remaining instructions inside the `for_each_neighbor_do` (physically following the `exit`) are not executed. The execution resumes at the instruction succeeding the `for_each_neighbor_do`. `exit` may be used to help determine a neighbor with a particular property. The neighbor is undefined on an exit from the loop; therefore, some variable (indicating the direction to the neighbor) must be set to determine the neighbor to use (if any) after exiting the loop.

```
exit
```

- **with_neighbor_in_direction:** This instruction is used to employ the attributes of a single specified neighbor. It takes two parameters: a direction variable and an instruction (or block of instructions). It finds the neighbor in the given direction and executes the instruction(s) using the attributes of this neighbor. Frequently, `for_each_neighbor_do` and `with_neighbor_in_direction` are used in conjunction. The `for_each_neighbor_do` may cycle through all the cell's neighbors to determine the direction to a specific neighbor, such as a neighbor with a given tissue type. Then `with_neighbor_in_direction` provides access to the attributes of the specific neighbor.

  `with_neighbor_in_direction` <direction-expression> <instruction>

  The net flow of *BcOne* can be determined, as in the `for_each_neighbor_do` example;

the instruction `with_neighbor_in_direction` can then be used to move in that direction.

```
with_neighbor_in_direction dirBcOne {
  if (neighbor.tissue_type == tissueA) move dirBcOne;
}
```

If the neighbor in the direction of the diffusion has tissue type A, the cell swaps locations with it.

- **divide:** The divide instruction causes the area of a cell to be split up into equal halves. The instruction has an option specifying the direction of the cell division line. The choices are horizontal, vertical, perpendicular to last division, and random (any) dividing lines.[12] In addition to the area division, the values of the variables of the parent (except biochemicals, which are split equally) are copied to the children. One of the two daughter cells then finishes executing the state definition.[13]

```
divide perpendicular
```

This causes the cell to divide `perpendicular` to its last axis of division.

- **grow:** The grow instruction causes the cell to grow in area by the given size in the specified direction. The size can be any expression evaluating to an integer. A negative size reduces the area of the cell (the cell shrinks). The direction can either be specified by a variable (perhaps representing a biochemical gradient, in which case the cell grows preferentially along that direction), or it can be `random_direction`, in which case the direction of cell growth is randomly chosen.

  grow <integer-expression> <direction-expression>

```
grow 5 random_direction
```

The above instruction causes the cell to grow in area by a unit in each of 5 randomly chosen directions.[14]

- **roundup:** The execution of this instruction rounds up the cell by examining the cell boundary and exchanging boundary points to form a more cohesive unit. This instruction *does affect* the shape of the neighboring cells, because rounding up the cell requires modifying the boundaries of the neighbors. Repeated executions of roundup should lead to decreasing perimeter.

```
roundup
```

- **die:** As the name suggests, this results in cell death; no more instructions of this cell are executed.

```
die
```

This concludes our discussion on the instructions in CPL that are directly influenced by the cell biology.

[12] Each of these choices is represented by a reserved word.
[13] The two daughter cells can now be distinguished.
[14] A cell is represented as a collection of discrete points. Growth by 5 units is equivalent to adding 5 lattice points to the cell.

5.3 Meta-instructions

The instructions discussed in this section enable us to control the simulation, aid program readability, and help visualize the results.

- **simulation_size:** Because the cells are modeled as a collection of discrete integral points, the user has to specify the maximum size (area) the simulation may occupy. This declaration must be the first statement in the program.

  `simulation_size` (<integer>, <integer>)

  ```
  simulation_size ( 25, 30)
  ```

  In the above example, the simulation is carried out for x coordinates ranging from 0 to 25, and y coordinates ranging from 0 to 30. The lower left hand corner is always (0,0), and the user specifies the upper right hand corner. Enough simulation space should be provided to ensure that the cells do not grow/move past any boundary. CPL uses fixed boundaries; growth or movement beyond the boundary results in a run time error.

- **time_interval:** This specifies the discrete time step size for the simulation. The right hand sides of the difference equations of biochemical catalysis and diffusion are implicitly multiplied by this quantity. If `time_interval` = 5, the cells execute their program at time 5,10,15..., which makes the simulation run 5 times faster. However, `time_interval` = 0.5 slows down the simulation by half. Only in the assignments to the derivatives of the biochemicals (`deriv`) is the `time_interval` implicit; the other instructions in the program have to use the `time_interval` explicitly. If some variable is monitoring the elapsed time in a cell, it should be incremented by `time_interval`. It is emphasized that the time interval should be a small number; otherwise, the validity of the discrete time simulation is questionable. The default value of the `time_interval` is 1. It may be overridden by a declaration following the `simulation_size` declaration.

  `time_interval` = <real>

- **echo**: This takes a string in double quotes as an argument and prints out this string on standard output when this instruction is executed. '\n' in the string is treated as a newline character.

  `echo` "<string>"

- **write**: This instruction takes an expression as an argument and prints out the expression value followed by a space on standard output.

  ```
  write <expression>
  write state
  write neighbor.state
  write tissue_type
  write neighbor.tissue_type
  ```

  The last four forms of this instruction print out the numerical representation of the corresponding state or tissue type. These are useful for printing out information about cell motion.

- **image**: This takes a biochemical or variable name as an argument and prints for each lattice point the particular biochemical/variable's value in the cell located at that lattice point. This can be used to visualize simulation results in image form.[15] Alternatively, it can be used to output a matrix of numerical representations of the tissue types or the states of various cells.

[15] Two utility X-windows programs, cplvisual and cplmovie, are included in the CPL software to visualize the results of the simulations. All the images, in Section 7, are produced using cplvisual. cplvisual produces snapshots, while cplmovie produces a continuously varying image that may be recorded on video.

`image` <biochemical-name>
`image` <variable-name>
`image state`
`image tissue`

- **save:** This instruction saves the system state in a file (called "STATUS"); the simulation can be restarted from the last saved state.

  `save`

- **constants:** The language permits C style #define's to define constants.

- **comments:** In addition to instructions, the language permits *comments*. It ignores everything on the line after encountering a //.

The meta-instructions aid both in writing readable CPL programs and in producing results that may be easily visualized.

5.4 Variable Declarations

CPL has plain variables, and it has provisions for specifying biochemicals.

5.4.1 Biochemicals

Each biochemical appearing in a CPL program must be declared as such.
`biochemical` <id-list>
<id-list> is a list of identifiers separated by commas.
Biochemicals can be either integer or floating point. Their storage type is declared akin to the storage type declaration for the other variables.

5.4.2 Other Variables

CPL variables have two characteristics: storage type (integer, float, or direction) and scope (static or local).

- **Storage type**: The variables must be declared to be either integer, float, or direction.

  `integer` <id-list>
  `float` <id-list>
  `direction` <id-list>

- **Scope:** Each cell has a different copy of most user-defined variables, including biochemicals, and it retains this copy throughout its lifetime. Such variables are termed `local` variables. This is the default condition of all variables. In addition, `static` variables are made available, whose values are shared by all the cells. Thus, a `static` variable has the same value no matter from which cell it is accessed, as opposed to a local variable which has different values in different cells. `static` variables are useful for collecting statistics about the cell aggregate, such as the count of cells of a specified tissue type, or the total amount of a biochemical present in the entire tissue. In fact, it is difficult to justify using `static` variables for any purpose other than data collection, because in the biological context `static` variables permit non-local communication.

  `static` <id-list>

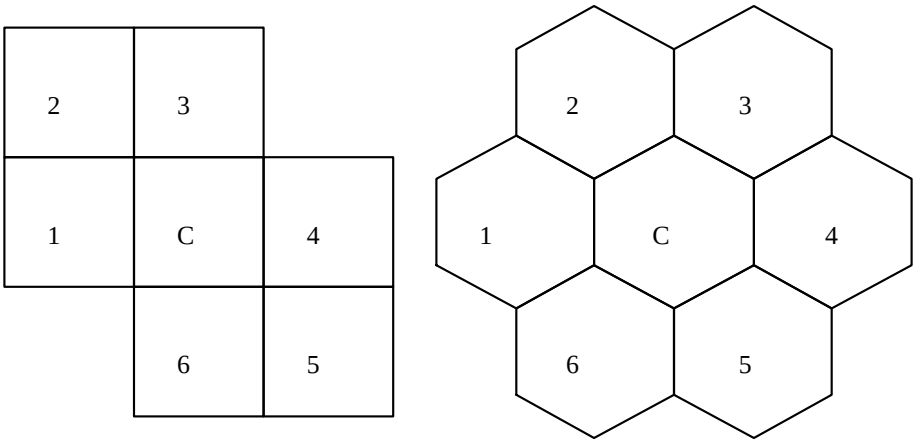


Figure 6: The map transforming the square lattice to a hexagonal lattice.

5.5 Cell Declarations

Before running a simulation, the locations of the initial cells have to be specified. Here is an example of an initial cell definition:

```
cell {
  type generic; // Tissue type of the cell
  start_up_area rectangle(20,20,21,21); // starts up with area 4
}
```

The following instructions may be used inside a cell definition and serve to elucidate the above example:

- **type:** This declares the tissue type of the cell.

  `type` <tissue-name>

- **assignment**: This statement may be used to initialize the biochemical concentrations, as well as other variables. It has the same structure as the assignment previously discussed. Assignments to derivatives of biochemicals are not useful in initialization of cells.

- **start_up_area**: This indicates the starting size and location of the cell. For the purpose of defining starting locations of cells, the user can specify the points on a square lattice. The CPL implementation uses the map given in Figure 6 to convert this square lattice to a hexagonal lattice (because cells contain points on the hexagonal lattice). Six of the eight neighboring point on the square lattice map to neighbors on the hexagonal lattice.

  `start_up_area` <object> [`union` <object>]$^+$

  where <object> is either `rectangle`, `circle`, `triangle`, or `hexagon`. The union operator enables the specification of composite shapes for the cell.

- **rectangle**: This declares a rectangular region of points as part of a cell.

  `rectangle` (<integer>, <integer>, <integer>, <integer>)

  The four integers parameters are (x1, y1, x2, y2), where (x1, y1) are the coordinates of the lower left hand corner, and (x2, y2) are the coordinates of the upper right hand corner.

- **circle**: This declares a disk of points as part of the cell. It takes three integers as parameters (x, y, r), where (x, y) is the center of the circle, and r is its radius.

  `circle` (<integer>, <integer>, <integer>)

  The circle produced is a circle on the square lattice; however, mapping it to the hexagonal lattice distorts it.

- **hexagon**: This declares a hexagon of points as part of the cell. It takes three integers as parameters (x, y, r), where (x, y) is the center of the hexagon, and r is its radius. This does not define a unique hexagon, but a family of hexagons. The hexagon with corners on $(x-r, y+r), (x, y+r), (x+r, y), (x+r, y-r), (x, y-r), (x-r, y)$ is chosen because under the map in Figure 6, it maps onto a regular hexagon.

  `hexagon` (<integer>, <integer>, <integer>)

  The hexagon on a hexagonal lattice has the property that all the points on its perimeter are equidistant from its center. It corresponds to a circle in the continuous domain.

- **triangle**: This declares a triangle of points as part of the cell. It takes six integers as parameters, namely the coordinates of the three corner points $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ of the triangle.

  `triangle` (<integer>, <integer>, <integer>, <integer>, <integer>, <integer>)

- **unit_area** A typical simulation may have thousands of cells. To avoid repeating the initial code for each cell, CPL provides notation for declaring arrays of cells of unit area (i.e. single point cells). If the first statement in a cell definition is the reserved word `unit_area`, then the entire area of the cell is split up into distinct cells, each possessing the same tissue type and unit area. This enables the definition of arbitrary shaped cellular array (with the constraint that each cell has unit area). The following example declares an array of $10 \times 10 = 100$ cells, starting from point (11, 11) to point (20, 20).

```
cell {
  unit_area; // Declares it to be an array of unit area cells
  type generic; // Tissue type of the cell
  start_up_area rectangle(11, 11, 20, 20);
}
```

Cells are initialized with their areas in the order of their definition; if their areas overlap, the cell declared last receives the area in question.

5.6 Accessing Cell Attributes

This section discusses each cell attribute in detail, providing both syntax and semantics.

- tissue type: Although the tissue type is a primary attribute, it is not necessary to use its value in the program for the tissue; as a program is written for each specific tissue type, the tissue type is implicitly coded in each instruction of the program.

- `biochemical`: All the biochemicals declared are assumed to be present in all the cells in the simulation space, though possibly in different concentrations. Any valid identifier can be chosen to represent a biochemical concentration. That identifier may be used as a legal variable in all situations, with the caveat that all assignments to this variable are deferred until the end of the simulation time step. All biochemical concentrations are updated simultaneously and synchronously at the end of each time step. Thus, when cells access biochemical concentrations of their neighbors, they are all consistent and independent of the order of execution of the cell programs. It is recommended that assignments to biochemicals use the special `deriv` assignment to highlight this difference.

- `area`: The cell area may be used in different contexts, such as determining if the cell has grown to a sufficient size for division. The cell area is a non-negative integer. Its value indicates the number of discrete points that form the cell's internal representation. It is possible to ignore the internal meaning of the area and use it as a representation of the relative areas of different cells. `area` is a reserved word and may only be used as a read-only variable. Its value cannot be changed by the assignment statement. There exists a separate instruction called `grow` that can modify this variable.

- `perimeter`: The cell perimeter is the number of discrete points (in the internal representation) on the boundary of the cell. It may be used to determine the fraction of the cell boundary that may be in contact with a particular cell. `perimeter` is a reserved word and may only be used as a read-only variable, whose value is affected by the instructions `grow`, `divide`, and `roundup`. A positive growth would tend to increase the perimeter; negative growth, division, and rounding up tend to decrease the perimeter.

- `location`: The physical location of a cell may be accessed for programming purposes, such as setting up initial conditions. This is guaranteed to be a lattice point belonging to the cell. For cells with unit area, this provides the unique x and y coordinates of the cell's location in direction form (i.e. `point(x,y)`).

- `cell_number`: This provides a unique integer identifying the cell, and images of its value may be used for identifying cell boundaries.

- Neighbor attributes: The physical layout implicitly defines the group of immediate neighbors of each cell. The attributes of a neighbor may be accessed in the cell's tissue program. These attributes are read-only and available only inside the `for_each_neighbor_do` and `with_neighbor_in_direction` instructions.

  - `neighbor.area` is the area of the neighboring cell.
  - `neighbor.contact_length` is the size of the interface with the neighboring cell.
  - `neighbor.perimeter` is the perimeter of the neighboring cell.
  - `neighbor.direction` is the direction to the neighboring cell. This is the direction perpendicular to the boundary between the two cells and is determined by a walk along the boundary with the neighbor. The magnitude of this vector represents the size of the contact.
  - `neighbor.`<biochemicalname> is the biochemical concentration in the neighboring cell of the given biochemical. The biochemical concentration accessed is the concentration in the neighbor at the end of the previous time instance. This may be used to compute the biochemical gradient at a given time, which would help determine the biochemical concentrations at the next time instance.
  - `neighbor.`<variable> accesses the value of the neighbor's variable. Cells may have a variable, for example age, that tracks their lifetime in a certain state, and its neighboring cells may access this information (`neighbor`.age) to synchronize their own life cycle.
  - `neighbor.tissue_type` is the tissue type of the neighboring cell. This may only be tested for equality or inequality with an identifier representing a tissue name (i.e. `neighbor.tissue_type` == Dictyostelium).
  - `neighbor.state` is the current state of the neighboring cell. This may only be tested for equality with an identifier representing a state name (i.e. `neighbor.state` == waitForSignal).

- `time`: It is reasonable to assume that cells have some idea of their lifetime. This variable captures the notion of lifetime, by storing the current running time of the simulation. The time attribute is useful for deciding when to switch cell states or perform other actions. It has a real value that starts from 0 and is automatically incremented by the `time_interval` at the start of each time cycle. This is a read-only variable; cell programs cannot modify its value.

- `time_interval`: This is a user-defined constant, and it sets the pace of the simulation. A description of its significance is provided in Section 5.3.

- `steps`: This is a system variable that provides the number of simulation time steps taken so far. It is equal to the `time` divided by `time_interval`.
- `random`: This provides random numbers, which may be used to take probabilistic actions. The function call $random(x, y)$ provides a random integer between x and y. For example, this may be used to differentiate a tissue into two different types with roughly half the cells of each type. Actual cells do not have random number generators, but some of their processes are stochastic, and this probabilistic nature is captured by providing a random number generator.

```
if (random(0,1) == 0) differentiate_to tissueA;
else differentiate_to tissueB;
```

5.7 Other CPL Features

In this section, we describe some additional CPL features and provide modeling suggestions that enable the writing of CPL programs.

- The first state in the tissue is the default state. Cells belonging to the tissue start by executing the code of the first state in the tissue. If a tissue has just one state, it need not be named. This is accomplished by omitting the state name declaration.

```
tissue simple {
  // state simpleOne { not needed
  <instruction-list>
  }
```

- A tissue *environment* is defined by default. The lattice points in the simulation space that are not defined explicitly as belonging to a tissue, belong to this environment tissue. The "null" program for the tissue type *environment* may not be modified.
- A cell of tissue type *observer* is defined by default; however, a program for the tissue *observer* has to be defined. This special cell always executes its instructions at the end of every time cycle and may be used to collect results (or display images). In addition, the static variables that are collecting data over the entire cell aggregate may be reinitialized after each time step in the code for the *observer*, because the *observer* always executes after all the other cells have executed for the current time value, and before they execute for the next time value.
- Cells of the same tissue type exhibit similar behavior (with minor differences) even in their different states. CPL contains a feature *like* that enables us to specify if a state is similar to another. This is accomplished

by:

```
tissue amoeba {
  state one: like amoebaCore {
      ⋮
  }
  state amoebaCore {
      ⋮
  }
}
```

In effect, this executes the code for the state *amoebaCore* before executing the code for state *One*. Normally, each multi-state tissue should have a core state (like *amoebaCore*) containing all the biochemical interaction and diffusion equations, because these are, for the most part, invariant during a cell's history.

- We do not model intercellular space implicitly; instead, intercellular space may be modeled by specifying "dummy" cells that act as intercellular space. An explicit definition for the tissue type "dummy", specifies the behavior of dummy cells.
- The chemical concentration of the biochemicals in the environment is assumed to be zero at all times. Control over the concentration of biochemicals in the environment can be achieved by surrounding the cell aggregate by some other tissue type and specifying a program for this user-defined tissue type.
- Even though the time step of the simulation can be specified by the user, the discrete simulation, by definition, assumes a "small" step size. In particular, the largest time step for running the simulation is limited by the fact that diffusion in a time step should at best equalize the concentration between neighbors. In a cell, the biochemical concentration should not fluctuate wildly at successive time steps because of diffusion. In that case, diffusion is being modeled incorrectly.

6 CPL Implementation

Our exercise of writing programs for developmental behavior began with writing special purpose programs for various developmental phenomena (notably, cell segregation and engulfment). However, we soon realized that these programs were for the most part similar, and it should be possible to write simpler descriptions (in program form) of this developmental behavior. A small lexical analyzer (using lex) and a parser (using yacc) were written for CPL. The initial CPL programs were converted into quadruples, which were in turn interpreted. The interpreter was written in C++. Eventually, the interpreter was discarded in favor of a CPL to C++ translator to improve execution speed. The current implementation provides a library of functions for CPL instructions. The parser replaces CPL instructions with calls to C++ functions. These are then compiled and linked with the library to produce fast optimized code for the specific application.

6.1 Function Libraries

Two libraries are provided: The first library (called UNIT) is optimized for CPL programs that only use cells with unit area, and the second library (called MULTI) provides functions equipped to handle cells with non-unit area. UNIT provides greater simulation speed and has been extensively tested; MULTI is more general and awaits serious biological applications.

For unit area cells (UNIT), the instructions grow, divide, and roundup are not needed. Each cell has a fixed perimeter, which makes it trivial for cells to identify their neighbors. If a cell moves, it is easy to identify (and tag) what other cells it may have affected, and only those cells need to recompute their neighborhoods.

In the general case (MULTI), a single cell operation (move, grow, divide, or roundup) may affect the geometry of a large number of cells. Thus, each operation may necessitate recomputation of a large number of cell boundaries. Overhead is also involved in ensuring that the cells remain connected, and that their areas are correct, because operations on other cells may change a cell's area.

6.2 Cell Execution Order

The implementation of CPL is sequential. At each time instant the code for all the cells is executed in random order. This eliminates any effects that may be introduced by executing the code for the cells in a specific order.

6.3 Stability of Neighborhoods

Before executing the code at any time instant, a cell recomputes its neighborhood to account for any changes that may have taken place. This is reasonable, if a sizable number of cells move at each time instant. If just a few of them move at each time instant, a more efficient scheme is utilized. In this case, neighborhoods are only recomputed if they may have been modified. Each time a cell moves, it tags all its old and new neighbors, informing them of possible changes in their neighborhood. Only such tagged cells recompute their neighborhood. A compiler directive enables the user to choose between the two alternatives. The default is to recompute the neighbors at each time instant.

The recomputation of the neighborhood is accomplished by selecting a neighbor at random, and then traversing the boundary and forming a list of neighbors along the boundary. Unless the neighborhood is recomputed, the `for_each_neighbor_do` instruction will access the neighbors in order starting from the same neighbor at every time step.

6.4 Choice of Topology

The hexagonal topology is the default. For multiple area cells, this is the only well-defined topology provided by CPL. However, for unit area cells, the user may choose the eight neighbor topology with a compiler directive. All the simulations in this paper use the hexagonal topology.

6.5 Cell Shape and Area Modification

In this section, we discuss the implementation of CPL instructions that may affect the shape or area of some cells. The implementations are difficult only when at least some cells have non-unit area.

6.5.1 Move

A cell can only move to a location occupied by another cell.[16] The move instruction is implemented as a complete exchange of the lattice points occupied by the two cells. The implementation is trivial if both the cells have equal area. If they have unequal area, then the move temporarily changes their area; however, the original area is restored by growing (shrinking) the cells by the requisite amounts. This may modify the shapes of the cells involved, and possibly other cells.[17]

A move, where cells of unequal area are involved, is computationally expensive, because it requires recomputation of neighborhoods for a potentially large number of cells.

[16] The space between cells is modeled as just another cell type.

[17] Other cells are affected, because the grow instruction has a non-local affect.

6.5.2 Divide

The divide instruction should split the lattice points that comprise the cell equally between the two daughter cells. Thus, each daughter cell should end up with half the lattice points. The division should also ensure that the two daughter cells are each connected. A cell is connected if and only if between each pair of lattice points (l_0 and l_n) belonging to the cell:

- There is a path of lattice points ($l_0 l_1 \ldots, l_{n-1} l_n$) belonging to the cell, where $l_1, \ldots, l_{n-1} \in$ cell.
- On the path of lattice points, l_i and l_{i+1} (for all $i = 0, 1, \ldots, n-1$) should be neighbors on the hexagonal grid.

The divide is implemented by finding the two extreme points on the cell in a direction perpendicular to the choice of dividing line. If a horizontal dividing line is desired, the topmost and the bottommost points on the cell serve as the seeds for the daughter cells. Lattice points from the mother cell are added to the daughter cell alternatively. Only lattice points that are neighbors of the lattice points already belonging to the daughter cells may be considered for addition to the daughter cell. This strategy ensures that the two daughter cells are connected, and halts when no more lattice points may be added to one of the daughter cells. The procedure may halt for two reasons. All the lattice points in the mother cell may have been divided (which ends the divide procedure), or one of the daughter cells may have cut off the path (for the other daughter cell) to the remaining lattice points in the mother cell. In the second case, the remaining lattice points in the mother cell are added to the daughter cell that has access (by means of a path) to them. Subsequently, the grow instruction is employed to correct the imbalance in area between the two daughter cells (by reducing the area of one and increasing the area of the other).

6.5.3 Grow

A possible implementation of growth (not employed by CPL) is accomplished by walking from the cell's center of mass in the chosen direction until its boundary is encountered; the boundary point so encountered (belonging to another cell) is stolen. This cell in turn steals a point from its neighbor by performing a walk in the same direction from its center of mass. With this implementation, if a rectangular cell is grown horizontally, the cell does not remain a rectangle; rather, only its center line expands. This does not have the desired effect, because we would like to be able to grow cells so that they maintain their shape. This implementation of grow is an extension of the one proposed by Ransom [27].

An alternative implementation involves computing the boundary and randomly picking the requisite number of points on the boundary. Each chosen point should be such that its neighboring point in the direction of growth belongs to a different cell. The cell then steals the neighboring point. The neighboring cells then steal from their neighbors in turn until the effect ripples out of the cell aggregate. With this implementation, if a rectangular cell is grown horizontally, all the points on the vertical edges have an equal chance of being chosen, resulting in a rectangular cell. This is the currently available implementation of the grow instruction in CPL .

This implementation is not perfect either, because the outward ripple alters the shape of the exterior cells.

6.5.4 Roundup

The implementation of `roundup` is based on an extension of a algorithm proposed by Goel and Rogers [10]. Their procedure for non-local exchange of cells was designed to cause segregation and engulfment of tissues. We have modified it to cause cells to round up. This is not surprising, because the same adhesive force is responsible for cells rounding up, tissue segregation, and tissue engulfment.

Description of the rounding up procedure:

1. Consider the boundary lattice points that are shaded dark in Figure 7. These points have a varying number (P) of adjoining points belonging to the cell. Because we use a hexagonal topology, P varies between 1 and

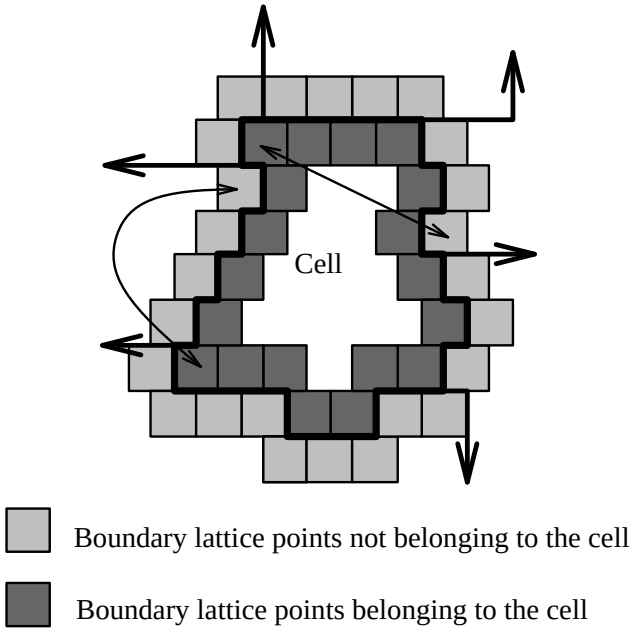


Figure 7: Cell rounding up. The thick line is the cell boundary. The cell is comprised of the dark shaded cells on the boundary and the light interior. The thin double-edged lines indicate possible lattice point exchanges that will round up the cell.

5.[18] Order these points in increasing order of P. Observe that points with low P are responsible for jagged cells, and removing them from the cell would round up the cell. However, this would reduce the area of the cell.

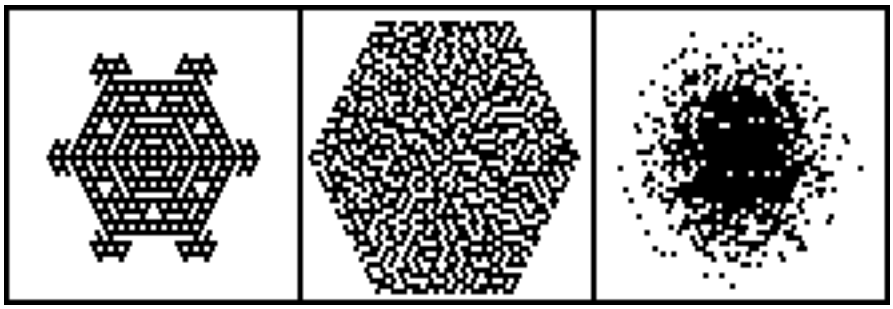
2. Now consider the boundary lattice points that are shaded light in Figure 7. Consider their P number (the number of adjoining lattice points belonging to the cell to be rounded up), which again varies between 1 and 5. Order these points in decreasing order of P. Observe that points with high P are responsible for incisions into the cell under consideration, and adding them to the cell would round up the cell. However, this would increase the area of the cell and reduce the area of a neighboring cell.

We combine the previous two steps to balance the additions and deletions of lattice points to the cell. This rounds up the cell under consideration. The lattice points deleted from the cell are added to a neighboring cell. We keep track of additions and deletions to neighboring cells, and if they do not cancel out, we invoke the grow procedure for the cell to correct its area. This grow invocation can undo the rounding up, and repeated invocation of round up may be required to achieve cohesive cells. In practice, the rounding up performs well.

7 Applications

In this section, we examine through models and simulations various developmental phenomena. Some of these models are reasonably realistic, especially slime mold aggregation, cellular segregation, and precartilage formation. The biological details about some of these simulations are presented in Section 3.

[18] For non-unit area cells, points on the boundary cannot have 6 neighboring points belonging to the same cell.



(a) (b) (c)

Figure 8: Cellular automaton simulations: (a) fractal, (b) dendrite formation by controlled growth, (c) diffusing gas.

7.1 Cellular Automata

Figure 8 displays traditional cellular automaton simulations. Similar figures may be seen in Toffoli's book on Cellular Automata [34]. Figure 8(a) exhibits an image with a fractal flavor. Figure 8b exhibits dendrite formation by allowing only cells that have exactly one neighboring cell of dendritic form to differentiate to form dendrites. Figure 8c exhibits a diffusing gas. Each cell is a gas molecule, and random motion causes it to diffuse. Although CPL has a distinct developmental biology flavor, it can be employed for other cellular automaton applications. However, the versatility of CPL programs is at the expense of speed as compared to cellular automaton programs: *cellular* [5] and *SLANG* [28]. An extensive review of cellular automaton approaches to biological modeling is provided by Ermentrout and Edelstein-Keshet [6].

7.2 Unit Area Cells

Even though CPL can be used for conventional cellular automaton models, it is most useful for biological models. Each of the following subsections displays a simulation with the patterns being formed mainly by the controlled motion of cells, each cell having unit area.

7.2.1 Stripe Formation

Utilizing a sorting mechanism, similar to a differential adhesion mechanism but restricting cell motion to be parallel to one axis, one can obtain striped patterns (Figure 9). The darker cells have stronger adhesivity to each other, and when they come in contact with each other, they tend to stay in contact. If the cells were mobile both horizontally and vertically, we would see small clumps. Restricting the motion to the horizontal axis provides vertical stripes, which are frequently seen developmental patterns (for example, in animal coats), and this example illustrates a technique for producing them. Alternatively, they may develop by the interaction of diffusing biochemicals (for example, Turing's stationary wave patterns) [17, 36].

7.2.2 Cell Migration

The ability of cells to follow biochemical gradients is useful in pattern formation. For example, the *Dictyostelium* cells exhibit an extreme sensitivity to cAMP gradients and are able to move in the direction of higher cAMP (Section 3.1). A simple example is provided in Figure 10, which displays the trail left by a single cell following a biochemical gradient. There is a single source of the biochemical at the top-center of the square cell aggregate. The biochemical diffuses throughout the aggregate, and the peripheral cells act as a sink. This establishes a biochemical gradient. The motile cell is initially present at the bottom-center of the square, and wanders randomly but does not stray too far. On detecting the biochemical gradient, it moves towards and finally reaches the biochemical source.

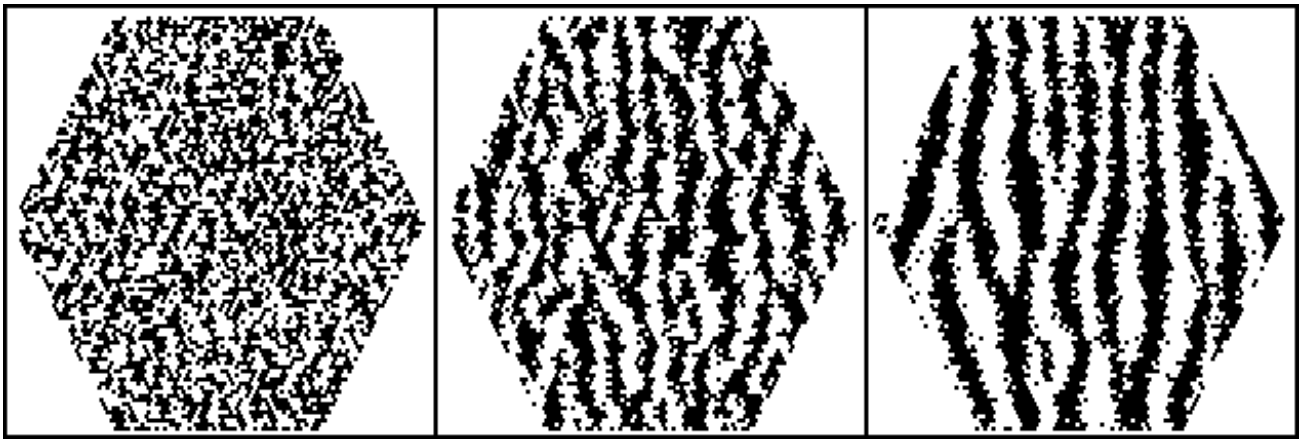


Figure 9: Striped pattern formation using lateral sorting.

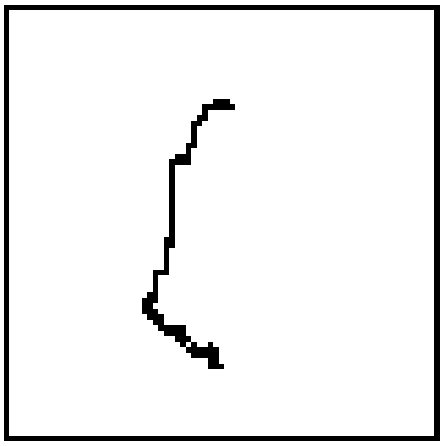


Figure 10: Cell migration along a chemical gradient in a square aggregate of cells. The initial position of the cell is in the bottom-center of the square, and its initial motion is mostly random. The biochemical source is in the top center of the square and as the wandering cell approaches the source, its motion is primarily directed towards the source.

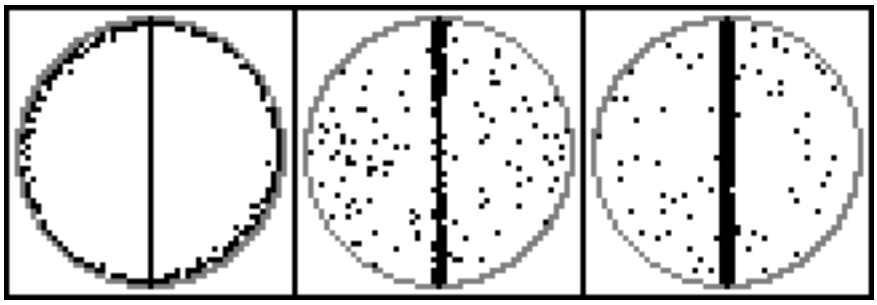


Figure 11: Random cell motion over underlying tissue. This assumes an underlying adhesive tissue along the central spine. The outer cells move randomly and adhere on contact to the spine. (a) Before the cells become mobile, (b) some of the mobile cells have already found the central spine, (c) the spine is covered by the mobile cells.

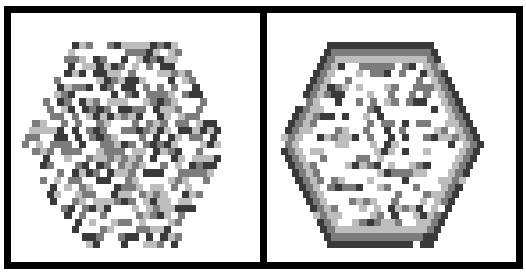


Figure 12: Three layers of tissues being formed using timing hypothesis of segregation.

There are numerous examples in developmental biology of cells wandering, apparently at random, until they happen to arrive at their destination, whence they loose their motility (for example, the migration of mesenchyme cells in the blastula). Figure 11 depicts this phenomena with a central spine acting as the destination. One could envision the central spine to be an underlying tissue to which the wandering motile cells adhere strongly. In part (a) of the figure, the motile cells have not yet gained motility, and are stuck to the periphery. On gaining motility, by pure random wandering they are able to find the spine and adhere to it.

7.2.3 Cell Segregation and Engulfment

This example demonstrates the formation of bands of tissues. Figure 12 exhibits an image of three layers of tissue. Initially, cells of all three kinds are randomly intermixed in the aggregate. We assume that the cells have a tendency to adhere to the outer layer, and that different tissue types regain their adhesivities at different times. The tissue type that regained adhesivity first would form the outermost layer, and the tissue regaining adhesivity last would form the innermost layer. A similar explanation called the *timing hypothesis of segregation* has been utilized to explain cellular sorting.

The more widely accepted model explaining segregation and engulfment is the *differential adhesion model*. A brief explanation of this model and its expected behavior is provided in Section 3.3, and the details are available elsewhere [3, 1, 19]. In these simulations, at every time step, each cell computed the neighboring cell that was energetically most favored for an exchange of locations. With probability 1/2, it made this energetically favored exchange; in addition, with probability 1/4, it made a random exchange with a neighboring cell. This simple rule is enough to cause both cell segregation and engulfment. Whether, we observe segregation or engulfment is determined by the initial shapes of the tissues and their relative adhesivities. Figure 13 depicts the complete segregation of three tissues employing only adhesivity differences and some random motion. The

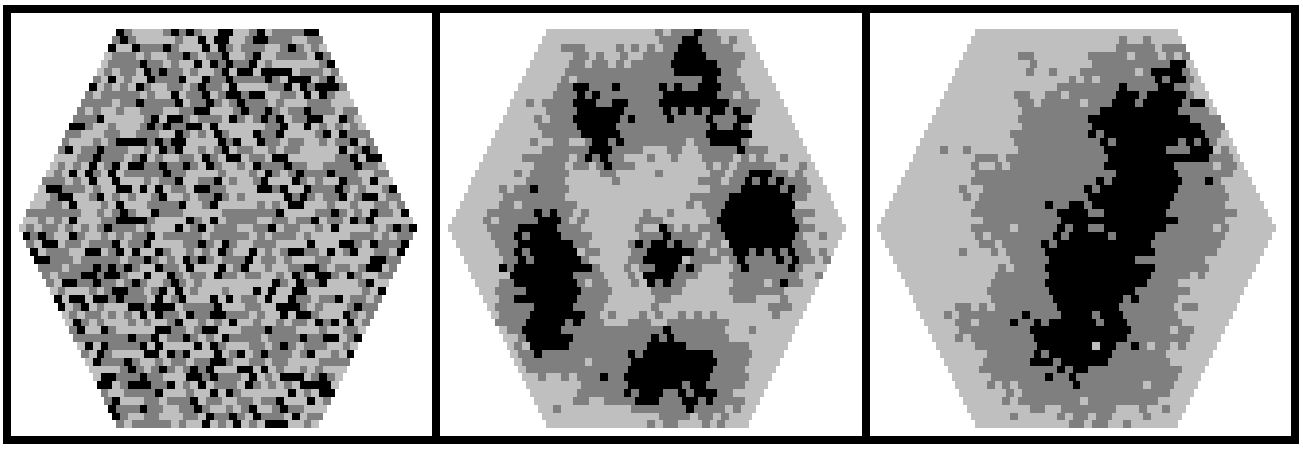


Figure 13: Segregation of tissues using adhesivity differences. The darkest points belong to the tissue with the highest adhesivity (they aggregate centrally), and the lightest ones have the weakest adhesivity (they aggregate externally). The subframes are after 1, 2000, and 30,000 simulation time steps. Excerpted from [3].

most adhesive tissue segregates internally, and the least adhesive tissue segregates externally, as expected from biological experiments. The same principle also explains the engulfment of tissues in Figure 14. If two tissues with different adhesivities are placed in contact, cells from the tissue with lower adhesivity envelop the higher adhesivity tissue. This maximizes the adhesive energy (and minimizes the free energy). Earlier results by Goel et al. [10, 19] indicated that complete segregation and engulfment were difficult to achieve using only local cell exchange rules, but our simulations demonstrate that local rules are enough. The critical difference between the two models is random motion. The movement of a cell to an energetically favorable neighboring position half the time and a random neighboring position a quarter of the time helps the system find a global energy minima. This is not surprising, but curiously had not been attempted earlier.

7.2.4 Cellular Slime Mold Aggregation

The aggregation in cellular slime molds is well-researched (discussed in Section 3.1), and a wealth of experimental data is available [35, 38]. Various models and simulations for this aggregation have been presented [16, 13]. We have also modeled this aggregation using CPL and observed both stream formation and spiraling. The model is reasonably simple, and yet incorporates most of the available quantitative data. In the appendix, we have included a program for *Dictyostelium* aggregation. The cells cycle between three states waiting for a chemotactic signal of cAMP, relaying the signal, and moving towards the source of the signal. The quantitative data, in terms of the chemical signal strength, various time delays, and the random and directed speed of cell motion are included in the CPL program. Random motion was an important component of the aggregation behavior, just as in the segregation and engulfment simulations. The complete details of our simulations are available [2]. Figures 15 and 16 exhibit simulations of aggregation, streaming, and spiral formation in *Dictyostelium* using CPL programs.

7.2.5 Limb Skeleton Formation

The organization of the bones in the limbs is regarded as a classical example of development. The developing limb bud is composed of mesenchymal cells encased by epithelial cells. Some of these mesenchymal cells form condensations (or clusters), and they differentiate to form chondrocytes. Chondrocytes are the precursors of cartilage. Bone eventually replaces cartilage by the process of ossification.

A reaction-diffusion model (Turing model, discussed in Section 2.1) has been proposed by Newman et al. to account for the condensations [22, 21, 14]. The formation of condensations is mediated by the protein

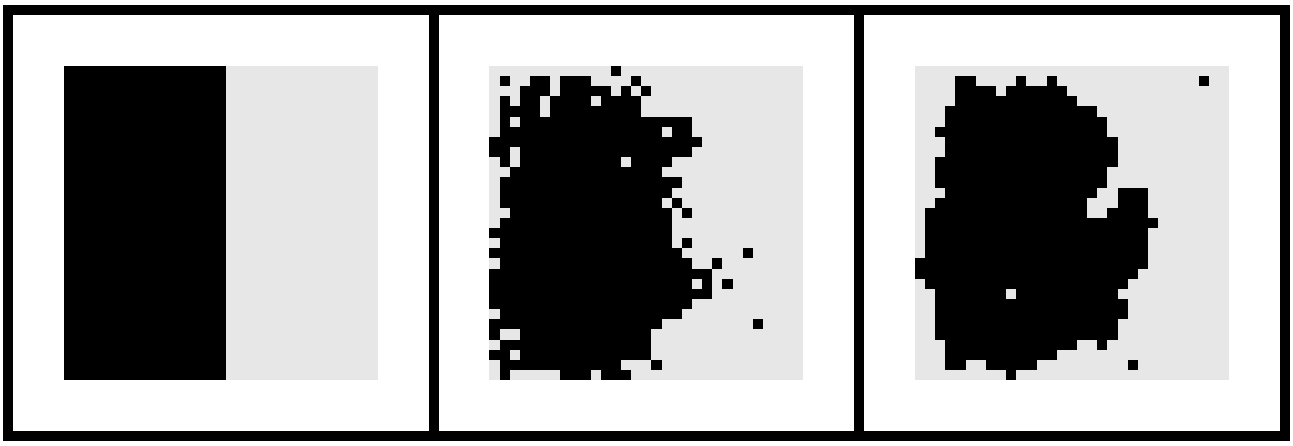


Figure 14: Engulfment of tissues placed in contact using adhesivity differences. The dark squares represent cells (496 in number) belonging to the tissue with the higher adhesivity, and the lighter ones (465 in number) have the weaker adhesivity. The subframes are at 0, 1000, and 2200 time steps. In the last subframe (at 2200 time steps) the tissue is rounded up, because random motion is discontinued after time step 2000. Excerpted from [3].

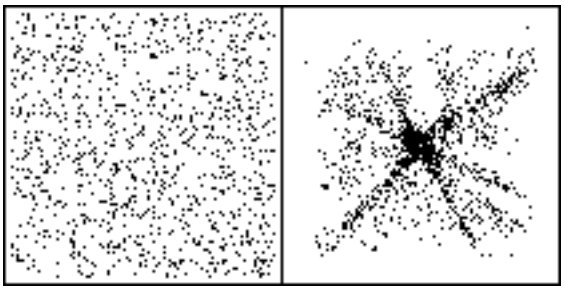


Figure 15: Aggregation in *Dictyostelium*. Each point represents a *Dictyostelium* amoeba. 10% of the lattice points are occupied by amoebae, which translates to a density of $10^5\ cm^{-2}$. The second subframe is after 4,000 time steps of the simulation. A central cell acts as the pacemaker sending out a periodic signal. The other amoebae are streaming towards the pacemaker. Reprinted from [2].

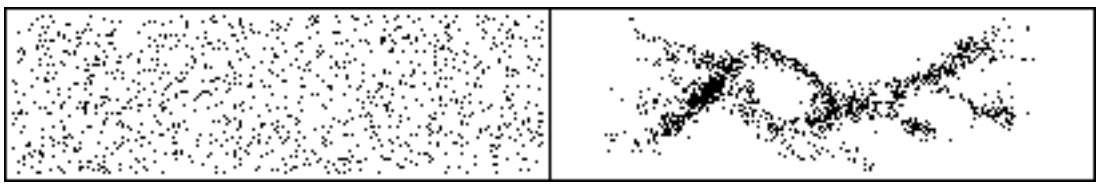
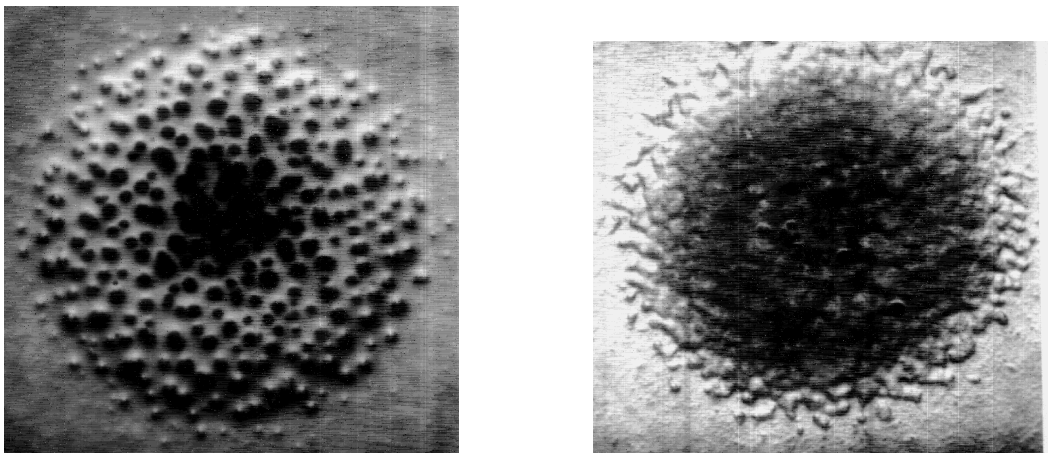


Figure 16: The amoebae in the top half of the first subimage do not react to the cAMP until a later time. This sets up a rotating cAMP wave throughout the aggregate, causing the amoebae to spiral (clockwise) around a spontaneously created empty-center. The second frame is the state after 10,000 seconds of the simulation clock; a central vortex has formed with cells spiraling clockwise around it. There is no autonomous pacemaker; the self-sustaining wave is set up due to the creation of an empty center. Reprinted from [2].



(a) Leg (b) Wing

Figure 17: The precartilage condensation patterns in leg and wing mesenchyme. Alcin blue-stained six day cultures are shown. Both tissues were isolated from stage 24 (4 1/2 day) chicken embryos. Culture diameter $\approx$ 5mm. Pictures courtesy of Downie and Newman [4].

fibronectin, which is present at the site of these condensations. Interference with fibronectin activity disrupts the condensations. The transforming growth factor-β (TGF-β) stimulates the production of both fibronectin and itself, and it is non-uniformly distributed in the precartilage tissue. There is evidence that TGF-β could be the primary morphogen that forms the standing waves in the reaction-diffusion model.

The formation of the vertebrate limb involves the formation of an increasing number of skeletal elements (cartilage/bone) from the base of the limb bud towards its tip (i.e. along the proximo-distal axis)[19]. Wilby and Ede (1975) proposed a model that, given the proximo-distal pattern, simulates the anterior-posterior positioning of skeletal elements [39]. They were able to produce patterning resembling the layout of the bones in the limb (humerus; radius and ulna; metacarpals; digits). These patterns were produced by computer simulations using local cell interactions, with one computer cell equivalent to a hundred real cells. A single morphogen system was used, and a wave of this morphogen was set up in the system using simple rules, including a boundary layer of cells that actively destroys this morphogen. Newman and Frisch have also mathematically analyzed a single morphogen model that accounts for the appearance of the precartilage elements [22].

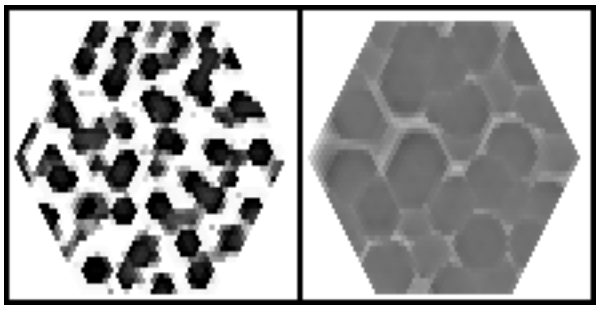
In Vitro

Newman et al. have studied the formation of precartilage (chondrogenic) condensations in chick limb bud mesenchyme *in vitro* [8, 14]. The condensation patterns in the wing and leg mesenchyme are different [4]. As can be observed from Figure 17, the condensations in the leg are focused, while the wing condensations are unable to stay focused and merge to form larger but weaker condensations.

We have proposed a linear activator-inhibitor model that accounts for these condensations. This model is only one *possible* explanation. However, the results of this model are encouraging in that the same model yields both the *in vivo* and *in vitro* precartilage patterns in vertebrate limbs. The model was tested using CPL.

The simulation result for the fibronectin concentrations in the leg and wing tissue are shown in Figure 18. Initially the activator and inhibitor concentrations in the cells are at equilibrium. There is no diffusion across the aggregate boundary; i.e. it is a closed system. Minor random positive perturbations in the activator concentrations of randomly chosen cells, over the course of the simulation time, result in variations in the activator-inhibitor values in the cells. Each perturbation results in unbounded maxima and minima. However, the simulation results

[19] This is referred to as the *in vivo* condensation pattern.



(a) Leg (b) Wing

Figure 18: The *fibronectin* concentrations for the *in vitro* condensation pattern in the chick *leg* and *wing*.

improve if the concentrations are bounded; this is also more realistic, because concentrations cannot have infinite range, especially in biological systems. The only difference in the simulation parameters is the response rate of the inhibitor (*Iresponse*) to changes in activator concentration. The inhibitor response is more pronounced in the wing simulation. The model is essentially a two-dimensional analog of the Turing model presented in Section 2.1.

In vivo

If activator values are perturbed in randomly chosen cells, the condensation patterns resemble the *in vitro* patterns, as previously shown. If instead of the random perturbations, we modify the boundary to permit a slight leakage of the activator we see the *in vivo* condensation patterns, i.e. the pattern of the skeletal elements in the limb. The leakage could be explained by the ectoderm (epithelial) cells surrounding the tissue having slightly lower activator levels.

This leakage results in banded patterns of activator concentration. The number of bands depends upon the width (anterior-posterior dimension) of the cellular aggregate, and the response strength of the inhibitor (*Iresponse*). For a 50 cell wide aggregate $Iresponse = 0.01, 0.08, 0.5$ yields 1, 2, and 5 bands of precartilage respectively, which corresponds to the humerus, radius-ulna, and metacarpals, Figure 19. If the width of the cellular aggregate is held constant, then increasing *Iresponse* increases the number of waves, and the number of precartilage elements[20]. For constant *Iresponse*, increasing the width of the cellular aggregate increases the number of waves. Thus, the cellular aggregate widths and *Iresponse* can be selected to produce the desired number of precartilage bands. The *talpid* mutant of the chick has a wider limb bud and not surprisingly a larger number of skeletal elements [22, 39], which is predicted by our model.

The proximo-distal dimension has *no* affect on the number of skeletal elements formed. In Figure 19, the proximo-distal dimensions (length of the cellular aggregate) were arbitrarily selected to be 70, 50, and 30 cells (corresponding to the humerus, radius-ulna, and metacarpals). These 3 aggregates are isolated from each other (no diffusion across boundaries). This is biologically justifiable, because the precartilage forms along the proximo-distal axis at different times: when the skeletal elements for the radius-ulna are being formed, those for the humerus are already developed, while the metacarpals etc. have not yet been initiated.

7.3 Multiple Area Cells

The previous models all involved cells of the same size and shape. In this section, we will develop some simulations that involve growing and dividing cells.

[20] If *Iresponse* is modeling the catalytic effect of a biochemical substance, then the appearance of each successive precartilage element along the proximo-distal axis is marked by an increase in the concentration of *Iresponse*.

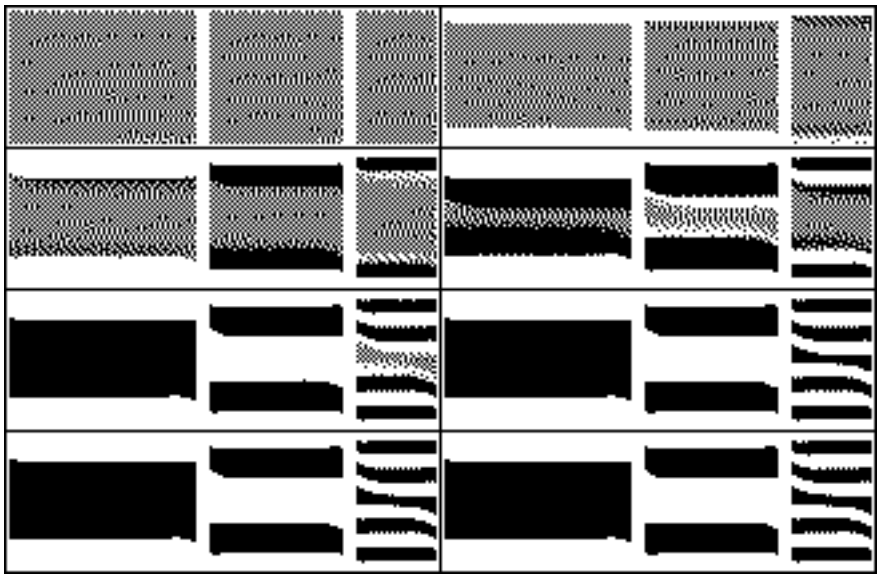


Figure 19: The activator concentrations for the *in vivo* precartilage pattern in vertebrate limb. The subframes are at every 50 time steps starting from 0. The proximo-distal axis is from left to right in each subimage. The anterior-posterior axis is 50 cells high in each subimage. The humerus, radius-ulna, and the metacarpals/digits appear as bands of 1,2, and 5 precartilage elements in aggregates with length 70, 50, and 30 cells along the proximo-distal axis respectively.

7.3.1 Cell Growth

Figure 20 contains images of cell growth. Originally the cell is small (area = 37). The cell grows by stealing points from its neighbors, which in this case is the environment. The second image reveals a rather jagged cell. Because, in nature, cell growth does not lead to such angular features, it is evident that a rounding up of the cell is needed to ensure cohesiveness. Figure 20c exhibits the effect of rounding up this jagged cell. Rounding up is effective in removing the jagged edges, but the cell still has a jagged skeleton. As Figure 20d shows, the best results are obtained by rounding up the cell at each stage of the growth process. This cell was grown by adding 50 points at each time step in randomly chosen directions.

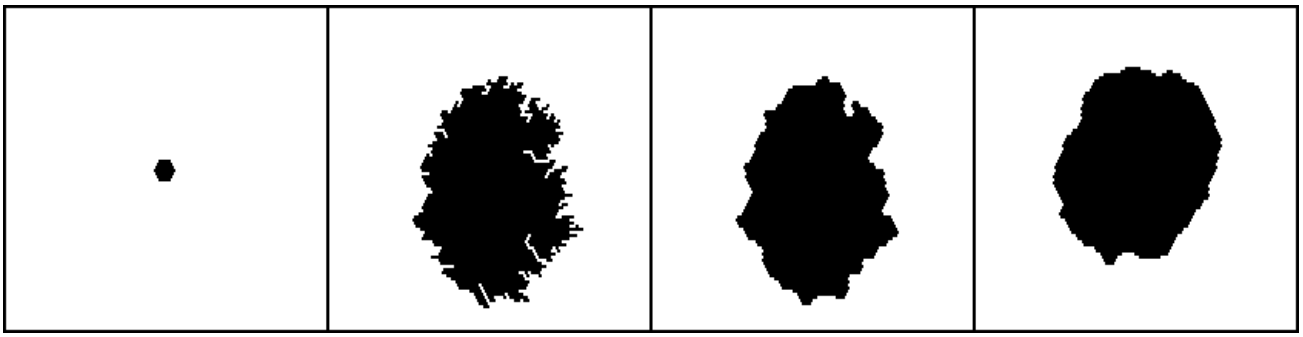
7.3.2 Cell Division

Figure 21 demonstrates the effect of repeated cell division on a reasonably large cell. Successive divisions are made at right angles to the previous one. Cell division lines have to be chosen with care, so as to divide the cell into daughter cells of equal area, while maintaining the connectivity of cells. The choice of hexagonal topology and the division algorithm is evident in the shape of the cell boundaries in Figure 21.[21]

7.3.3 Cell Growth and Division Example

Figure 22 combines the cell growth and division of the previous two examples. The simulation commences with a cell with an area of 37 points (as in the growth example). The cell increases in area by 50 points at every time step, and rounds up after every growing step. The cells divide if their area surpasses 400 points. Successive divisions are perpendicular to the previous one. The rounding up is seen to have a positive effect on cell shape in Figure 22c.

[21] Refer to the implementation note on the division algorithm in Section 6.5.2.



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

Figure 20: Cell growth from 37 points to 2537 points by adding points in random directions. (a) Initial 37 point cell. (b) After random growth by 2500 points. (c) After rounding up the boundary from previous figure (d) After rounding up the boundary at each stage of the growth process (50 stages).

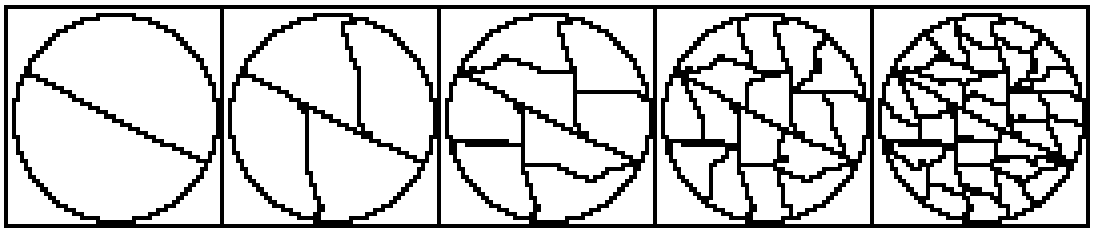
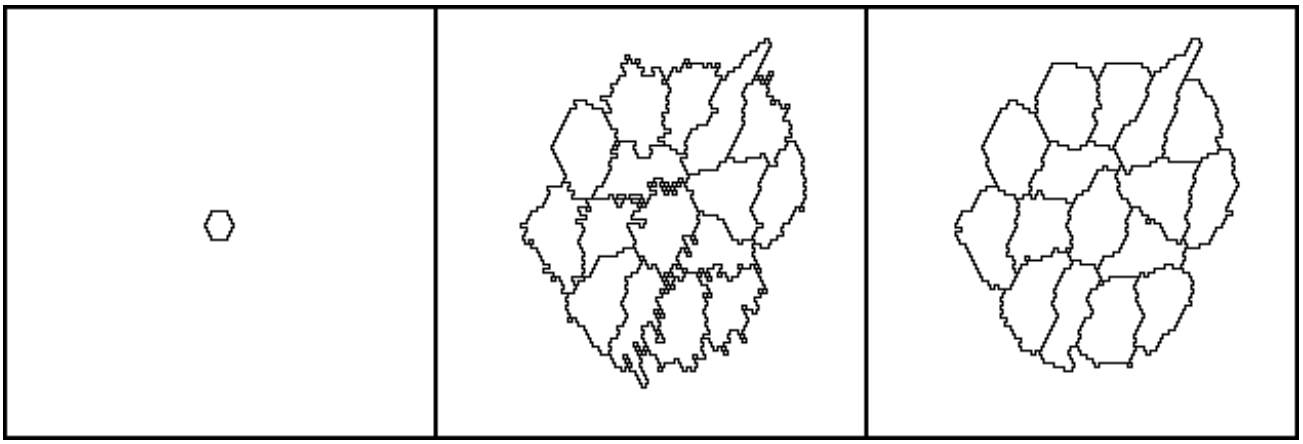


Figure 21: Repeated cell division.



(a) (b) (c)

Figure 22: Tissue grows by about 3500 points. The cells divide upon reaching a size of 400 points. (a) Initial 37 point cell (b) After growth and division (c) After a few stages of rounding up the cells.

These growth and division examples reveal interesting patterns but lack serious biological significance. Growth and division are intrinsically three-dimensional activities. It would be interesting to explore some simple examples of development, where two-dimensional growth and/or cell-division play significant roles in pattern formation. However, we have been unable to locate such biological examples.

8 Conclusions

The aim of this work was to develop a simple language (a means of description) for a variety of developmental behavior. This language took the shape of a programming language (CPL). The various models we developed emphasized the scope of CPL. The modeling of *Dictyostelium* aggregation involved chemotactic movement due to biochemical diffusion [2]. The formation of skeletal elements in limb has been explained by a Turing reaction-diffusion model [1]. The segregation and engulfment of tissues involved minimization of free energy and dealt with mechanical forces in the form of contact, adhesion, and motion [3].

The examination of the instruction sequences for the applications we have developed has proved useful.

- The CPL programs serve as succinct descriptions of the cell's activities (sample program in appendix).
- Comparison between the instruction sequences of distinct cells has highlighted both the similarity and the differences between them.

Running simulations on aggregates of these cells has yielded developmental behavior. We have been able to encode various hypotheses regarding developmental behavior in terms of these instruction sequences, and the simulation results have reinforced or weakened the hypothesis. Some simulations have suggested possible experiments to confirm hypotheses. For example, from the cell segregation simulations an experiment on the time required to double the clump size is suggested, because in the simulations the time required to double clump size increases rapidly (perhaps exponentially). The activator-inhibitor model for precartilage formation in vertebrate limb is being refined to accommodate for new experimental data, and the simulations, in turn have suggested new laboratory experiments.

CPL is similar in some respects to Fleischer and Barr's work on designing a system to study multicellular pattern formation [7]. Both CPL and Fleischer's work are concerned with observing pattern formation in multicellular simulations. However, the goals are different: Fleischer and Barr want to create artificial genomes to simulate evolution, while CPL's intention is to shed light on natural processes by making it easy to model different possible explanations. Fleischer and Barr employ a continuous model with differential equations, whereas CPL utilizes a discrete model with difference equations. CPL can model a much larger number of cells, but the difference equations involved make our calculations more approximate than the continuous model of Fleischer and Barr.

8.1 Computational Power

CPL takes an approach akin to the cellular automata approach; however, the cells are infused with more power. In cellular automata, cells compute simple next-state functions based on their own and their neighbors current states. Cellular automata arrays can represent logical circuits, and they are Turing computable (because circuits are Turing computable). In CPL, each cell is Turing computable, and we believe this adds to its descriptive power. Run-time computational efficiency is a major concern for most cellular automata. CPL is designed with a specific objective — modeling in developmental biology — and can afford to be more descriptive. In CPL, we write programs for mobile-cells, not for fixed points on the lattice.

8.2 Open Problems

Major problems in the realm of growth, division, and motion of variable-sized cells are still unresolved. The solutions CPL adopts for these problems are unsatisfactory. CPL has drawn inspiration from biology for the design of its instruction set, but simple biological models of growth, division, and motion are not available.

Motion can be managed if all the cells are similar in shape and size, but in other cases our solution is somewhat ad hoc.

Most development is essentially three-dimensional. It is evident that the two-dimensional models are in most cases approximations to reality. In the *Dictyostelium*, the conical mound that forms is three-dimensional and removes cells from the two-dimensional aggregation plane. This provides extra space for the *Dictyostelium* streams to converge. Cellular segregation is a three-dimensional process, and the thermodynamic validity of the two-dimensional model has been questioned. The extension of CPL to three-dimensions is straightforward (essentially the neighborhood function has to be modified). However, the extra computational memory and time such simulations require raise questions about the choice of discrete representation of cells. Spheres and/or ellipsoids of varying radii, or polyhedrons with a limited number of surfaces, are possibilities for cell representations in three dimensions.

The current implementation of CPL is a sequential one, but cells inherently work in parallel. The behavior of cells would be further illuminated if we adopted a parallel implementation that mimics their natural behavior. Cells only have local information, and they utilize it to decide their next state. Somehow, by utilizing local schemes, the entire cell aggregate is able to maintain energetically favorable shapes. But, if a cell moves or grows, its effect is felt by its immediate neighbors, and through them by the rest of the aggregate. We need better understanding of exactly how local behavior determines global shape and size. That knowledge would enable us to assign processes to each cell which work semi-synchronously, and would make a good parallel implementation.

Simulation of biological activity will play an increasingly important role in its comprehension. We hope that in the near future it will be possible to build detailed cellular- and possibly even molecular-level models of the simplest living systems.

Software

A prototype of CPL is available on the World Wide Web (http://ibc.wustl.edu/~agarwal/cpl) and by anonymous ftp from ibc.wustl.edu, file /pub/agarwal/cpl/cpl.tar.Z.

Acknowledgments

I would like to thank Jacob T. Schwartz for the introduction into the world of cells; Jerome K. Percus, Stuart A. Newman, and Ron K. Cytron for their patient listening and comments; Laureen C. Treacy for her dedicated proof-reading; and the anonymous reviewers for their excellent suggestions.

References

[1] P. Agarwal. *Cell-based Computer Models in Developmental Biology.* PhD thesis, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012, September 1993. Available by anonymous ftp from cs.nyu.edu, file /pub/theses/agarwal.ps.Z.

[2] P. Agarwal. Simulation of aggregation in *Dictysotelium* using the Cell Programming Language. *Comput. Appl. Biosci.*, 10(6):647–655, December 1994.

[3] P. Agarwal. Simulation of cellular segregation using the Cell Programming Language. *J. Theor. Biol.*, 176(1):79–89, September 1995.

[4] S.A. Downie and S.A. Newman. Morphogenetic differences between fore and hind limb precartilage Mesenchyme: Relation to mechanisms of skeletal pattern formation. *Developmental Biology*, 162(1):195–208, 1994.

[5] J.D. Eckart. A *cellular* automata simulation system. *SIGPLAN Notices*, 26(8):80–85, 1991.

[6] G.B. Ermentrout and L. Edelstein-Keshet. Cellular automata approaches to biological modeling. *J. Theor. Biol.*, 160:97–133, 1993.

[7] K.W. Fleischer and A.H. Barr. A simulation testbed for the study of multicellular development: The multiple mechanisms of morphogenesis. In C. Langton, editor, *Artificial Life III*, pages 389–416. Addison-Wesley, Redwood City, CA, 1993.

[8] D.A. Frenz, N.S. Jaikaria, and S.A. Newman. The mechanism of precartilage Mesenchymal condensation: A major role for interaction of the cell surface with the Amino-terminal Heparin-binding domain of Fibronectin. *Dev. Biol.*, 136:97–103, 1989.

[9] S.F. Gilbert. *Developmental Biology.* Sinauer Associates, Sunderland, MA, 1991.

[10] N.S. Goel and G. Rogers. Computer simulation of engulfment and other movements of embryonic tissues. *J. Theor. Biol.*, 71:103–140, 1978.

[11] R. Gordon. On stochastic growth and form. *Proc. Natl. Acad. Sci. USA*, 56:1497–1504, 1966.

[12] B.K.P. Horn. *Robot vision.* The MIT press, Cambridge, Massachusetts, 1986.

[13] D.A. Kessler and H. Levine. Pattern formation in *Dictyostelium* via the dynamics of cooperative biological entities. *Phys. Rev. E*, 48:4801–4804, December 1993.

[14] C.M. Leonard, H.M. Fuld, D.A. Frenz, S.A. Downie, J.Massagué, and S.A. Newman. Role of Transforming Growth Factor-β in chondrogenic pattern formation in the embryonic limb: Stimulation of Mesenchymal condensation and Fibronectin gene expression by exogenenous TGF-β and evidence for endogenous TGF-β-like activity. *Dev. Biol.*, 145:99–109, 1991.

[15] A. Lindenmayer. Mathematical models for cellular interaction in development, I & II. *J. Theor. Biol.*, 18:280–315, 1968.

[16] S.A. MacKay. Computer simulation of aggregation in *Dictyostelium Discoideum*. *J. Cell Sci.*, 33:1–16, 1978.

[17] H. Meinhardt. *Models of Biological Pattern Formation.* Academic Press, New York, 1982.

[18] E. Mjolsness, D.H. Sharp, and J. Reinitz. A connectionist model of development. *J. Theor. Biol.*, 152:429–453, 1991.

[19] G.D. Mostow, editor. *Mathematical Models for Cell Rearrangement.* Yale University Press, New Haven, 1975.

[20] J.D. Murray. *Mathematical Biology.* Springer-Verlag, New York, 1989.

[21] S.A. Newman. Lineage and pattern in the developing vertebrate limb. *Trends in Genetics*, 4:329–332, 1988.

[22] S.A. Newman and H. L. Frisch. Dynamics of skeletal pattern formation in developing chick limb. *Science*, 205:662–668, 1979.

[23] G. Odell, G. Oster, P. Alberch, and B. Burnside. The mechanical basis of morphogenesis. I. Epithelial folding and invagination. *Dev. Biol.*, 85:446–462, 1981.

[24] Q. Ouyang and H.L. Swinney. Transition from a uniform state to hexagonal and striped Turing patterns. *Nature*, 352:610–612, 1991.

[25] P. Prusinkiewicz. Visual models of morphogenesis. *Artificial Life*, 1(1/2):61–74, 1994.

[26] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer Verlag, New York, 1990.

[27] R. Ransom. *Computers and Embryos: Models in Developmental Biology.* John Wiley, New York, 1981.

[28] H.B. Sieburg and O.K. Clay. The cellular device machine development system for modeling biology on the computer. *Complex Systems*, 5:575–601, 1991.

[29] M.S. Steinberg. Reconstruction of tissues by dissociated cells. *Science*, 141:401–408, 1963.

[30] M.S. Steinberg. Does differential adhesion govern self-assembly processes in histogenesis? Equilibrium configurations and the emergence of a hierarchy among populations of embryonic cells. *J. Exp. Zool.*, 173:395–434, 1970.

[31] M.S. Steinberg. Cell-cell recognition in multicellular assembly: Levels of specificity. In *Cell-Cell Recognition*, pages 25–49. Cambridge University Press, Cambridge, 1978.

[32] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[33] D.W. Thompson. *On Growth and Form.* Cambridge University Press, Cambridge, 1917.

[34] T. Toffoli and N. Margolus. *Cellular Automata Machines : A New Environment for Modeling.* MIT Press, 1987.

[35] K.J. Tomchik and P.N. Devreotes. Adenosine 3', 5'- Monophosphate waves in *Dictyostelium Discoideum*: A demonstration by isotope dilution-fluorography. *Science*, 212:443–446, 1981.

[36] A. Turing. The chemical basis of morphogenesis. *Phil. Trans. Roy. Soc. B*, 237:37–72, 1952.

[37] C.H. Waddington. *Principles of Development and Differentiation.* Macmillan Company, New York, 1966.

[38] D. Wessels, J. Murray, and D.R. Soll. Behavior of *Dictyostelium* amoebae is regulated primarily by the temporal dynamic of the natural cAMP wave. *Cell Motil. Cytoskeleton*, 23:145–156, 1992.

[39] O.K. Wilby and D.A. Ede. A model generating the pattern of cartilage skeletal elements in the embryonic chick limb. *J. Theor. Biol.*, 52:199–217, 1975.

[40] H.V. Wilson. On some phenomena of coalescence and regeneration in sponges. *J. Exp. Zool.*, 5:245–258, 1907.

Appendix: CPL program for *Dictyostelium* aggregation

(reprinted from [2])

```
#define SignalStrength 33
#define BaseCAMP 1
#define SignalDuration 60
#define OutputPeriod 500
#define SignalDelay 12
#define Factor 8.0 // Controls the rate of diffusion
#define Tau 12.0   // Controls the rate of decay of cAMP

simulation_size (102,102);
biochemical cAMP;
float cAMP,cAMPchange,clock;
direction cAMP_dir,random_dir;
integer relayPeriod;

tissue generic{
 if (location == point(51,51))
   differentiate_to PaceMaker;
 for_each_neighbor_do
   if (neighbor.tissue_type == environment) die;
 if (random(1,100) <= 10)
     differentiate_to slimeMold;
 else differentiate_to space;
}

tissue space{
  deriv cAMP = -(cAMP-BaseCAMP)/Tau; // cAMP removal due to phosphodiesterase
  for_each_neighbor_do
    deriv cAMP = (neighbor.cAMP - cAMP)/Factor; // diffusion
}

tissue PaceMaker{
  clock -= time_interval;
  if (clock < 0)
    deriv cAMP = SignalStrength;
  if (clock + SignalDuration <=0)
    clock = relayPeriod-SignalDuration-SignalDelay; // quit signalling
  deriv cAMP = -(cAMP-BaseCAMP)/Tau;
  for_each_neighbor_do
    deriv cAMP = (neighbor.cAMP - cAMP)/Factor;
}

tissue observer{
  if (int(time) mod OutputPeriod == 1)
    image state; // produces an image of the current state of each cell
}


tissue slimeMold{
```

```
  state waitForSignal:like core {
    // amoeba wait in this state for cAMP signal to arrive
    if (cAMPchange > 0.02 && clock < 0) {
      // This value reaches cells up to 57um away
      // once a signal arrives they determine the direction of the signal
      cAMP_dir = point(0,0);
      for_each_neighbor_do
        cAMP_dir += neighbor.direction * (neighbor.cAMP-cAMP);
      clock = relayPeriod;
      goto readyToSignal;
    }
  }
  state readyToSignal:like core {
    if (clock <= relayPeriod-SignalDelay) goto signal;
  }
  state signal:like core {
    deriv cAMP = SignalStrength;
    if (clock <=  relayPeriod-SignalDelay-SignalDuration){
      relayPeriod -= 10;
      if (relayPeriod < 180) relayPeriod = 180;
      goto waitForSignal;
    }
  }
  state core{
    clock -= time_interval;
    deriv cAMP = -(cAMP-BaseCAMP)/Tau;
    cAMPchange = 0;
    for_each_neighbor_do
      cAMPchange += (neighbor.cAMP- cAMP)/Factor;
    deriv cAMP = cAMPchange;
    if (random(1, relayPeriod) <= 2 && clock > 0) // move twice every relay period
      with_neighbor_in_direction cAMP_dir
        if (neighbor.tissue_type == space) move cAMP_dir;
        else clock = neighbor.clock;
    if (random(1,240) <= 1) { // one cell diameter every 240 seconds
      random_dir = random_direction;
      with_neighbor_in_direction random_dir
        if (neighbor.tissue_type == space) move random_dir;
    }
  }
}
cell {
  unit_area;
  type generic;
  start_up_area rectangle(1,1,101,101);
  cAMP = BaseCAMP;
  clock = 0;
  cAMP_dir = point(0,0);
  relayPeriod = 420;
}
```