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

Eric Mjolsness, David H. Sharp, and John Reinitz. A connectionist model of development. *JTB*, 152:429–453, 1991.

P.C. Newell. Aggregation and cell surface receptors in cellular slime molds. In J.L. Reissig, editor, *Receptors and Recognition: Series B Volume 3: Microbial Interactions*, chapter 1, pages 1–58. John Wiley and Sons, New York, 1977.

B. Novak and F.F. Seelig. Phase-shift model for the aggregation of amoeba: A computer study. *Journal Of Theoretical Biology*, 56:301–327, 1976.

Q. Ouyang and Harry L. Swinney. Transition from a uniform state to hexagonal and striped turing patterns. *Nature*, 352:610–612, 1991.

H. Parnas and L.A. Segel. Computer evidence concerning the chemotactic signal in *dictyostelium discoideum*. *Journal Of Cell Science*, 25:191–204, 1977.

H. Parnas and L.A. Segel. A computer simulation of pulsatile aggregation in *dictyostelium discoideum*. *Journal Of Theoretical Biology*, 71:185–207, 1978.

Robert Ransom. *Computers and Embryos: Models in Developmental Biology*. John Wiley, New York, 1981.

Kenneth B. Raper. *The Dictyostelids*. Princeton University Press, Princeton, NJ, 1984.

Gary Rogers and Narendra S. Goel. Computer simulation of cellular movements: Cell-sorting, cellular migration through a mass of cells and contact inhibition. *Journal Of Theoretical Biology*, 71:141–166, 1978.

R. Ransom and R.J. Matela. Computer modeling of cell division during development using a topological approach. *J. Embryol. Exp. Morphol.*, 83:Supplement, 233–259, 1984.

Malcolm S. Steinberg. Does differential adhesion govern self-assembly processes in histogenesis? equilibrium configurations and the emergence of a hierarchy among populations of embryonic cells. *JEZ*, 173:395–434, 1970.

Malcolm S. Steinberg. Reconstruction of tissues by dissociated cells. In G.D. Mostow, editor, *Mathematical Models for Cell Rearrangement*, chapter 6, pages 82–99. Yale University Press, 1975.

Deborah Sulsky. *Models of Cell and Tissue Movement*. PhD thesis, Courant Institute of Mathematical Sciences, June 1982.

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.

D'Arcy Thompson. *Of Growth and Form*. Cambridge University Press, Cambridge, 1917.

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

Richard Cowan and Valerie B. Morris. Division rules for polygonal cells. *Journal Of Theoretical Biology*, 131:33–42, 1988.

M.H. Cohen and A. Robertson. Chemotaxis and the early stages of aggregation in cellular slime molds. *Journal Of Theoretical Biology*, 31:119–130, 1971.

M.H. Cohen and A. Robertson. Wave propagation in the early stages of aggregation in cellular slime molds. *Journal Of Theoretical Biology*, 31:101–118, 1971.

Kurt Fleischer. Generating heterogeneous asymmetric artificial neural networks using developmental models. In *Symposium on Pattern Formation*, 1993.

Narendra Goel, Richard D. Campbell, Richard Gordon, Robert Rosen, Hugo Martinez, and Martynas Yčas. Self-sorting of isotropic cells. In G.D. Mostow, editor, *Mathematical Models for Cell Rearrangement*, chapter 7, pages 100–144. Yale University Press, 1975.

Scott F. Gilbert. *Developmental Biology.* Sinauer Associates, Sunderland, MA, 1991.

Richard Gordon. On stochastic growth and form. *Proceedings of the National Academy of Sciences*, 56:1497–1504, 1966.

Narendra S. Goel and Gary Rogers. Computer simulation of engulfment and other movements of embryonic tissues. *Journal Of Theoretical Biology*, 71:103–140, 1978.

Hisao Honda. Description of cellular pattern by dirichlet domains: The two-dimensional case. *Journal Of Theoretical Biology*, 72:523–543, 1978.

Ardean G. Leith and Narendra S. Goel. Simulation of movement of cells during self-sorting. In *Mathematical Models for Cell Rearrangement*, chapter 9, pages 159–175. Yale University Press, 1975.

Aristid Lindenmayer. Mathematical models for cellular interaction in development, i & ii. *Journal Of Theoretical Biology*, 18:280–315, 1968.

S.A. MacKay. Computer simulation of aggregation in *dictyostelium discoideum. Journal Of Cell Science*, 33:1–16, 1978.

Hans Meinhardt. *Models of Biological Pattern Formation.* Academic Press, New York, 1982.

Raymond J. Matela and Robert J. Fletterick. A topological exchange model for cell self-sorting. *Journal Of Theoretical Biology*, 76:403–414, 1979.

Raymond J. Matela and Robert J. Fletterick. Computer simulation of cellular self-sorting: A topological exchange model. *Journal Of Theoretical Biology*, 84:673–690, 1980.

G.D. Mostow. *Mathematical Models for Cell Rearrangement.* Yale University Press, 1975.

Raymond J. Matela and Robert Ransom. A topological model of cell division: Structure of the computer program. *BioSystems*, 18:65–78, 1985.

6 Conclusions and Future Work

CPL is a computational tool which provides insight into biological models. The two examples we have in this paper demonstrate the versatility of the language. Cellular sorting works on adhesive differences, and *Dictyostelium* aggregation works with a chemical signal relaying mechanism. CPL can also be used to simulate reaction diffusion mechanisms.

We are working on further improvements to the *Dictyostelium* aggregation model, and on modeling precartilage formation in chicken limb.

We need to examine more characteristic growth and division behavior which would lead to improved definitions and implementations of those instructions. It is not clear as to what cell characteristics should be conserved by the growth and division operations.

Is it possible to automatically determine invariants under these programs? Perhaps a modified form of the law of conservation of mass and energy can be implemented by accounting for the energy each cell operation consumes and the energy stored in each biochemical. This would ease writing of correct CPL programs.

Can physical forces be accounted for in a simple manner? Cells do exert forces (attractive at medium distances, repulsive at close proximity) on each other.

Computationally, the challenge is to extend the simulations to three dimensions. One of the considerations in designing CPL was that we should be able to model cellular behavior in some generality, yet be able to simulate a significant number (thousands) of cells. While a thousand cells is a reasonable number of cells in two dimensions, in three dimensions it would be just a cube 10 cells wide, which is perhaps not enough to observe biologically significant behavior. Exploiting the inherent parallelism in the language is the key for extending the work to three dimensions. Some of CPL instructions, move and grow, require the cooperation of neighboring cells. Parallel implementations of these instructions would require further insight into cell behavior.

Acknowledgments: I would like to thank Jacob T. Schwartz, who suggested examining the programming of cells; Jerome K. Percus for the many inspiring discussions; Stuart A. Newman for the biological insights; and Laureen C. Treacy and Michael J. Hind for careful proofreading and encouragement.

References

Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James D. Watson. *Molecular Biology of the Cell.* Garland Publishing Inc., New York, 1989.

F. Alacantra and M. Monk. Signal propagation during aggregation in the slime mold *dictyostelium discoideum. Journal of General Microbiology*, 85:321–334, 1974.

P. Antonelli, T.D. Rogers, and M. Willard. Cell aggregation kinetics. In G.D. Mostow, editor, *Mathematical Models for Cell Rearrangement*, chapter 10, pages 176–195. Yale University Press, New Haven, 1975.

J.T. Bonner. *The Cellular Slime Molds.* Princeton University Press, Princeton, NJ, 1967.

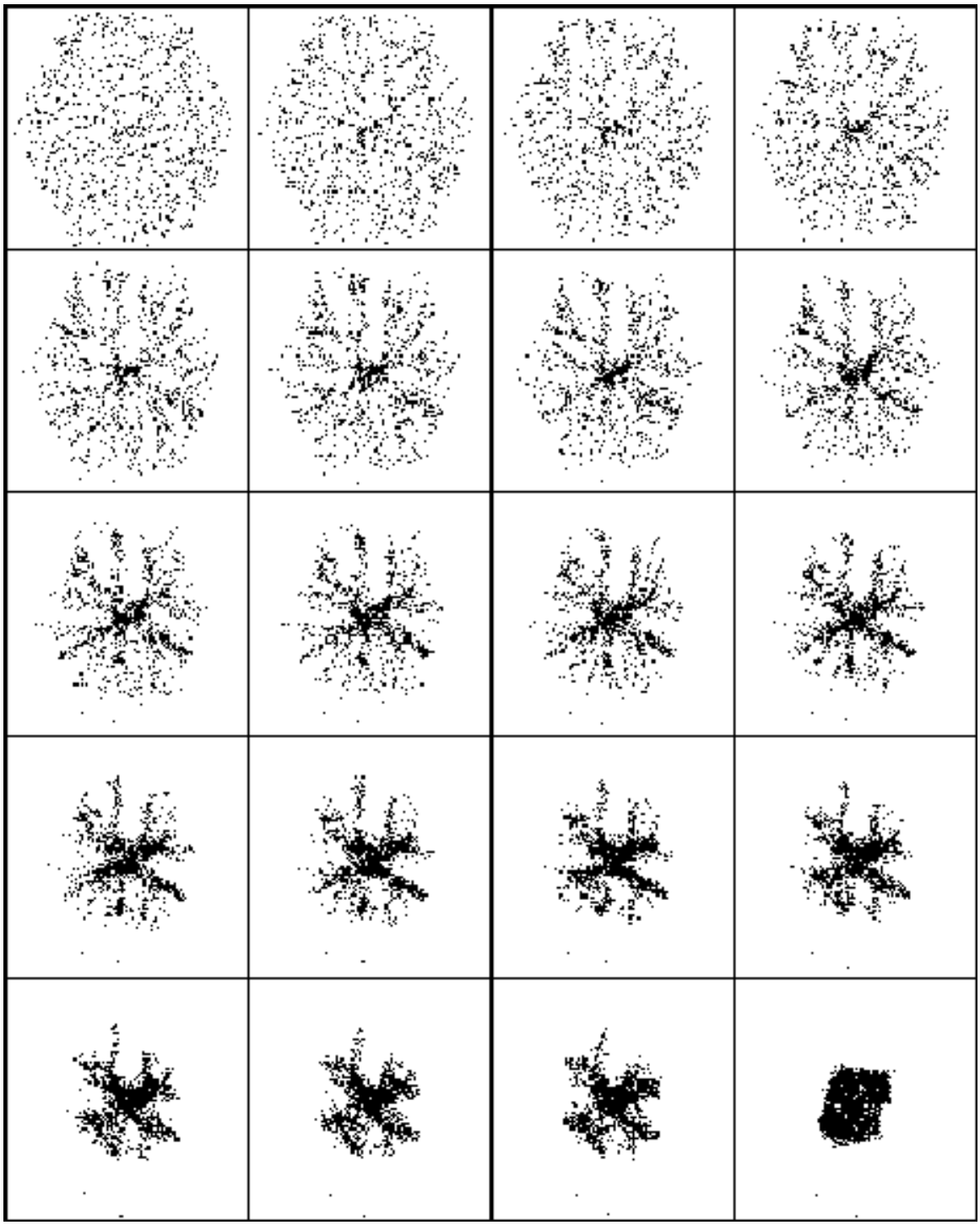

Figure 8: Aggregation in *Dictyostelium*. Each point represents a *Dictyostelium* amoeba. The subframes are images taken after each successive refractory period (140 time steps of simulation). The last subframe shows the final aggregate and is after 7000 time steps of the simulation.

```
tissue generic {
    // make sure only internal cells can be dictyostelium
  for each neighbor do
    if (neighbor.tissue_type == environment) differentiate_to space;
  if (random(1, 10) == 1) // select 10% of the remaining cells
    differentiate_to dictyostelium;
  else differentiate_to space;
}

tissue observer{ // This is a dummy tissue which is used to output information
    image tissue;
}

cell { // initializes all the cells
  unit_area;
  type generic;
  start_up_area hexagon( 52, 52, 50);
  cAMP = 0;
  step = 0;
}
```

Figure 7: CPL program for aggregation in *Dictyostelium* . (continued)

```
// cellular slime mold aggregation
#define FACTOR 12 /* diffusion factor for cAMP */
#define SignalDelay 8 /* Signal Delay = 8 time steps (ts) = 12 sec */
#define RefractoryPeriod 140 /* Refractory period = 140ts = 210sec*/
#define AcrasinaseAct 1 /* Amount of cAMP/ts destroyed by phosphodiesterase*/
simulation_size (105,105);
biochemical cAMP;
integer step;
direction cAMP_dir;

tissue dictyostelium{
  state waitForSignal:like core {
    cAMP_dir = point(0,0);
    for each neighbor do
      cAMP_dir += neighbor.direction * (neighbor.cAMP- cAMP);
    if (cAMP_dir*cAMP_dir > 1) { // measures the magnitude of cAMP_dir
      // if signal is strong enough
      step = 0;
      goto signal;
    }
  }
  state signal:like core {
    if (step >= SignalDelay) {
      //signalling delay to let the original signal get ahead
      step = 0;
      deriv cAMP = 5000;
      goto refractoryPeriod;
    }
  }
  state refractoryPeriod:like core {
    if (step == 33 or step == 66) {
      // cell motion: 2 cell diameters in 66 time steps
      with neighbor in direction cAMP_dir
        if (neighbor.tissue_type == space) move cAMP_dir;
    }
    else if (step >= RefractoryPeriod) goto waitForSignal;
  }
  state core{
    step += time_interval;
    if (cAMP >= AcrasinaseAct) deriv cAMP = -AcrasinaseAct;
    for each neighbor do
      deriv cAMP = (neighbor.cAMP- cAMP)/FACTOR;
    if (random(1,80) == 1) move random_direction; // 1 cell diameter every 80ts
  }
}

tissue space{ // dummy cells used to fill up space
  for each neighbor do
    deriv cAMP = (neighbor.cAMP - cAMP)/FACTOR;
  if (cAMP >= AcrasinaseAct) deriv cAMP = -AcrasinaseAct;
}
```

Figure 6: CPL program for aggregation in *Dictyostelium* .

the amoebae move inward. Due to the relay of the pulse by the responding amoebae, their radial motion is unstable, and the amoebae tend to gather into streams, which increasingly act as strong local sources of attraction. Amoebae can, at times, even be observed to move outward from the center in order to join a stream that happens to run behind them. Eventually, the amoebae in these moving streams all reach the aggregation center and a compact aggregate of cells is formed. This summary of the aggregation procedure has been paraphrased from Newell (New77).

The model

We wrote programs in CPL to model this aggregation. We modeled each *Dictyostelium* amoeba by a single point in our discrete space. We used a discrete space of about 100x100, which translates to $1mm \times 1mm$ of real space. The number of amoebae is about 10% of the total points, about 1000.

We used arbitrary units (a.u.) for cAMP concentration. 1 a.u. of cAMP was the threshold signal. Phosphodiesterase removed 1 a.u. of cAMP from every cell if the concentration of cAMP was higher than 1 a.u. in that cell. The signal strength of each amoeba was 5000 a.u. This number was determined from the fact that the cAMP from the previous signal should be removed before the new signal arrives. The signals may be as close as 150 seconds; therefore, the time taken to remove this signal from the surroundings should be less than 150 seconds.

Each time step (ts) in our simulation represented 1.5 seconds. Therefore, the signalling delay = 12 seconds = 8 ts, and the relay refractory period = 150 seconds = 100 ts. *Dictyostelium* takes 100 seconds = 66 ts to move $20\mu m$ = 2 cell diameters, which translated to 33 ts for moving 1 cell diameter. There was also a random motion of $5\mu m/min$ equivalent to 1 cell diameter every 2 minutes.

Figures 6 and 7 contain a program similar to the one used to produce the aggregation images in figure 8. For brevity and clarity some of the instructions have been removed (for example, the cell which starts the signalling process). The cells cycle between 3 states:

- waiting for a cAMP signal to arrive (detected by the magnitude of the signal direction being greater than 1);
- relaying the signal after waiting a short duration;
- moving towards the source of the original signal for 100 seconds, and then going back to the state of waiting for a signal.

In all these states, the cell computes the new cAMP values due to diffusion and destruction by phosphodiesterase, and has a small random speed (state core).

5.2 Aggregation in Cellular Slime Mold

Dictyostelium discoidea is a free-living amoeba which has been termed the hydrogen atom of developmental biology. It has an intriguing life cycle. In its vegetative cycle, solitary amoebae survive until they exhaust their food supply. When they have exhausted their food supply, tens of thousands of these amoeba join together, to form moving streams of cells that converge at a central point. There they form a conical mound, which eventually absorbs all the streaming cells, and bends over to produce the migrating slug. The cells in the slug differentiate into two varieties: stalk cells and spore cells, which together form a fruiting body. The spore cells disperse, each one becoming a new amoeba.

The cellular aggregation is not due to a simple radial movement. Rather, cells join with each other to form streams; the streams converge into larger streams, and eventually all streams merge in the center. This motion has been shown to be due to chemotaxis, the chemical involved being cyclic adenosine monophosphate (cAMP). There is no *dominant* cell or predetermined center. Neighboring cells respond to the cAMP in two ways: they initiate a movement towards the cAMP pulse, and they release cAMP of their own. Following this stage the cell is unresponsive to further cAMP pulses for several minutes (Gil91).

This life cycle has been well researched, and there is a wealth of experimental data available (AM74; Bon67; Rap84; New77). Tomchik and Devreotes accumulated quantitative data on the cAMP wave patterns (TD81). Cohen and Robertson have mathematically analyzed the aggregation process (CR71b; CR71a). Parnas and Segel simulated the aggregation of 40 slime mold cells arranged in a row (PS77; PS78). Novak and Selig conducted simulations on two-dimensional (49×49) aggregate of slime mold cells (NS76). Mackay has conducted extensive of simulations on various aspects of slime mold aggregation (Mac78), with images which closely mimic *Dictyostelium* aggregation. Our results are similar to those obtained by Mackay but are achieved using a simple discrete model.

Quantitative examination

We need to examine the aggregation quantitatively in order to model it in CPL. *Dictyostelium* is typically $10\mu m$ in diameter, and their aggregation may involve up to 100,000 amoebae from as far as $20mm = 20,000\mu m = 2,000$ cell diameters. The initial stimulus needed to start aggregation is starvation. This induces in some cells the ability to produce slow rhythmic pulses of the attractant acrasin (cAMP) with an initial frequency of roughly 1 pulse every 10 minutes. Meanwhile, the rest of the starving population produce cAMP receptors on their surface which enables them to receive the pulsed signal. The cAMP signal does not diffuse far from the centers of its production but is destroyed within $57\mu m$ (roughly 6 cell diameters) by acrasinase (a phosphodiesterase enzyme) also produced by starving amoebae.

Two responses to this signal are observable. The amoebae begin to move in the general direction of the signal source (and continue to move for 100 seconds covering $20\mu m = 2$ cell diameters), and about 12 seconds after receiving the signal (signalling delay), the amoebae themselves emit a pulse of cAMP. The amoebae closer to the center are prevented from relaying the signal inward by being refractory to further relay stimulation for 2 to 7 minutes after producing the signal (the relay refractory period). By this system of relay, a series of waves of cAMP production, destruction and response, move outward from the center as

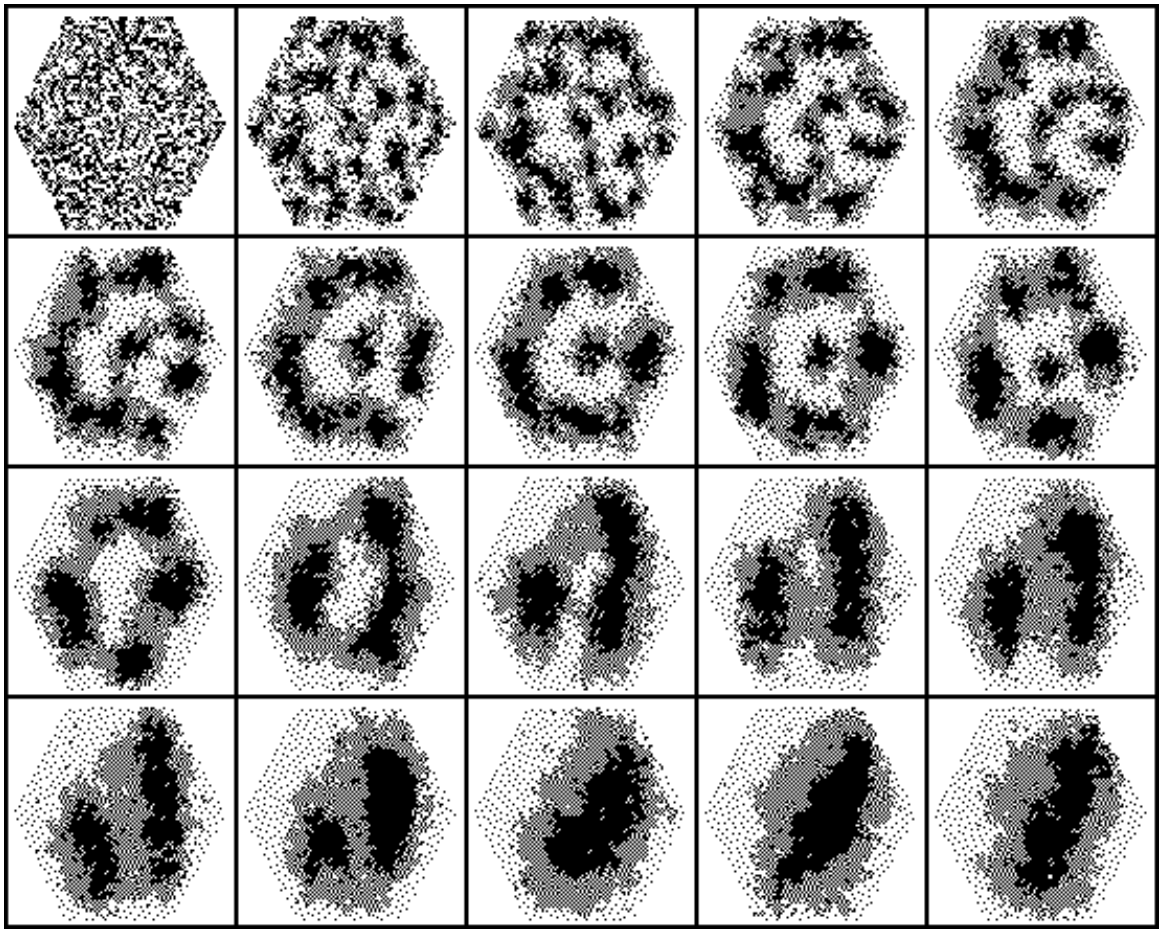

Figure 5: Sorting of tissues using adhesivity differences. The darkest points belong to the tissue with the highest adhesivity (they aggregate centrally), and the lightest ones have the weakest adhesivity (they aggregates externally). The subframes are after 1, 20, 60, 120, 200; 300, 500, 800, 1300, 2000; 3000, 6000, 9000, 12000, 15000; 18000, 21000, 24000, 27000 and 30000 simulation time steps.

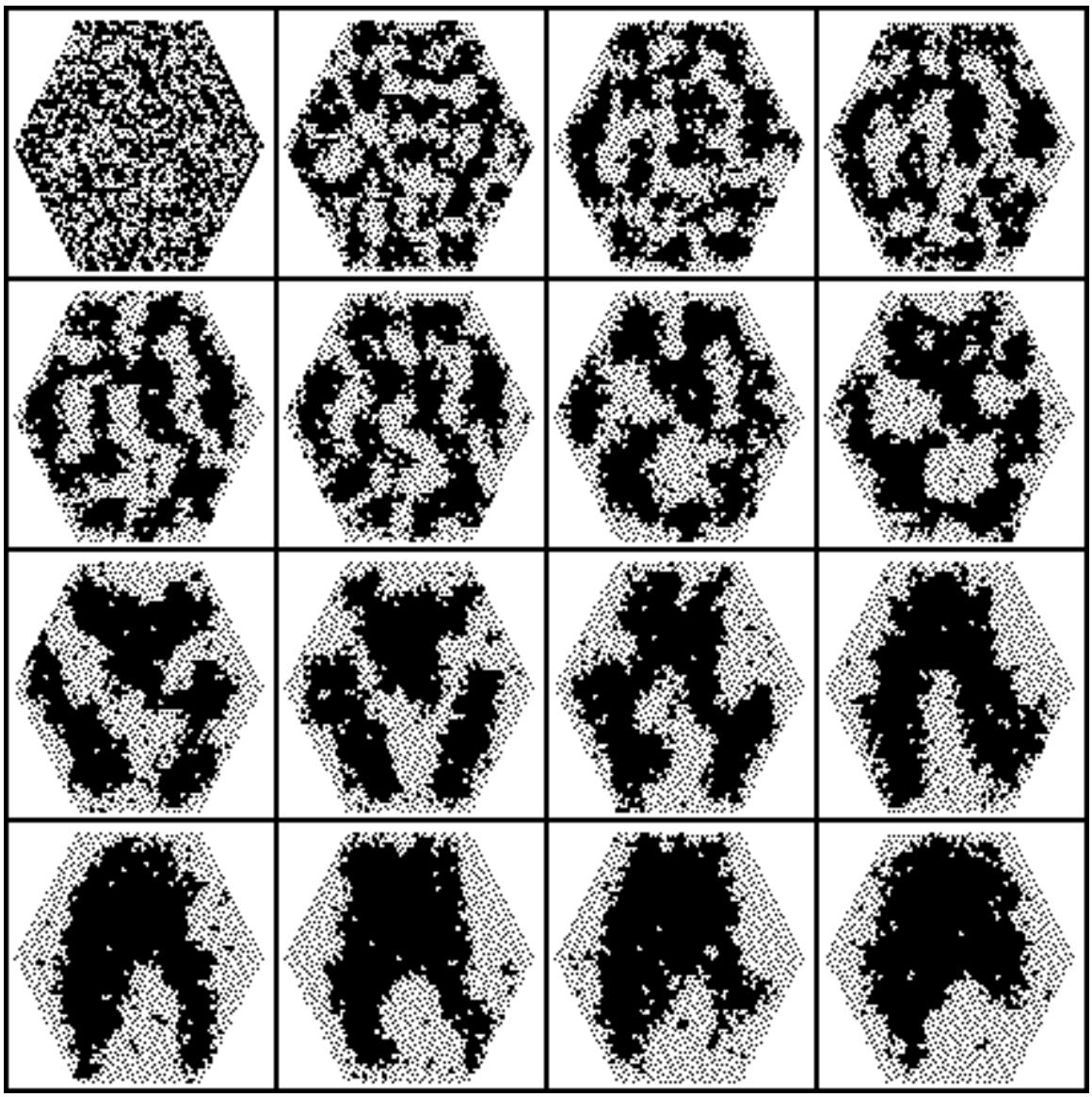

Figure 4: Sorting of two tissues using adhesivity differences. Each point represents a cell, and the two different tissue types are represented by light and dark points. The subframes are after 1, 20, 60, 120; 200, 300, 500, 800; 1300, 2000, 3000, 5000; 10000, 15000, 20000 and 25000 simulation time steps.

```
#define ContactA 4
#define ContactB 1
simulation_size (53,53);
integer contactSum;
integer contact;
vector dir;
integer best;
integer thisExchange;

tissue generic{
  state splitUp{
     if (random(0,1) == 0) contact = ContactB;
     else contact = ContactA;
     goto motion;
  }
  state motion{
    contactSum = 0;
    for each neighbor do
      contactSum += neighbor.contact;
    best =0;
    for each neighbor do
      if (neighbor.contact != contact) {  //different tissue type
        thisExchange =
            contactSum*neighbor.contact+neighbor.contactSum*contact -
            (contactSum*contact+neighbor.contactSum*neighbor.contact) -
            (contact - neighbor.contact)*(contact - neighbor.contact);
        if (thisExchange > best) {
          best = thisExchange;
          dir = neighbor.direction;
        }
      }
    if (best > 0 and random(0,1) == 0) move dir;
       // half the time the best move is made
    if (random(0,3) == 0) move random_direction;
       // quarter of the time a random move is made
  }
}

tissue observer{
    image contact;
}

cell{
  unit_area;
  type generic;
  start_up_area hexagon(26,26,25);
}
```

Figure 3: CPL program for sorting of tissues using adhesivity differences.

cells, and pigmented retinal cells migrate towards the center when mixed with neural retinal cells.

- The central migration is transitive. If A sorts internally to B and B sorts internally to C, then A sorts internally to C. In fact there is a complete hierarchy among cell types of such central migration.

Steinberg concluded that cells adhere to one another with varying strengths, and interact to maximize adhesive energy (or minimize free energy). This thermodynamic model was termed the "Differential Adhesion Hypothesis" model. In section 2.1, we have already discussed some of the simulations which have used this model. [9]

We tested the differential adhesion hypothesis using CPL . The results we obtained are a validation of the hypothesis. In fact they are significantly better than the results obtained by Goel et. al. (GR78). The improvement was mainly due to the addition of random motion. The simulations by Goel et. al. produced only local clumping. In CPL simulations almost all the cells formed a single clump; however, the clump does not completely round up. The difficulty of getting the smaller clumps to combine and to form larger clumps increases exponentially with clump size.

Figure 3 on page 21 contains the CPL program that was used to test the differential adhesion hypothesis. The program specifies a simulation starting with a hexagonal arrangement of cells in a hexagon of radius (side) 25, each cell of unit area. These cells, numbering about 2000 are of a *generic* tissue type, and randomly choose the number of contacts on their surface to be either 1 or 4. The adhesion between cells is proportional to the product of the contacts on their surface. Thus cells with different contacts represent different tissue types.

The first definition is that of a generic tissue, which uses a random number generator to split the tissue into cells with differing contact strengths. Each cell at every time step computes its own energy by summing up the number of contact sites on the cells surrounding it. The adhesive energy is given by the product of these contact sites and the number of the cell's own contact sites. The cell computes the move to a neighboring point, which would result in the maximum increase in the adhesive energy, and with probability half it makes that move. Cells also move in a random direction once every 4 time steps.

Figure 4 on page 22 and figure 5 on page 23 contain the images produced by the simulation. Figure 4 represents sorting out of two different tissue types. Figure 5 represents the central movement that occurs when 3 different tissues are mixed together.

[9] The volume by Mostow is a good collection of papers on cellular sorting and engulfment (Mos75).

similar to another. This is done 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*) which contains all the biochemical interaction and diffusion equations, because these are for the most part invariant in a cell's history.

- We do not model intercellular space implicitly; instead, intercellular space is modeled by specifying dummy cells which act as intercellular space.
- The chemical concentration in the environment of the biochemicals 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. The biochemical should not fluctuate in a cell between extremes with every time step. In that case diffusion is not being correctly modeled.

5 Applications

5.1 Sorting of tissues

Dissociated embryonic tissue reaggregates to resemble the original structure. Wilson in 1907 discovered that a dissociated sponge cells would reaggregate to form a functional sponge structure. Steinberg has extensively investigated the phenomena of reaggregation and sorting out of cells in embryonic tissue (Ste70) (Gil91, page 532). Steinberg observed:

- A fragment of tissue rounds up into almost spherical shape.
- If two different cell types are mixed with each other, they sort out, with one of the cell types always aggregating centrally and the other cell type surrounding it. For example, heart cells migrate towards the center when mixed with pigmented retinal

perform other actions. It has a real value which 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 change its value by an assignment statement.

- `random`: This provides random numbers, which may be used to take probabilistic actions. $random(x, y)$ is a function call which provides a random integer between x and y. For example, this may be used to have a tissue differentiate 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 therefore, this probabilistic nature is captured by providing a random number generator.

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

4.6 Other CPL features

In this subsection we explain some CPL features which ease writing CPL programs.

- The first state in the tissue is the default state. If a tissue has just one state, it need not be named. This is done by omitting the state name declaration.

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

- A tissue *environment* is defined by default. The points in the simulation space that are not explicitly defined as belonging to a specified tissue, belong to this environment tissue.

- 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 should be used to collect results (or display images). Thus, static variables which are collecting data over the entire cell aggregate may be reinitialized at each time step in the code for the *observer*, since 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*, which lets one specify if a state is

cells access biochemical concentrations of their neighbors, they are all consistent and independent of the order of execution of the cell programs.

- `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 which 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 be used as a read only variable. Its value cannot be changed by the assignment statement. There exists a separate instruction called `grow` which 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 be used as a read only variable, whose value is affected by the instructions `grow` and `divide`. The relationship between these instructions and the value of the perimeter is indirect.

- Neighbor attributes: The physical presence leads to well-defined neighboring cells, whose attributes 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.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.
  - `neighbor.direction` is the direction to the neighboring cell. This is the composite direction determined by a walk along the boundary with the neighbor. The magnitude of this vector represents the size of the contact.
  - `neighbor.contact_length` is the size of the interface with the neighboring cell.
  - `neighbor.perimeter` is the perimeter of the neighboring cell.
  - `neighbor.`<biochemical_name> is the biochemical concentration in the neighboring cell of the given biochemical. The biochemical concentration accessed is the concentration in the neighbor at 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 (age) which tracks their progress in their life cycle (for example, the time since the last division), and its neighboring cells may need this information (neighbor.age) so as to synchronize their own life cycle.

- `time`: It is not unreasonable 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

- **hexagon** 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 2, it maps onto a regular hexagon. The general form of the instruction is:

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

- **triangle** 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. The general form of the instruction is:

  `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 a cell definition contains the reserved word `unit_area`, the entire area of the cell is split up into many different cells, all with the same tissue type and unit area. This enables the definition of arbitrary shaped cellular array (with the constraint that each have unit area). Thus 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; thus, if their areas overlap, the cell declared last receives the area in question.

4.5 Accessing cell attributes

In Section 2, the cell attributes were briefly listed; in this section each of them is discussed in greater detail, giving both syntax and semantics.

- tissue type: Although 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 throughout the cell aggregate, 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

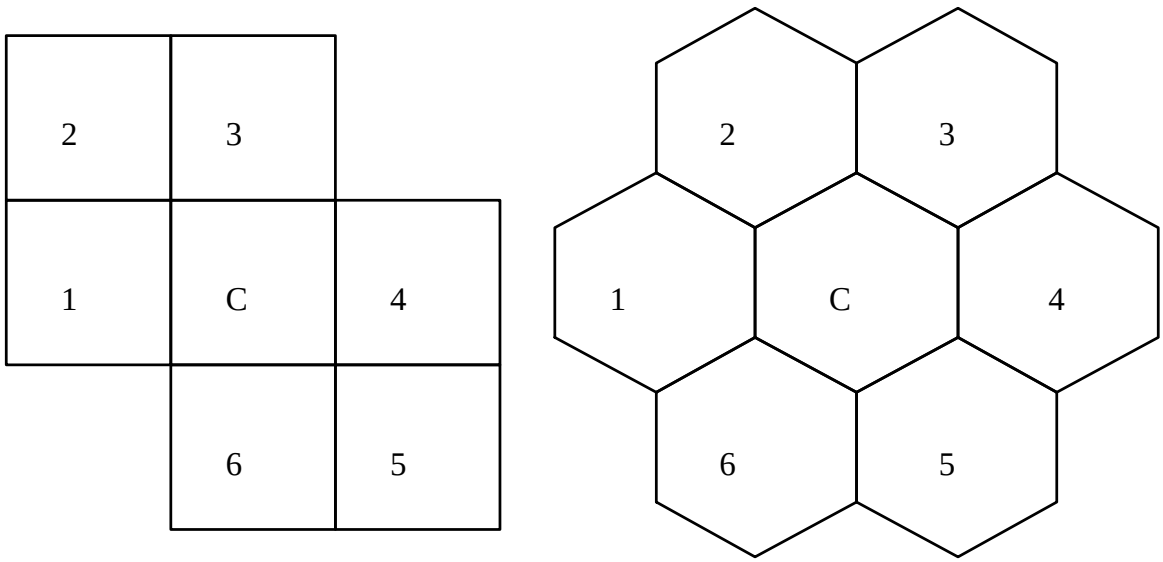

Figure 2: The map transforming the square lattice to a hexagonal lattice.

`type` <tissue-name>

- **assignment** 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 this context (i.e. initialization of cells).

- **start_up_area** 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. We then use the map given in figure 2 on page 15 to convert this square lattice to a hexagonal lattice (since cells contain points on the hexagonal lattice). Six of the eight neighboring point on the square lattice map to the neighbors for the hexagonal lattice.

  The general form of the `start_up_area` is:

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

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

- **rectangle** declares a rectangular region of points as part of a cell. It takes four integers as parameters (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. Its general form is:

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

- **circle** 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 its radius. Its general form is:

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

4.3 Variable declarations

Variables have three characteristics: type (integer/real/direction), variable/biochemical, and static/local. Integer, variable, and local are the default values.

- **Type**: The identifier must be declared to be an integer, float, or direction. Biochemicals may only be either integer or float. The general forms are:

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

  <id-list> is a list of identifiers separated by commas.

- **Biochemical**: Biochemicals need a special declaration. All variables are assumed not to be of the biochemical variety unless there is a biochemical declaration for them. The general form of the declaration is:

  `biochemical` <id-list>

- **Static**: 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, since in the biological context `static` variables permit non-local communication. The general form of the static declaration is:

  `static` <id-list>

4.4 Cell declarations

Before running a simulation, the locations of all the cells initially present 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** declares the tissue type of the cell. This is followed by an identifier which is the tissue name. Its general form is:

- **simulation_size:** The cells being modeled as a collection of discrete integral points, the user has to specify the maximum size (area) the simulation may occupy. Thus if the `simulation_size` is declared as (25, 30), 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. It should be ensured that the cells do not grow past a boundary, so enough space should be provided. This declaration must be the first statement in the program.

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

- **time_interval:** The user can specify the time interval at which the program for each cell is executed. It is also the time difference with which the biochemical difference equation is implicitly multiplied. Thus, if `time_interval` = 5, then every step of the simulation corresponds to 5 time units of the model, 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. Thus, if some variable is monitoring the elapsed time in a cell, it should be incremented by `time_interval`. It is again 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** 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** takes an expression as an argument and prints out the expression value on standard output.

  `write` <expression>

- **image** takes a biochemical or variable name as an argument and prints out a matrix of that particular biochemical/variable's value by cell location. This can be used to view simulation results in image form.

  `image` <biochemical/variable-name>

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

  `save`

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

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

(if the default copying is not satisfactory) of the variables as a result of this split; specifically, how the biochemicals divide.

```
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. The direction can either be a specified by a variable (perhaps representing a biochemical gradient, in which case the cell grows preferentially along that direction), or a `random_direction` , in which case the cell's growth direction is randomly chosen.

  The general form of the grow instruction is:

  grow <integer-expression> <direction-expression>

  ```
  grow 5 random_direction
  ```

  The above instruction causes the cell to grow in area by 5 units in randomly chosen directions. A cell is represented as a collection of discrete points. Growth by 5 units is equivalent to adding 5 lattice points to the cell.

  Growth 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. An alternative implementation of growing a cell in a direction is by computing its boundary and randomly choosing a growth size number of points on the boundary. The chosen point should be such that their neighboring point in the growth direction belongs to a different cell. This cell then steals these neighboring points. 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 would have an equal chance of being picked, resulting in a rectangular cell.

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

  ```
  die
  ```

4.2 Meta-instruction list

These instruction are not biological in nature. They enable us to control the simulation, make the program readable, and help visualize the results.

the sum of the direction to its neighbors weighted by the amount of the biochemical diffusing from the respective neighbor.

- **exit:** The `exit` may only be used inside a `for each neighbor do`, and it causes execution to skip the remaining instructions inside the `for each neighbor do`, and go to the instruction succeeding the `for each neighbor do`. `exit` may be used to 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 which neighbor to use (if any) after the loop exit. The general form of the exit instruction is:

  `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. `with neighbor in direction` then enables accessing the attributes of this specific neighbor.

  The general form of the `with neighbor in direction` instruction is:

  `with neighbor in direction` <direction-expression> <instruction>

  The net flow of *BcOne* can be determined, as in the `for each neighbor do` example; `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 is of tissue type A, the cell swaps locations with this neighbor.

- **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 in which the cell should divide. The choices are horizontal, vertical, perpendicular to last division, random (any), and shortest dividing lines.[8] The area is split up, and the values of all the variables of the parent (including biochemicals) are copied to the children. Each of the two daughter cells then finishes executing the state definition separately. The code that the children execute immediately after the divide should determine the values

[8] Each of these choices is represented by a reserved word.

The general forms of this instruction are:

`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
```

- **for each neighbor do:** This permits the execution of an instruction (or a block of instructions) using the parameters of each of the neighbors in sequence. This instruction takes another instruction as its argument. Some extra variables are available in the scope of the `for each neighbor do`, specifically, the area of the neighbor; direction to the neighbor; contact length with the neighbor; the tissue type 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 instruction with each neighbor in turn. It randomly picks up the first neighbor and then cycles through the remaining by going around the cell boundary. The `exit` instruction may be used to exit the loop. Nested `for each neighbor do`'s are not allowed.

  The general form of the `for each neighbor do` instruction is:

  `for each neighbor do` <instruction>

```
for each neighbor do
  deriv BcOne = (neighbor.BcOne − BcOne)/Drate;
```

The above statement 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.[7] In a similar fashion, diffusion for the other biochemicals in the cell could be represented. The 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;
}
```

This enables the accumulation of the direction of the biochemical diffusion, which is

[7] The user should ensure that the cells do not cheat in diffusing biochemicals. If a cell receives some amount of biochemical due to diffusion, some other 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 since all the cells access the same biochemical values at any time.

The move is well defined if the two cells excels exchanged are of equal size; however, the move is not as well defined when cells of unequal sizes are involved.[5]

`move` *direction1*

The general form of the move instruction is:

`move` <direction-expression>

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.

- **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 would be used to cycle between a set of states (such as *waitForSignal, signal*, and *refractoryPeriod*).[6] These states consist of a set of instructions specified by the user. There are no predefined states.

  The general form of the goto instruction is:

  `goto` <state-name>

  `goto` *waitForSignal*

- **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).

  The general form of the `differentiate_to` instruction is:

  `differentiate_to` <tissue-name>

  `differentiate_to` *epithelial*

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

- **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`).

[5] In the unequal area case, the `move` conserves the areas of all the cells involved; however, their shape need not be conserved.

[6] This is used in the cellular slime mold example in section 5.2.

the form:

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

`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 also implicitly multiplied by the size of the time step in the simulation (which is further explained on page 13). Moreover, the addition to biochemical values is only effected at the end of the time period. The above equation assumes the following reaction:

$$BcOne \rightleftharpoons BcTwo + BcThree$$

with the rate of production of BcOne ($d\ BcOne/dt$) being given by

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

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

The biochemical assignment statement in CPL resembles the above equation, though the multiplication of the right hand side by Δt is implicit. Δt should be small for the approximation to be valid.

In addition, the biochemical concentrations can also be set to a specified value by using the simple assignment (without using the `deriv` reserve 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. This statement is primarily used to set initial concentrations.

Thus 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 in a specified direction by exchanging the location of the cell with that of a neighboring cell in the given direction.

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

Table 1: CPL operators

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.

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. All reserved words are in lower case. All examples of CPL instruction sequences are placed in rectangular boxes.

CPL uses a subset of the language C's operators. These are listed in table 1.

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 brief descriptions of the syntax and semantics of each of them.

4.1 Instruction list

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

  ```
  step = 1
  ```

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

  ```
  step += 1
  ```

  ```
  step -= 1
  ```

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

Cell attributes

To complete the model we need to examine the attributes of a cell. CPL provides us with a mechanism for specifying operations on these attributes.

The main cell attributes are its tissue type, biochemicals concentrations, and a physical presence (which defines its neighbors).

- **Tissue Type:** Each cell has a specific tissue type which dictates the cell's response to its environment. The tissue type determines what program the cell executes. The program may be thought of as representing the cell's genome, the effect of the environment on the cell, and the physical chemistry of the cell constituents.
- **Biochemicals:** The concentrations of all the biochemicals present in a cell, along with their equations of catalysis and diffusion, can be specified. These concentrations may represent either the interior or the surface concentrations. These equations determine the biochemical concentrations in the cell at the end of the current time step. Cells are modeled to be homogeneous; therefore, the biochemical concentration are uniform inside the cell.
- **Physical presence:** A cell has the attributes of area, perimeter, and neighbors (other cells). Only cells in direct physical contact are treated as neighbors. This neighbor attribute forms the basis for all intercellular communication. Cells can sense the attributes of their neighbors, and react accordingly. Biochemical diffusion also depends on the biochemical concentrations in the neighbors.
- **Neighbor's attributes** A cell can sense its neighbor's attributes: tissue type, biochemical concentrations, area, perimeter, the contact length between the two cells, as well as the direction in which that neighbor lies.

In addition to the biologically motivated attributes listed above, other attributes are required to write programs for these cells. These are variables used to store information about the cell. The variables have specific types, and may either be integer, real, or direction. Direction variables store values of the form (x,y), where x,y are integers or reals. Direction variables may be used to compute and store the direction of diffusion of a biochemical. Integer or real variables may store the number of divisions, time in a particular state etc.

To access and modify all the cell attributes described above, a set of instructions is needed; this is explored in section 4.

4 The language

As a prelude to discussing the syntax and semantics of the instructions, we present the notation we use. Reserved words appear 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, <*instruction*> refers to any single instruction, or a block

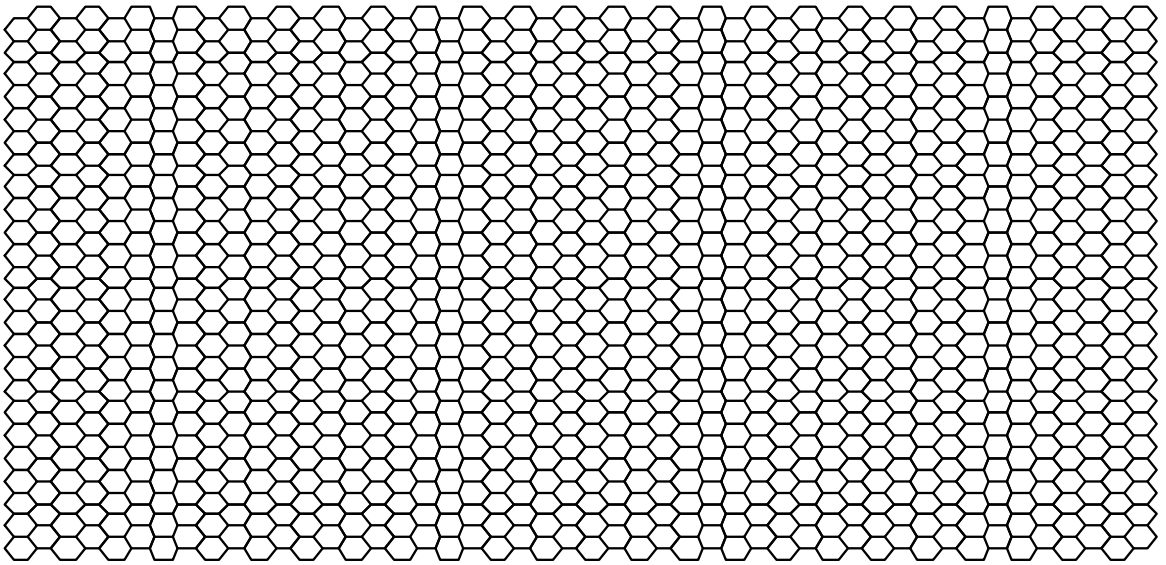

Figure 1: Hexagonal topology. Each cell occupies one or more hexagons.

model is optimal. [3] It adds little overhead in the case where all the cell are simple. If they all have the same shape and size, then each cell can be represented by a single point.

In CPL , we chose to represent each cell by a collection of discrete connected points. These points can be regarded as the points in a hexagonal lattice.[4] Figure 1 contains an hexagonal lattice structure. In hexagonal lattices, each point has six equidistant neighbors. A hexagon in the figure represents a lattice point. 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.

In section 4 we explain how different operations on cells are performed given this physical representation.

A CPL program for a single cell consists of a set of states. In each state, rules are specified which 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. Thus a cell could be in a state awaiting a signal , and once it receives a signal, it enters the state in which chemotactic movement takes place.

Each cell has a tissue type associated with it. Cells of the same tissue type execute the same 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.

[3] Modeling each cell as a polygon in continuous space is too computationally expensive to be useful for modeling a reasonable number of cells.

[4] The hexagonal structure where each lattice point has six neighbors is better defined than the four or eight neighbor lattice structure, which violates the Jordan Curve Theorem.

which implies that in the cell map no corner has a degree more than three (i.e. no four cells meet at a point).

The advantage of this model is that it permits cells with different shapes (any n-gon). However, representing the cell map by its dual graph also presents problems. The graph does not define a unique cell map, nor does it specify the size of each cell or the contact length with other cells.

D'Arcy Thompson considered the problem of how cell division contributes to producing particular shapes and patterns during development (Tho17) . Cowan and Morris have experimented with divisions of polygonal cells using an imaginary organism (CM88).

2.3 Voronoi Diagrams

Another computationally feasible model is the *Voronoi diagram* model. This cellular model was first suggested by Hisao Honda (Hon78). It was also employed by Sulsky (Sul82). A Voronoi diagram enables the representation of a cell by a single point called the nucleus. Each nucleus has a voronoi region associated with it. The voronoi region belonging to a nucleus is the set of all points closer to this nucleus than to any other nucleus. Each voronoi region corresponds to a cell.

Voronoi regions are polygonal subdivisions of the plane. However, not all polygonal subdivisions of a space correspond to Voronoi diagrams. In fact, Voronoi diagrams form a small specific class, and even the projection of a voronoi diagram at an angle or a scaling along an axis is not Voronoi. Honda did a study on the kind of patterns occurring in nature that are voronoi (Hon78).

Sulsky developed a model for a two dimensional sheet of cells (Sul82) . The cell is chosen as the fundamental unit for the construction of a numerical algorithm, with Voronoi polygons used to represent cells. This choice of geometry makes the numerical work tractable since n cells are defined by the positions of n nuclei. The model has been successfully used to simulate cell reaggregation and sorting experiments. In addition, the embryological process of neuralation, in which a circular monolayer of cells changes to a keyhole shape, has been modeled using columnar cells.

3 The cell model in CPL

Physically, an actual cell is a solid which 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 so as to be able handle additions to the set of operations.

All the models that have been designed for cells so far, including the one CPL uses, model the cell as two dimensional. The model used by Goel et. al. (GR78) treats the cell as a rigid body of fixed size and shape. This model does not permit variable sized cells or cells with different shapes. An extension of this model, where 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. For discrete representation of cells this

The *game of life* was devised by J.H. Conway. It consists of a two dimensional array, each array position being designated a cell. Cells are either full or empty, and they can switch states depending on the number of full neighbors they have. Simple rules can lead to various periodic patterns, patterns which move in space and/or time. This model is popular because of the patterns it produces, but its biological significance is questionable.

Gordon used a simple two-dimensional grammar with the rules applied probabilistically to generate spiral patterns.

The models by Ransom, Ulam, Conway, and Gordon are all specific cases of the cellular automaton. A cellular automaton is a theoretical model of a parallel computer. It is an interconnection of identical cells, where a cell is a model of a computer with finite memory. Each cell computes an output from the input it receives from a finite set of cells forming its neighborhood, and also possibly from an external source. A cellular automaton which 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*. This system is of interest as a model for the growth and development of living things.

Lindenmayer used a one dimensional cellular automata to grow branching and non branching filaments (Lin68). These exhibit various developmental patterns, such as constant apical pattern, non-dividing apical zone, and banded patterns.

Goel et. al. (GR78; RG78; GCG+75; LG75) have conducted numerous simulations on cellular sorting and engulfment, based on Steinberg's Differential Adhesion hypothesis (Ste75). Steinberg believes that cells adhere differently to one another, and that this plays a role in development. Goel et. al. model cells as rigid objects with preassigned adhesive properties. These cells mostly form a regular square tessellation of the plane. The medium is modeled as just another cell type. Two neighboring cells are allowed to *interchange positions*, only if the interchange increases the total adhesive energy. Their model assumes differential affinities between cells of different types and local motility of cells. [2]

Antonelli et. al. (ARW75) examined simulations involving hexagonal cells, and an exchange principle that only allowed moves (which increased total adhesive energy) among nearest neighbors. Three or more cell types (one of them being the medium) were used to model tissue engulfment. They considered a hexagonal tessellation of the plane.

2.2 Polygonal Cell Models

The advantages of *Discrete cell models*, in terms of speed and ease of manipulation, are obvious. However, these models ignore the fact that cells have shape, their geometry need not remain fixed, and that cells move, while slowly changing their contacts with their neighbors. Polygonal cell models avoid these drawbacks; however, most operations on them are not so easily defined. For example, how should the surrounding polygons adjust so as to give a polygon some extra room to grow.

Matela and Fletterick (MF79; MF80; RM84; MR85) have used graphs to represent cell maps. They represent each cell by a vertex. If two cells share a common boundary, there is an edge between the two respective vertices. The graph so obtained is planar and is the dual of the cell map. In most of these simulations they restrict the graph to be triangulated,

[2] The volume by Mostow contains most of the papers on cellular sorting and engulfment (Mos75).

Turing, in his seminal paper in 1952, demonstrated that the feasibility of having stable patterns using reaction-diffusion systems (Tur52). Ouyang and Sweeney have recently discovered such patterns in chemical systems (OS91). Meinhardt presented activator-inhibitor systems for a variety of biological patterns (Mei82). Such reaction-diffusion (activator-inhibitor) systems account for a large share of the proposed models in developmental biology.

Gordon suggested a general model for development (Gor66). 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 make the following decisions: do nothing; reset its internal clock; differentiate; send messages to other cells; divide; expand or shrink; eat or fuse with a neighbor; move; and die.

The resulting interactions between the cells would be probabilistic due to thermodynamic fluctuations, inaccuracies in the division process, underconstrained or probabilistic next state function, and varying initial states. Genetic control of development is indirect in this model. The genes presumably determine the next state function, albeit implicitly.

Our programming language, CPL , is essentially based on Gordon's ideas. The primary consideration while designing CPL was that CPL should provide the ability to model cellular behavior in some generality, yet be able to simulate a significant number of cells[1].

Kurt Fleischer has designed a somewhat similar system, though it is targeted towards growing artificial neural networks (Fle93). Mjolsness et. al. have developed a connectionist model for stripe formation in Drosophilia (MSR91).

Section 2 contains a survey of the various computer models for cells. In section 3 we introduce the model CPL uses. Section 4 contains the definition of the language, including the instruction set. In section 5, we present a few examples which demonstrate the usefulness, power, and simplicity of programming biological models in CPL . Each example contains the biological theory, the CPL program which encodes the theory, and images from simulation runs. The last section, section 6, contains concluding remarks.

2 Cell Models

In the following subsections we examine the various approaches researchers have taken to model cells.

2.1 Discrete Cell Models

Theoretical biologists have been using computers to test simple models. The array is an obvious data structure; most of the early cell growth and division models revolve around it. The one dimensional filaments (which simulated algae growth) by Ransom, the abstract patterns by Ulam, Conway's game of life, and Gordon's stochastic spirals were the forerunners (Ran81). Subsequently, Lindenmayer used concepts from automata theory to devise a general theoretical model.

[1] A few thousand on a workstation.

The Cell Programming Language ‡

Pankaj Agarwal
Courant Institute of Mathematical Sciences
New York University
251 Mercer Street New York, NY 10012
agarwal@cs.nyu.edu

Abstract

We describe the Cell Programming Language (CPL), which we have designed to write programs that mimic the life of a biological cell. The aim is to study the complex interactions between cells which lead to diverse shapes of cellular aggregates.

Each cell is treated as a two-dimensional homogeneous polygon with a specific area. A cell goes through a series of states in its lifetime. In each state, one can specify the cell's growth rate; information about cell division and cell differentiation; the chemical constituents of the cell and their interactions; and cell motion. This behavior may be conditional based on the cell's own status (chemical concentrations and size) or on its neighborhood (the type of cells surrounding it, the contact lengths with each of them, their areas, the directions to these cells, and their chemical concentrations). The language is explored by modeling cellular sorting *in vitro*, and aggregation in the *Dictyostelium discoidea*.

1 Introduction

The transformation of a single cell into a complex organism is still not well understood. Most of our knowledge in developmental biology has arisen from observation and experimentation in the laboratory, but increasingly, computer models are being used to test theories developed in the laboratory. Some theories are impossible to test in the lab and others just too expensive in terms of time and money. Computer simulations present ways of providing evidence for or against theories. Often the results from these simulations can be used to devise newer experiments or they may provide biologists with clues as to alternative explanations.

Development may be described at various levels. The interactions of the various constituent molecules are important at the molecular biology level. This level is not yet amenable for computer simulation due to the vast number of such molecules and their little understood interactions. Cells are better understood, and most cells are basically similar. Cells are a level higher in the hierarchy, and thus smaller collections of cells may explain developmental behavior.

Gilbert (Gil91) and Waddington (Wad66) are good introductions to the field of Developmental Biology. Alberts et. al. (ABL$^+$89) provide detailed descriptions of cellular behavior. Ransom surveyed the field of computer modeling of cellular behavior in 1981 (Ran81). He discussed most of the prevalent computer models of development along with some of the biological motivation and underlying mathematical theory.

‡Technical Report 630, Department of Computer Science, Courant Institute of Mathematical Sciences (available by anonymous ftp from cs.nyu.edu, file /pub/agarwal/tr630.ps.Z)