Cancer Simulation

Process Information

The simulation uses Microsoft's XNA framework as backbone of application, specifically in regard to displaying graphical representation of the model. WPF back technology is used to create the functional GUI. Dynamic Data Display library is used to generate real-time plots of the simulation data.

EntryPoint

This class contains the Main function. Here the SimulationCore, Game class, is created and started. Important to note that the program is run in SingleThreadApartment (STA) mode, meaning that the program runs on a single thread. This is not ideal but a workaround to get be able to use XNA and WPF together.

SimulationCore

This class is the backbone of the application. It extends XNA's Game class, and thus gains access to Draw()which is essentially the main loop for the application. From inside of this loop I both update and draw the simulation. Simulation update is performed by calling Environment's Tick() method and the various components of the application, such as `Cells`, `Signals`, and pipes, are drawn based on their locations retrieved from the `Environment`. The view/world transformation matrix is provided by the `Camera`, which is a modified version of dhpoware's XNA 4.0 Camera Demo (http://www.dhpoware.com/demos/xnaCamera.html). The code for drawing cubes is adapted from XNA sample code. `SimulationCore` employs the

Singleton pattern, there can only be one instance of it and it can be referenced from anywhere by `SimulationCore.Instance`.

Environment

The `Environment` contains the `Cells`, `Signals`, the pressure, and the `Circulatory System`. It keeps track of the `environmentMatrix` a discretized representation of the model space. At every location in the environment matrix is a `LocationContents` object that contains all of the `Cells`, blood vessels, `Signals`, and pressure value at that particular location.

The `Environment` employs the Singleton pattern, there can only be one and it can be referenced by any object with `Environment.Instance`. The `Environment` is bounded by two 3D vectors. All points between these two vectors are initially instantiated to empty `LocationContents` and stored in `environmentMatrix`. The array data structure is used instead of a map for performance reasons; computation of hash value of Vector3 takes too long. The `Environment` is then subdivided into `Sectors` and pressure is initialized around every tissue and pipe.

`Environment` has more than one collection of `Cells` – cells, newCells, and dyingCells. This done because in C#, a collection cannot be modified while being traversed. In other words, `Cells` are created in this loop

```
foreach (TissueCell cell in cells)
        cell.Tick();
```

and the cells collection cannot have objects added or removed from it during the loop.

To solve this problem, the `Environment` includes two other collections – newCells and dyingCells. When a new `Cell` is created, it is added to the newCells collection, similarly when a `Cell` dies it is added to the dyingCells collection.

In `Environment.Tick`(), before stepping through all `Cells` in the cells collection, the program steps through newCells, adds them to cells collection, and performs appropriate actions such as adding them to the `environmentMatrix`, adding pressure at their location, and adding them to their `Sector`. Similar process occurs with dying cells.
There are also newSignals collection and expiredSignals collection, for the same reasons.

Tissues and Pressure Initialization

`Tissues` are areas in the `Environment` that are initialized with pressure of 0. This allows `Cells` placed in the `Tissues` to freely grow. The locations up to `SectorPressureDistance` away are initialized with pressure. All locations immediately outside of each `Tissue` start with `SectorPressureInitial` pressure. The locations X distance out of the `Sector` start with `SectorPressureInitial` + `SectorPressureIncrement`*X pressure. This is achieved by the following method:

First the locations on the edge of the tissue are discovered and stored in *past*, and the locations directly outside of the tissue are discovered and stored in *present*. Now, in `Environment.sendPressureWave()`, pressure is set for all locations in *present*. All immediate neighbors of every location in *present* are discovered, stored in a temporary set, and *past* is subtracted from this set. All remaining elements are +1 distance away from the tissue. This process is repeated *distance* times.

Pressure

Pressure is used as a way to both confine healthy `Cells` to `Tissues` and allow mutated `Cells` to grow outside of the tissue, effectively creating bumps around the tissue where the tumor is growing. When a `Cell` is created or moves to a particular location, the pressure at that location is increased by `PressureAtCell` and pressure in the neighboring locations is increased by `PressureNearCell`. Both of these values are stored in `SimulationParameters` object. `Cells` can only divide when pressure at their location is less than `MaxPressureToleranceAtLocation` and pressure in neighboring locations is less than `MaxPressureToleranceAtNeighbors`. `Cells` also cannot be pushed, by `PleaseMoveSignal`, into locations with pressure higher than `MaxPressureToleranceAtLocation`. These pressure tolerances are stored in `Cells' BehaviorAggregate` objects and are mutateable, and should be mutated to achieve high-density `Cell` growth.

Sector

The `Environment` is subdivided into `Sectors` of `SectorWidth` x `SectorHeight` x `SectorDepth` dimensions. The `Sectors` are bounded by two `Vector3s`. Their main purpose is to keep track of the number of `Cells` and blood vessel value within the `Sector`. These values are used to compute `CellPipeRatio`, a value used by `Cells` to determine if there is enough food in their `Sector`.

Cells

There are two kinds of `Cells` in this simulation – `BlastCells` and `FinalCells`. `BlastCells` represent stem cells. They can divide an infinite number of times and can divide into both `BlastCells` and `FinalCells`. The division is random and the probabilities for dividing into each can be configured in `BlastCell.initializeBlastCell()`. `FinalCells` represent fully differentiated cells and can only divide `FinalCellGenerationLimit` times. Every time that a `FinalCell` divides, its generation count is updated; when this limit is reached the cell dies. Because of this mechanism, `BlastCells` are vital to the successful growth of a cell family.

PleaseMove Signals

When a `Cell` divides, it new cell is created at the parent's location. The parent then sends a `PleaseMove` signal to move the child away. During a `Cell's` Tick, it checks if there are any `PleaseMove` signals at its location. If there are, the cell moves to a randomly chosen location that does not contain have a `PleaseMove` signal.

Cells and Food

`TissueCells` require nourishment for growth that is naturally excreted from blood vessels. When there is not enough food in the area, the `Cell` begins to starve and if it has the ability, requests for additional blood vessels to be grown. In the simulation, this process is modeled by the ratio of `TissueCells` to `EndothelialCells` in a given sector. When this ratio is in the `TissueCells'` favor, they increase their food supply, otherwise, the food supply gets depleted. When the food supply drops to `FoodConcernLevel`, if it is able to, requests additional blood vessels (angiogenesis).

Each `Cell` keeps track of the amount of food that it currently has, and has the mutateable properties of

`foodConsumptionRate` – rate at which the stored food is consumed.
`maximumStoredFood` – the limit on the cell's food storage.
`foodConcernLevel` – the point below which the food supply must drop for the cell to request additional blood vessels.

When a `Cell` requests blood vessels, it puts itself on a globally accessible collection of hungry `Cells` in the `CirculationSystem`. At each `CirculationSystem` tick, the collection of hungry `Cells` is traversed, the closest blood vessel to the hungry `Cell` is located, and begins to grow. If the `Cell's` food supply ever drops to 0, it dies.

CirculatorySystem

This is a major component of the simulation. It provides food for the nearby cells which allows them to grow, and provides a transportation network between the tissues.

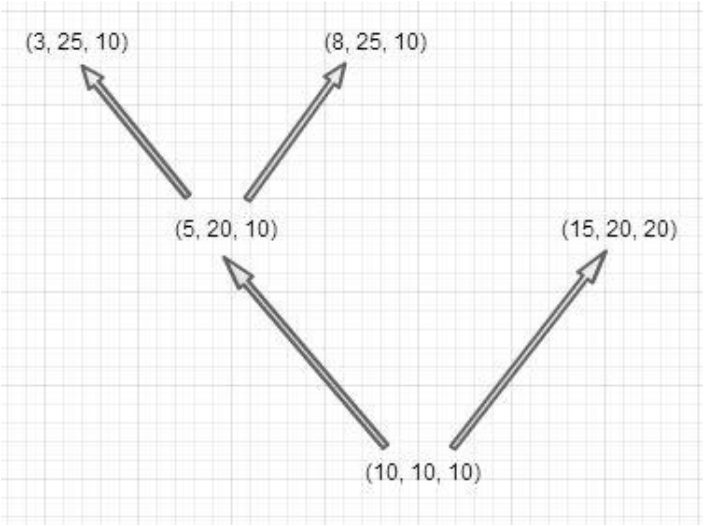
Pipe is used interchangeably with `EndothelialCell`. At every Tick the `CirculatorySystem` grows `maxNumGrowthsPerTurn`, these are randomly distributed throughout needy Cells.

The initial pipe locations are defined by `PipeParameters` object, which is a recursively defined, tree-like, structure. It has a simple definition:

```
class PipeParameters
{
	Vector3 location;
	List<PipeParameters> children;
}
```

Each object essentially defines a corner of the pipe by its location and children corners that grow from it, for example, the following JSON representation of the object and the tree diagram are equivalent.

```
{
   "location":"10,10,10",
   "children":[
      {
         location":"5,20,10",
         "children":[
            {
               "location":"3,25,10",
               "children":[]
            },
            {
               "location":"8,25,10",
               "children":[]
            }
         ]
      },
      {
         "location":"15,20,20",
         "children":[]
      }
   ]
}
```



The outermost object is considered the root, and ultimate parent in the tree, it is constructed first and the rest of the tree is constructed recursively.

```
setupPipes(EndothelialCell currPos, List<SimulationParameters.PipeParameters> children)
{
        if (children == null || children.Count == 0)
                return;

        foreach (SimulationParameters.PipeParameters child in children)
        {
                EndothelialCell newestPiece = createPipe(currPos, child.location);
                setupPipes(newestPiece, child.children);
        }
}

EndothelialCell createPipe(EndothelialCell from, Vector3 target)
{
        EndothelialCell currCell = from;
```

```
        Vector3 currLoc = currCell.CellLocation;
        float size = from.size;

        while (currLoc != target)
        {
                size *= pipeShrinkRate;
                pipeInfo = EndothelialCell.getGrowthDirection(currLoc, target);
                currLoc = currLoc + pipeInfo.Direction;
                orientation = pipeInfo.Orientation
                EndothelialCell newPipePiece = new EndothelialCell(currLoc, orientation, size,
        currCell);
                currCell.Children.Add(newPipePiece);
                currCell = newPipePiece;
        }

        return currCell;
}
```

Finding Closest Pipe to Hungry Cell

The slow way of finding the closet pipe to the hungry `TissueCell` is to step through every `EndothelialCell` and find smallest distance to the target. This is inefficient. As optimization, `TissueCells` keep track of the closest pipe to them, and this information is passed down to children `Cells`. When looking for the closest pipe to the hungry `Cell`, local search method is used, starting at the "guess," the stored location.

```
private EndothelialCell pipeSearch(TissueCell target, EndothelialCell startingLocation)
{
        Vector3 targetLocation = target.CellLocation;
    int currDistance = CityBlockDistance(startingLocation.CellLocation, targetLocation);

        EndothelialCell parent = pipeSearchParent(target, startingLocation, currDistance);
        int pDistance = CityBlockDistance(parent.CellLocation, targetLocation);

        EndothelialCell child = pipeSearchChildren(target, startingLocation, currDistance);
        int cDistance = CityBlockDistance(child.CellLocation, targetLocation);

        if (pDistance < cDistance)
                return parent;
        return child;
}
```

`pipeSearchParent()` steps through pipes, starting from the startingLocation, in direction of the parent looking for the smallest distance between the current location and the target. Once the distance increases, algorithm assumes that search is heading in wrong direction and the best location is returned. This is then compared with a similar search made through the children direction.

The first generation of `Cells` do not have a closest pipe value set, so the slow method of stepping through the whole pipe tree is used. All subsequently created `Cells` have a guess of where the closest pipe section is, which is confirmed by local search.

After the closest pipe section to the cell is found, pipe grows one step towards the hungry `Cell`, and that `Cell's` closest pipe section is updated to contain the value of the newly created `EndothelialCell`.
When `EndothelialCell` is created, the cellPipeRatio in its sector is updated.

Traveling Through Pipes

`TissueCells` contain `PipeTravelInfo` objects. These objects contain the `TravelDirection`, either to parent or to children, and a reference to the containing `EndothelialCell`. When this object reference is set to null, the assumption is that the cell is not currently traveling through pipe. When this is set to a value other than null, the assumption is that the cell is indeed traveling through a pipe.

When a Cell moves into a location containing a pipe, a roll is made against the Cell's EnterPipeProbability. If roll passes, then check is made if pipe is large enough to hold Cell, based on MinPipeSizeForTravel parameter. If the EntothelialCell is suitable for Cell's travel, its PipeTravelInfo reference is set to a new instance of the class and a travel location is chosen randomly, either towards parent or children. The Cell's containing Sector's TissueCell count is decremented, the Cell is removed from the location in the environmentMatrix, and the pressure at Cell's location and neighboring locations is reduced. At the Cell's next Tick, because PipeTravelInfo is not null, the Cell will behave differently and travel through the pipe.

```
TissueCell.Tick()
{
        if(pipeTravelInfo == null)
                act like a tissue cell
        else
                travel through the pipe
}
```

While in the pipe, a Cell might die, depending on its SurivePipeProbability, or it might leave the pipe either randomly based on LeavePipeProbability or if the pipe gets too small, based on MinPipeSizeForTravel. Easiest way to guarantee that Cells will move between Tissues is to have the Cell exit when it gets to the end of a pipe, that is, when the EndothelialCell it is in has no children.

When a Cell does leave a pipe, it is deposited in a randomly selected neighboring Cell that doesn't have a pipe in it. It is added to the environmentMatrix, its Sector is rediscovered, the Cell count is updated, and the pressure at the Cell's new location and neighboring locations is increased.

Cell Behaviors

Each `Cell` has a `BehaviorAggregate` object. This object contains all of the mutateable values and probabilities that define the `Cells'` behavior.

```
divideProbability
deathProbability
moveProbability
pipeEnterProbability
pipeLeaveProbability
pipeSurvivalProbability
pipeRequestProbability
pressureToleranceAtLocation
pressureToleranceAtNeighbors
foodConsumptionRate
foodMaxStorage
foodConcernLevel
```

When the first `Cells` are created at the beginning of the simulation, their `BehaviorAggregate` objects are populated with values from the `SimulationParameters` object. `BehaviorAggregate` objects are designed so that `SimpleBehaviors`, containing just values not backed by anything, and `GOBehaviors`, values backed by `GOTerm` data, can be interchanged.