Multiscale modeling

Simple Grain Growth Cellular Automata and Monte Carlo

All my code can by found on Git repository at https://github.com/vonProteus/MultiscaleModelling-SimpleGrainGrowthCA

My code

The main idea of the cellular automata technique is to divide the material into lattices of finite cells, where cells have clearly defined interaction rules between each other. Each cell in this space is called a cellular automaton, while the lattice of the cells is known as cellular automata space.

Cell

In my code the cell is *MKCell* which is very simple construct that looks like

```
//
//  MKCell.h
//  Simple Grain Growth CA
//
//  Created by vonProteus on 2014-03-15.
//  Copyright (c) 2014 Photep. All rights reserved.
//

#import <Foundation/Foundation.h>

@interface MKCell : NSObject

@property (readwrite) NSInteger grainId;
@property (readwrite) BOOL isOnBorder, isLiving, willGrow, wasChanged,
wasRecristalized;
@property (readwrite) NSInteger coordinateX, coordinateY;
@property (readwrite) CGFloat energy;

- (id)init;
- (MKCell*)getAllFrom:(MKCell*)hear;
- (void)clear;

@end
```

As you can see it is a very simple structure that contains *grainID* and some internal helpful variables like *isOnBorder*, *wasChanged*

Automat Space

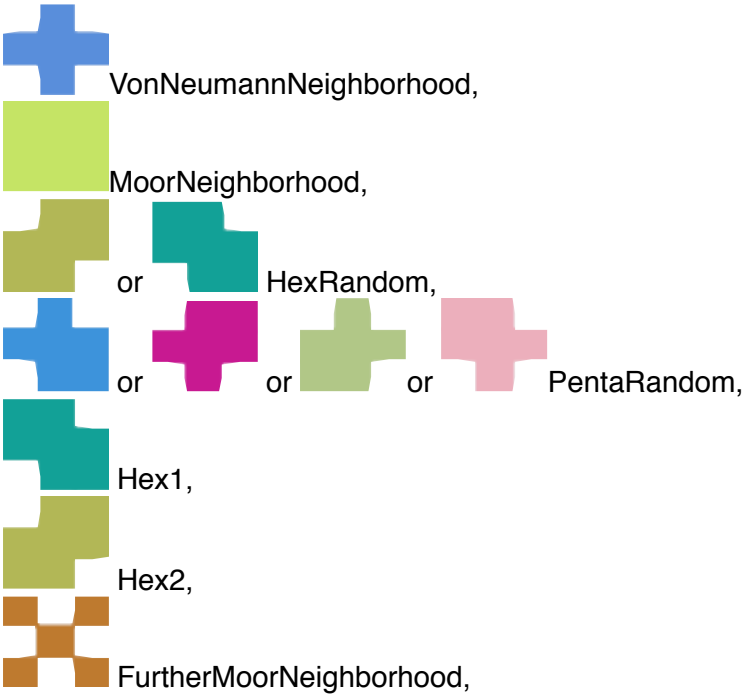
At the core of my code is a class named *MKAutomat*. It is used for Cellular Automata and Monte Carlo depending on internal variables and gov that are set. The header of this class looks like

```
//
//  MKAutomat.h
//  Simple Grain Growth CA
//
//  Created by vonProteus on 2014-03-15.
//  Copyright (c) 2014 Photep. All rights reserved.
//

#import <Foundation/Foundation.h>
#import "MKCell.h"
#import "MKEnums.h"

@interface MKAutomat : NSObject {
    NSInteger x, y, lastId;
    NSArray* ca;
    NSArray* caPrev;
    enum BoundaryTypes boundaryType;
    enum NeighborsTypes neighborsType;
    enum TransitionRules transitionRules;
    enum EnergyDystrybution energyDystrybution;
    MKCell* absorbingCell;
}

@property (readonly) NSInteger x, y, lastId;
@property (readwrite) enum BoundaryTypes boundaryType;
@property (readwrite) enum NeighborsTypes neighborsType;
@property (readwrite) enum TransitionRules transitionRules;
@property (readwrite) enum EnergyDystrybution energyDystrybution;

- (id)init;
- (id)initWithX:(NSInteger)X Y:(NSInteger)Y;

- (NSInteger)andrzej;
- (MKCell*)getX:(NSInteger)X Y:(NSInteger)Y;

- (NSInteger)changeGrainID:(NSInteger)gid toNewGrainID:(NSInteger)newGrainId;

- (NSInteger)addNewGrainAtX:(NSInteger)X Y:(NSInteger)Y;
- (bool)addNewDislocationAtX:(NSInteger)X Y:(NSInteger)Y WithR:(NSInteger)R;
- (bool)addNewDislocationAtX:(NSInteger)X Y:(NSInteger)Y WithD:(NSInteger)D;
- (NSInteger)saveGrainAtX:(NSInteger)X Y:(NSInteger)Y;
- (void)clear:(NSSet*)grainToSaveOrNil;

- (NSInteger)sizeOfGrainWithId:(NSInteger)grainId;
- (CGFloat)energyOfGrainWithId:(NSInteger)grainId;
- (CGFloat)maxEnergy;
- (CGFloat)minEnergy;

- (void)addEnergyForGrain:(CGFloat)energyForGrain;
@end
```

The main function that performs CA or MC iteration is named *andrzej* and it does one iteration and returns the number of changed cells

Neighborhood

Neighborhood Types are very important things for CA. The results are determined by this variable in my code. I have seven types of neighborhoods, they look like this after one *andrzej*



Boundary Conditions

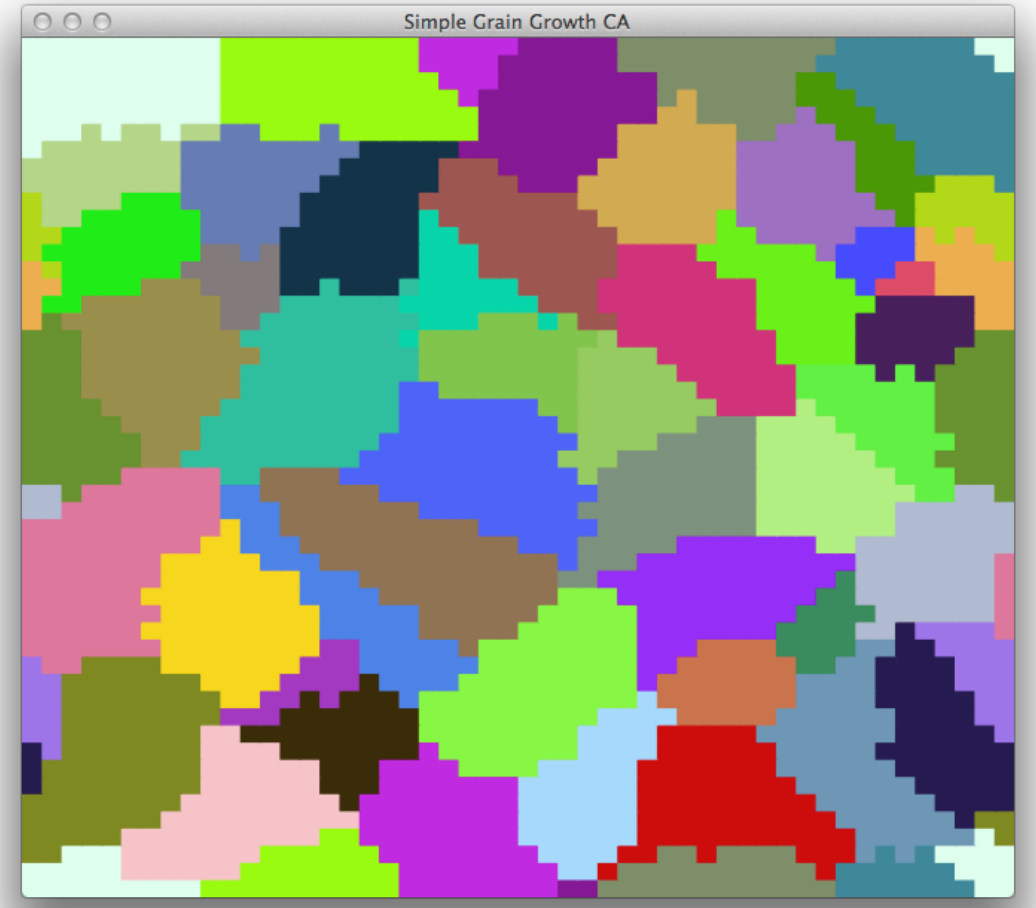
In my code there are only two boundary conditions used. Periodic boundary conditions that don't lose any material and represent a torus, and absorbing boundary conditions that remove material which want to go out of the computation domain.

9	7	8	9	7
3	1	2	3	1
6	4	5	6	4
9	7	8	9	7
3	1	2	3	1

This is an example of space with periodic boundary condition green are real cells and yellow are virtual cells provided by boundary condition

Transition rules

I have two types of transition rules and the Montecarlo mode. This is "rule one" and its results look like this after full growth.

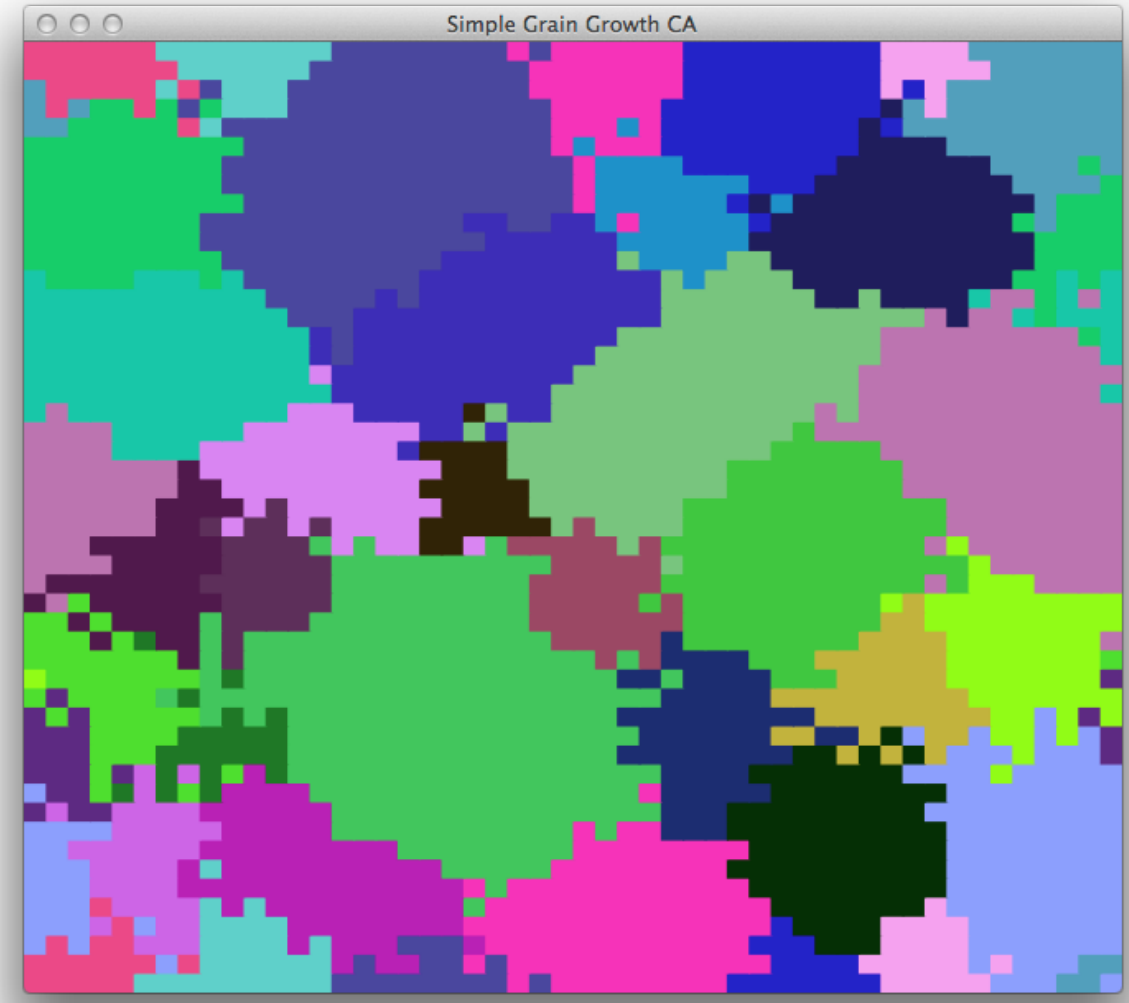


I start in 50x50 space with 50 randomly distributed nucleons using rule 1 and hex random neighborhood

Code for Rule one looks like

```
if (currentCell.isLiving) {
    continue;
}
NSSet* neighbors = [self getAllNeighborsWhoCanGrowForX:b
                                                  andY:a];
NSMutableArray* neighborsIds = [NSMutableArray array];
for (MKCell* neighbor in neighbors) {
    if (neighbor.grainId > 0) {
        [neighborsIds addObject:[NSNumber
numberWithInteger:neighbor.grainId]];
    }
}

if (neighborsIds.count > 0) {
    currentCell.grainId = [[neighborsIds objectAtIndex:arc4random() %
neighborsIds.count] intValue];
    currentCell.isLiving = YES;
    ++changes;
}
```

As we can see it is a very simple rule. If a cell is filled with grain in the neighbourhood, the examined cell's Id will change to that cell's Id"



Here we can see a different result using the same starting conditions (nucleons differently placed because it is random) but with transition rules 1-4

Rules 1-4 are more complicated but they give more realistic results and they don't stop when space is completely filled with grains.

Code for rules 1-4 looks like

```
if (currentCell.grainId != -1 && currentCell.isOnBorder == YES) {
    if ([self rule1On:currentCell]) {
        ++changes;
    } else if ([self rule2On:currentCell]) {
        ++changes;
    } else if ([selfish rule3On:currentCell]) {
        ++changes;
    } else if ([self rule4On:currentCell]) {
        ++changes;
    }
}
```

As you can see it calls four rules. First three rules call the same function with different parameters. - *(BOOL)genericRuleForNeighbors:(enum NeighborsTypes)neighborhood*

minimumOfNeighborers:(NSInteger)min onCell:(MKCell)currentCell* and look like this

```
- (BOOL)genericRuleForNeighbors:(enum NeighborsTypes)neighborhood
minimumOfNeighborers:(NSInteger)min onCell:(MKCell*)currentCell
{
    neighborsType = neighborhood;
    NSSet* neighbors = [self
getAllNeighborsWhoCanGrowForX:currentCell.coordinateX

andY:currentCell.coordinateY];
    NSArray* count = [self getStatsFor:neighbors];

    MKAns* bestAns = nil;
    NSInteger max = 0;

    for (MKAns* ans in count) {
        if (ans.greinId > 0) {
            if (max < ans.count) {
                bestAns = ans;
            }
        }
    }

    if (bestAns != nil) {
        if (bestAns.count >= min) {
            currentCell.grainId = bestAns.greinId;
            currentCell.isLiving = YES;
            return YES;
        }
    }
    return NO;
}
```

First rule uses *MoorNeighborhood* and requires 5 neighbors. The second rule uses *VonNeunmannNeighborhood* and requires 3 neighbors. The third rule uses *FurtherMoorNeighborhood* also requires 3 neighbors.

Ffourth rule is necessary for initial growth and is very similar to my rule1 and looks like this

```
- (BOOL)rule4On:(MKCell*)currentCell
{
    neighborsType = MoorNeighborhood;
    NSSet* neighbors = [self
getAllNeighborsWhoCanGrowForX:currentCell.coordinateX
                          andY:currentCell.coordinateY];

    NSMutableArray* neighborsIds = [NSMutableArray array];
    for (MKCell* neighbor in neighbors) {
        if (neighbor.grainId > 0) {
            [neighborsIds addObject:[NSNumber
numberWithInteger:neighbor.grainId]];
        }
    }

    if (neighborsIds.count > 0) {
        currentCell.grainId = [[neighborsIds objectAtIndex:arc4random() %
neighborsIds.count] intValue];
        currentCell.isLiving = YES;
        return YES;
    }
    return NO;
}
```

Montecarlo

The Monte Carl method is very interesting because it uses random numbers to meaningful and realistic results.

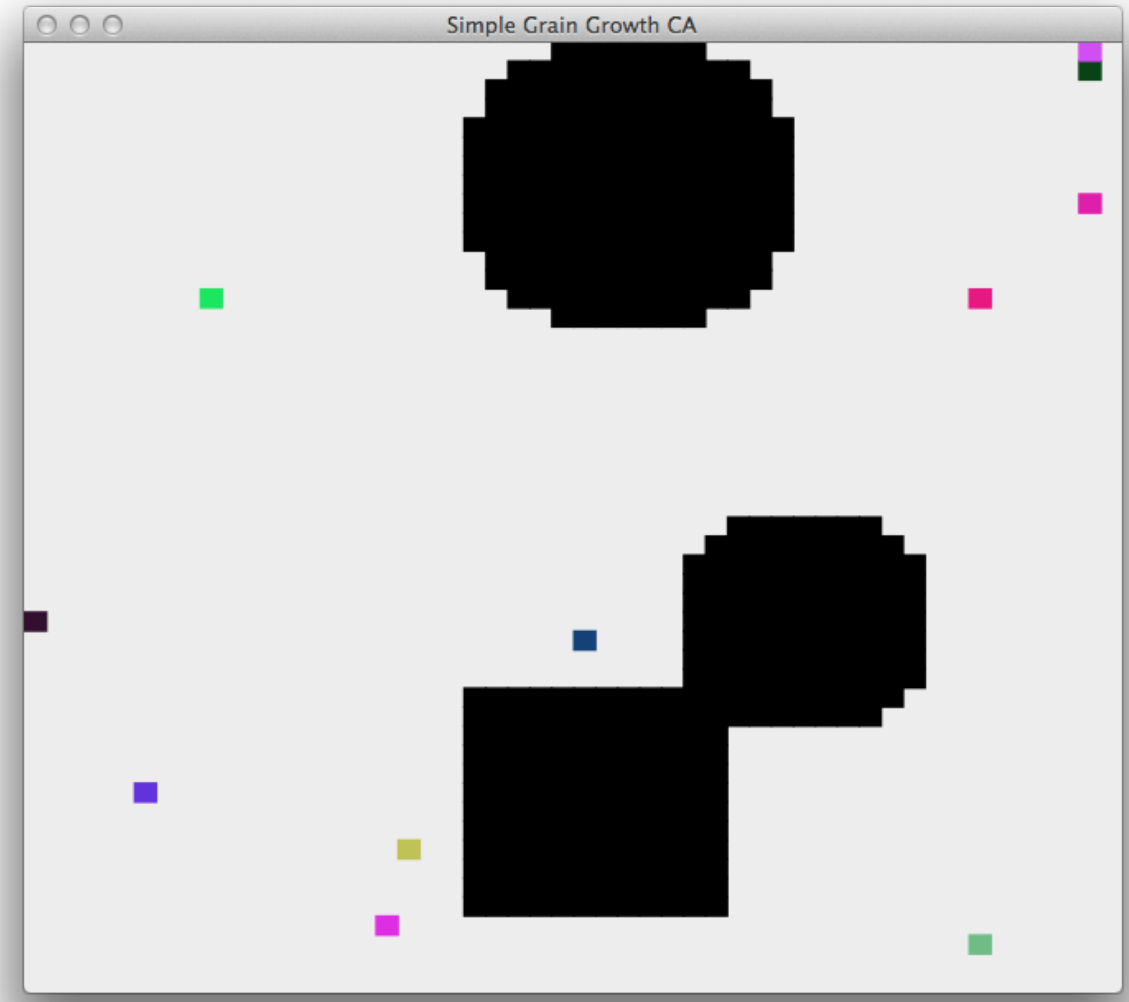
Iteration of Montecarlo looks like

```
self.neighborsType = MoorNeighborhood;
NSMutableArray* toGo = [NSMutableArray array];
for (NSInteger a = 0; a < y; ++a) {
    for (NSInteger b = 0; b < x; ++b) {
        MKCell* currentCell = [self getX:b
                                       Y:a];
        if (currentCell.isOnBorder) {
            [toGo addObject:currentCell];
        }
    }
}

while ([toGo count] > 0) {
    MKCell* cell = [toGo objectAtIndex:arc4random() % [toGo count]];
    if (cell.grainId > 0) {

        NSSet* neighbors = [self
getAllNeighborsWhoCanGrowForX:cell.coordinateX
                         andY:cell.coordinateY];
        CGFloat energy = [neighbors count];
        CGFloat newEnergy = [neighbors count];

        MKCell* newCell = [[neighbors allObjects] objectAtIndex:arc4random() %
[neighbors count]];
        NSInteger newId = newCell.grainId;
        for (MKCell* neighbor in neighbors) {
            if (neighbor.grainId == cell.grainId) {
                energy -= 1.0;
            }
            if (neighbor.grainId == newId) {
                newEnergy -= 1.0;
            }
        }
        if (newEnergy <= energy) {
           cell.grainId = newId;
            [self getPrevX:cell.coordinateX
                         Y:cell.coordinateY].grainId = newId;
        }
    }
    [toGo removeObject:cell];
}
```

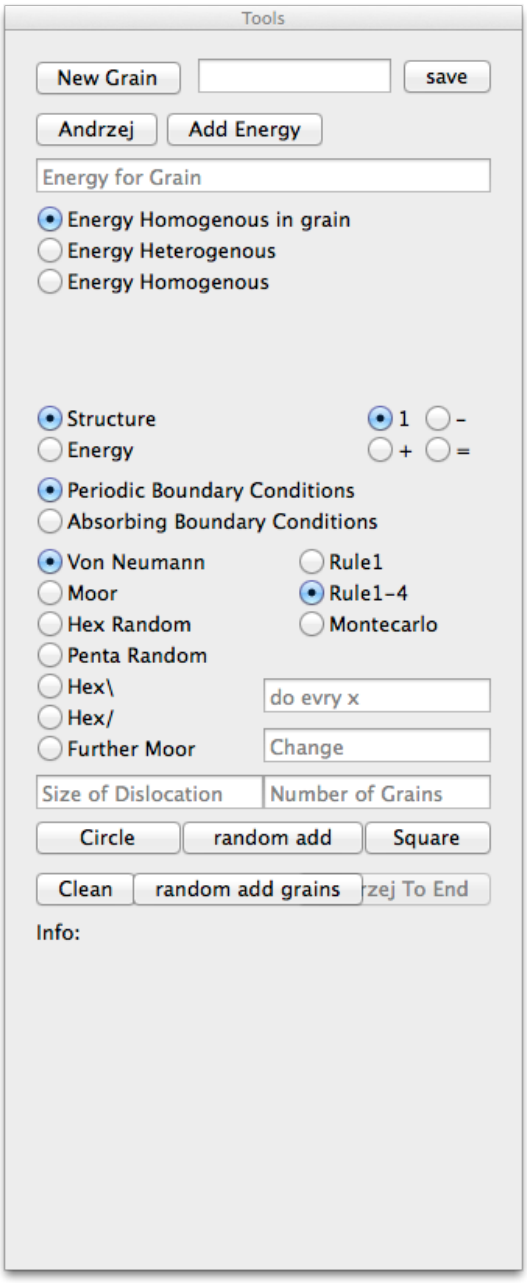
In the first block we only get cells that are on the border. This considerably reduces time of computation.

Then we randomly get one cell from the border and start counting energy of a cell before change and after. If the energy after change is lower then we commit the change, otherwise we do not change the cell

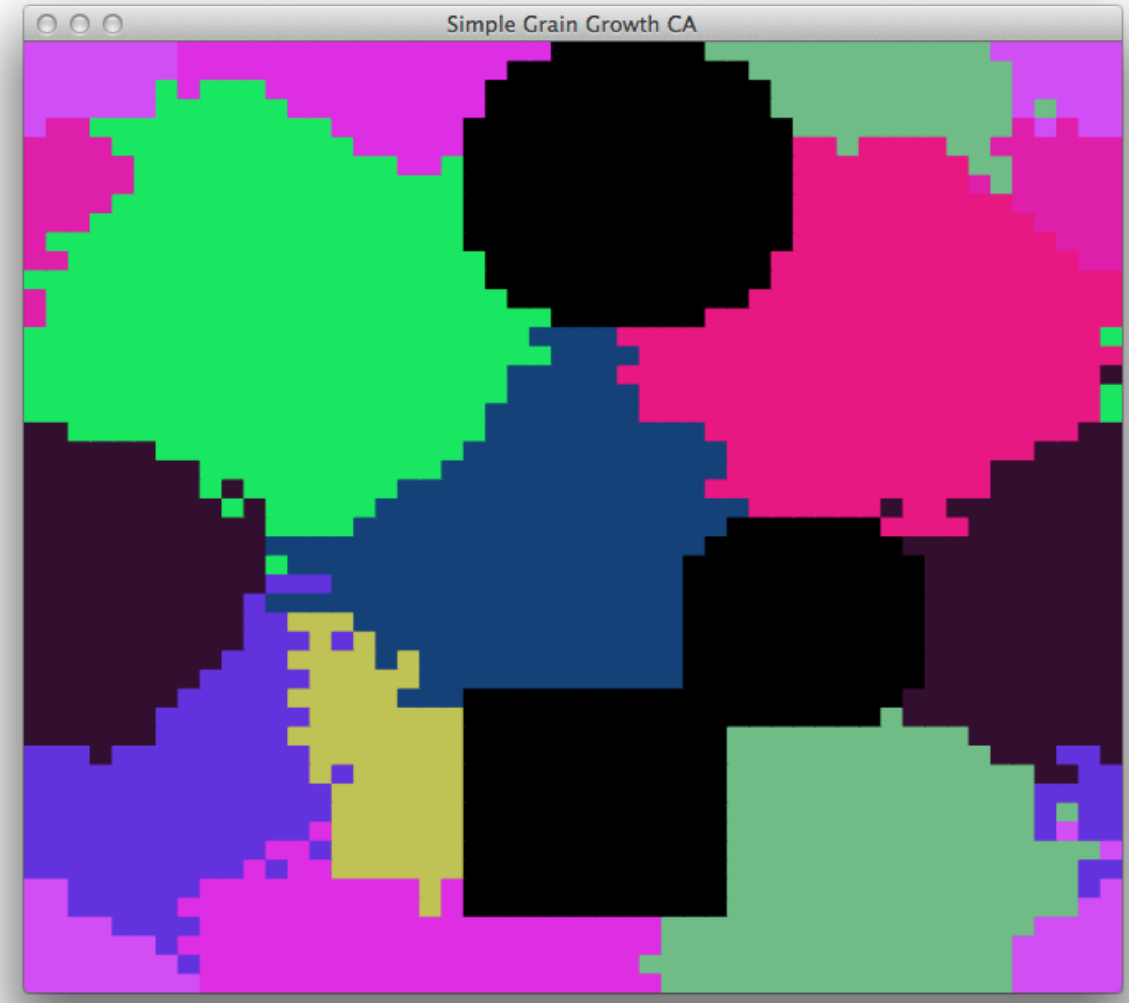
Capabilities software



This is how software looks like after launch. As we can see we have some grains nucleons and some inclusions (black). Now we will run simulation with com parameters for example HexRandom neighborhood Rule1-4 and periodic boundary condition



Here we can see (I will describe only elements relevant to this project):
button to add new grain (by clicking in main window)
button to save grain (by clicking in main window)
button to execute one iteration
choose boundary condition
choose neighborhood type
chose transition rule
add dislocation with size randomly or by clicking
clean structure (with saving grains that were selected to save)
add some random grains



After several iteration we can sea how structure looks.
and lets save some grains lets say with ids10, 8 and inclusions with -1 and start the Montecarlo
this is how structure looks when we start Montecarlo

after few steps

and after a lot of steps


Simple Grain Growth CA

Conclusions

My conclusions after writing this code

- Montecarlo method is quite interesting with stochastic behavior
- CA is much more fun to implement ^_^
- My code has high requirements but it has not been optimized
- I can see some places when we can use GPU
- It should be fairly easy to parallelize
- In CA I get the best results with the hex random neighborhood
- Montecarlo in my code gives better results than CA in my opinion because grains are more grainish
- The first Montecarlo iteration takes the most time c each cell is on a boundary