Leaf Vein Formation

New Mexico
Supercomputing Challenge
Final Report
April 2, 2008
Team 6
Albuquerque Academy

Team Members:
Brandon Oselio
Vlado Ovtcharov
Teacher:
Jim Mims
Project Mentor:
Jim Mims

Index:

Summary	-pg1
Problem Statement	-pg1
Mathematical Model	-pg1
Computational Model	-pg2
Conclusion	-pg4
References	-pg5
Appendix A simulations:	-pg5
Appendix B code:	-pg9

Summary:
Our project is intended to model leaf vein formation using given controlling and limiting factors. We used an auxin flow model for our computer simulation of the leaf. Auxin is an enzyme that controls cell growth. Auxin flow, or canalization, is controlled by placement of sinks and producers. The producers are special cells that create auxin, and sinks are where auxin is depleted. A common sink in a leaf is the stem. Using our simulation, we can see the auxin flow given various sinks and producers and therefore simulate the forming of leaf veins. We are also ignoring environmental factors such as wind that would have an effect on the growth of the leaf veins. There are other types of models, but auxin canalization is the model that has biological influence. The two others we considered were tensorial stress fields, which deals with mechanical stress on the leaf, and Turing fields, which is a purely mathematical based model.
Problem Statement:
We are attempting to simulate leaf vein formation using the auxin canalization model and controlling auxin sinks and producers.
Mathematical model:
For our model we assume that auxin can flow via diffusion or facilitated diffusion. Diffusion is just the movement of auxin from a place of high concentration to a place with low concentration. Facilitated diffusion is transporters carrying auxin along the concentration gradient.
Variables
Φ is the flux of auxin across that cell wall.
D is the facilitated diffusion coefficient.
D is the diffusion coefficient.
Ci is the concentration of auxin in cell i.
Cj is the concentration of auxin in cell j.
α a constant, production of carriers based on flux

β a constant, background production of carriers
γ a constant, decay rate of carriers
Initial equations proposed by Mitchison, 1980
To measure flux using facilitated diffusion:
$\Phi = (F)(C\ i - C\ j)$
To calculate facilitated diffusion coefficient:
$\frac{dF}{dt} = \alpha (\Phi^2) + \beta - \gamma F$
After introducing background diffusion
substitute
$D' + D = F$
Making the flux using facilitated diffusion and background diffusion:
$\Phi = (D' + D)(C\ i - C\ j)$

also
$\frac{dF}{dt} = \frac{dD'}{dt} + \frac{dD}{dt}$, but D is a constant so it's derivative is 0 therefore
Therefore
$\frac{dD'}{dt} = \alpha (\Phi^2) + \beta - \gamma D - \gamma D'$
then substitute the constant $\beta - \gamma D$ for β' to get
facilitated diffusion coefficient with background diffusion:
$\frac{dD'}{dt} = \alpha (\Phi^2) + \beta' - \gamma D'$

We then use these equations to calculate the concentration change using the following equation
Variable
C is the concentration of auxin in the cell
σ is the auxin producing capability of the cell (most cells don't produce auxin so this is 0 for most)
$\sum \Phi$ is the sum of the flux across all the walls of the cells
$\frac{dC}{dt} = \sigma + \sum \Phi$
We then make a discrete form of the differential equations used to calculate the facilitated Diffusion coefficients.
$\Delta D' \approx \Delta t(\alpha (\Phi^2) + \beta' - \gamma D')$ as long as Δt is relatively small
this makes the change in concentration
$\Delta C \approx \Delta t(\sigma + \sum \Phi)$

Computational Model:

Basics:

For the computational model we set up a grid of leaf cells, using a two-D array. The color of the cell indicates how much auxin it has. Green indicates low concentration as where orange indicates high concentration. Each cell also has four walls, top, bottom, left, and right. For each of these walls we store the flux and facilitated diffusion coefficient. A high positive flux is indicated by green and large negative flux is indicated by red, and low magnitude of flux is determined by how much of these colors it has, so an almost black wall would have almost no flux. Each cell also has a neighbor for each wall, that auxin can flow to (except for the border cells).

Cells that reach the maximum facilitated diffusion coefficient value are represented by white colored cells, and represent leaf veins.

Initial:

We experiment giving each of the constants a different initial value but the standard set up is

```
number_of_cells_wide=30;
number_of_cells_tall=30;
step=.01;
D=.325;
alpha=.00005;
```

beta=.005;
gamma=.05;
beta_prime=beta-gamma*D;
T=D;
F=15; (how much auxin flow into the top row of cells every step)
max_D_prime= 10; (how large the facilitated diffusion coefficient has to be for a vein to turn into a vein)
Also we have a constant inflow of auxin T at the top of the cell and the bottom cells act as a sink (always have 0 concentration).
At the start to calculate concentration we place more auxin at the top and in a liner fashion decrease the amount of auxin in each cell the further down till it reaches 0 at the sink cells.

Calculation:

In order to calculate the concentration changes we first calculate the flux of auxin through each cell wall. In order to optimize this we actually on calculate every other wall, knowing that two walls that are adjacent will have the same flux, but one will be positive one the other will act as negative for their respective cells.
After the flux is calculated we then calculate the diffusion coefficient. We then calculate the concentration changes. (go to appendix b for code)

Analysis:

By looking at the sample runs (appendix a) we learned several things about how the variables affect the system, and also saw some characteristics of vein formation such as vein loops, spontaneous veins (veins forming without an auxin source by them), and different order veins.
We realized that if we increased alpha (figure 4), production of carriers based on flux, the entire leaf began differentiating into veins making the entire leaf a large vein. This obviously would not be good for a leaf. However if we decreased alpha to much (figure 5) leaf vein would stun completely and veins would not form. This would mean that the leaf would not be able to receive any water and would therefore die. The biological reasons for this effect is that if there are to many carriers than the auxin moves around rapidly increasing the flux, which increases the production of carriers, which goes into a self feeding loop. If it's to small how ever the production of carriers is overpowered by the decay and therefore not enough of them can form to move the auxin fast enough.
Increasing and decreasing gamma, decay rate of carriers, has the inverse effect, if it's increased the carriers decay to quickly and no veins are formed if to little the carriers stay around for to long and buildup turning the entire leaf into a vein, and has the same biological idea behind it.
Increasing the maximum amount the facilitated diffusion coefficient can be, and therefore raising the amount the facilitated diffusion coefficient has to be to turn into a vein had similar effects as raising gamma. But instead of spreading through the entire leaf like gamma and alpha did raising the max seemed much more localized to where veins were already forming, instead it just made the veins thicker or thinner. This could be useful in nature for plants in different environments requiring different kinds of veins. Ones that are abundant in water probably will not need as thick veins as the water in the air will help the leaf, as in places where there is not abundance of water it might be beneficial to

have slightly thicker veins because more water will need to be transported to the leaf to compensate for they dry air (although a lot of other factors go into determining what kind of leaf vein sizes are required, but in any case this is a simple way for the leaf to control this).
The most interesting effect we saw was when we changed beta, background production of carriers. When beta was too large it was similar to increasing alpha and the entire leaf turned into a vein, but by decreasing beta we got some interesting results. Veins formed spontaneously, where no auxin producers or sinks were placed, and were

completely detached from the primary vein. This kind of behavior has been seen in leas with mutated genes, and has been used as evidence against canalization, because I doesn't make much intuitive sense. But in our simulation we see how these can form. By looking at the steps it looks like the primary veins goes first from the auxin source placed at the top and then flows down to the sinks, then it begins to branch out on the top to the left and right (like in figure 1) this however begins to actually create a large enough concentration difference for the facilitated diffusion coefficient to increase. Normally beta acts as an evening out variable since it increases the diffusion coefficient everywhere regardless of the concentration, but by lowering it down, it does not have enough influence to even out the auxin and therefore spikes of flux start to appear (figure 7) which eventually turn into leaf veins.
It was also believed that facilitated diffusion could not explain loop formation, and lower order veins, but when we expanded the size of our simulation we found both of these features (figure 9). Leaf vein orders, is simply how large a vein is, with the largest vein (primary vein) usually running down the middle. And secondary veins branching off. We got a similar result in this simulation with a thick primary vein extending from the tip to the base, and also is thicker at the base then at the tip, like a real leaf. We also had secondary veins branching off of it, and many lower order veins below it. Several of the lower order veins even formed loops. We believe that if we had an even more powerful computer and a more accurate shape of a leaf we can use this program to accurately model life size leafs.

Conclusion:

We were able to make a program that accurately models how auxin flows and can be used to see where leaf veins can form. We have also shown that through just facilitated diffusion vein loops can form, and also spontaneous veins (one's not connected to an auxin source) can form. These were the two largest arguments against the canalization diffusion theory but in our program we show that under the correct parameters this phenomenon can occur. We have also been able to some degree model different vein orders (if you simply take the width of the veins as a vein order) showing

that diffusion can not only explain how the primary veins form but also lower order veins.

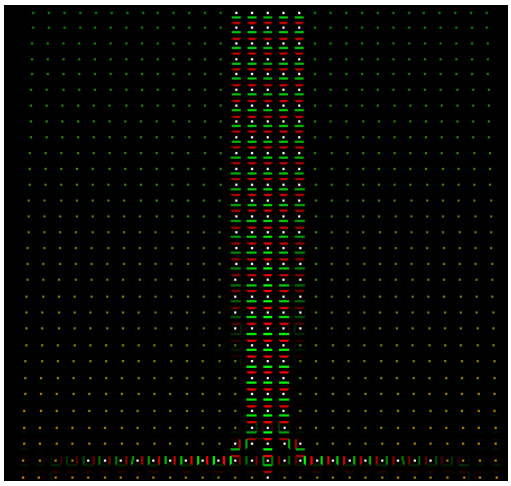
References

Plant Function and Structure by Victor A. Greulach, The Macmillian Company, 1973
Plant Physiology by Frank B. Salisbury and Cleon W. Ross, Wadsworth Publishing Compnay, 1969
Websites
http://homepages.uni-tuebingen.de/anita.roth/AnnBot-2001.pdf
http://www.lps.ens.fr/~adda/papiers/EPJB02.pdf
http://www.esf.edu/efb/course/EFB530/lectures/waterxpo.htm
http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=419854
http://www.blackwell-synergy.com/doi/abs/10.1111/j.1365-313X.2005.02581.x

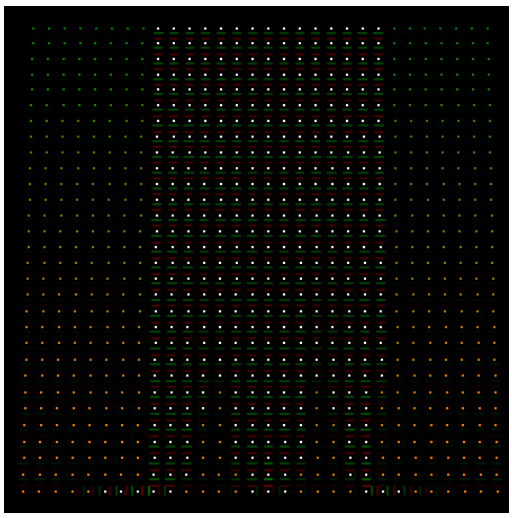
Appendix A: Sample Runs

number_of_cells_wide=30;
number_of_cells_tall=30;
step=.01;
D=.325;
alpha=.00005;
beta=.005;
gamma=.05;
beta_prime=beta-gamma*D;
T=D;
F=15;

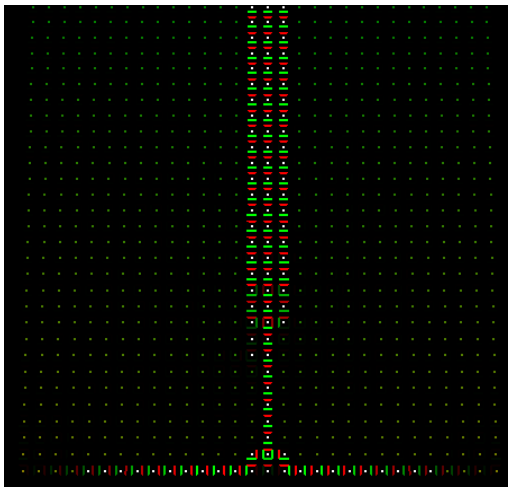
max_D_prime= 10; (100,000 steps) figure 1 | max_D_prime= 1; (100,000 steps) figure 2



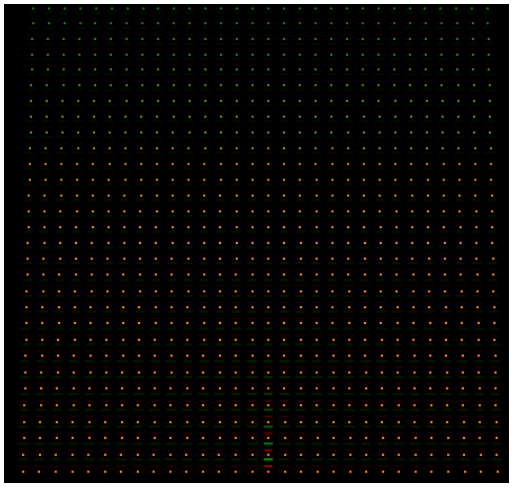
max_D_prime= 30; (100,000 steps) figure 3 |



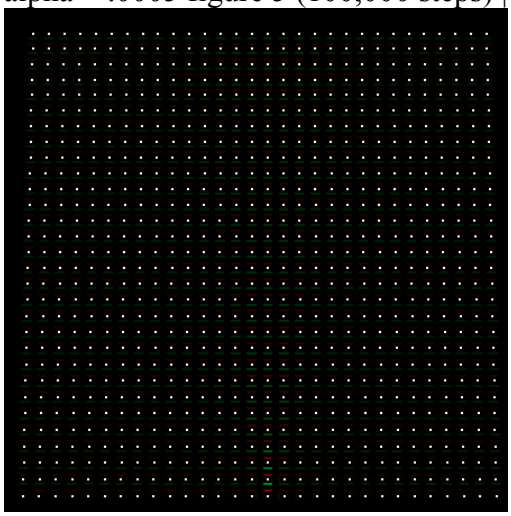
alpha = .000005 figure 4 (100,000 steps)

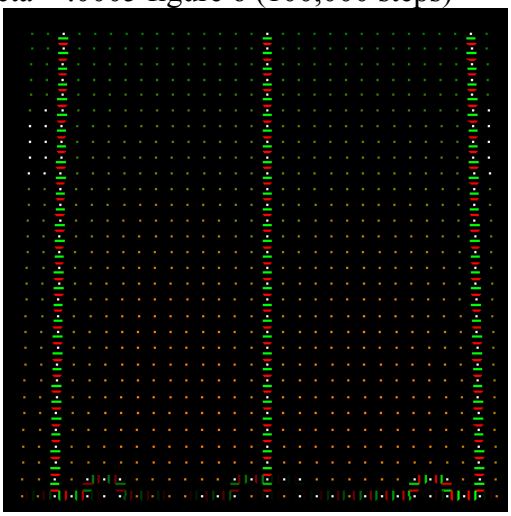


alpha = .0005 figure 5 (100,000 steps) |

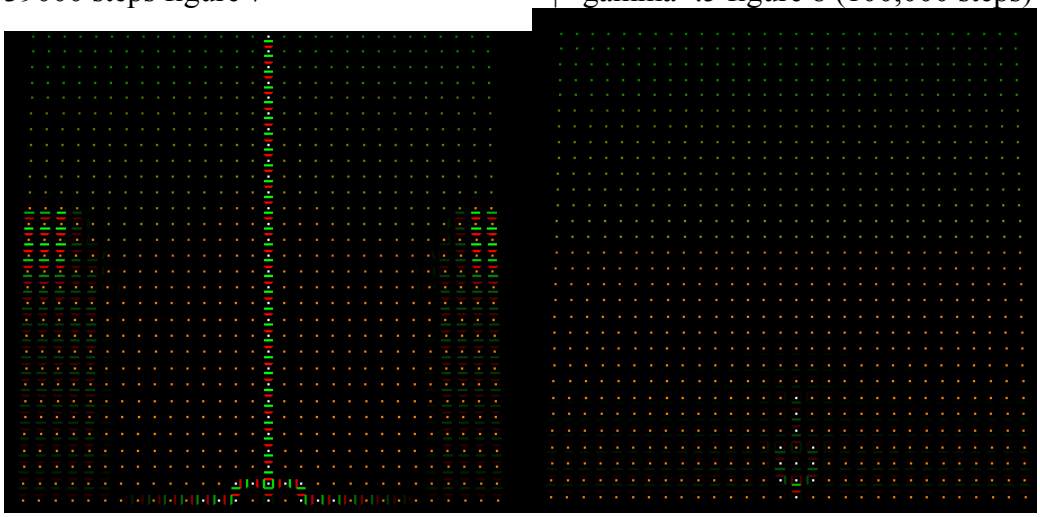


beta = .0005 figure 6 (100,000 steps)

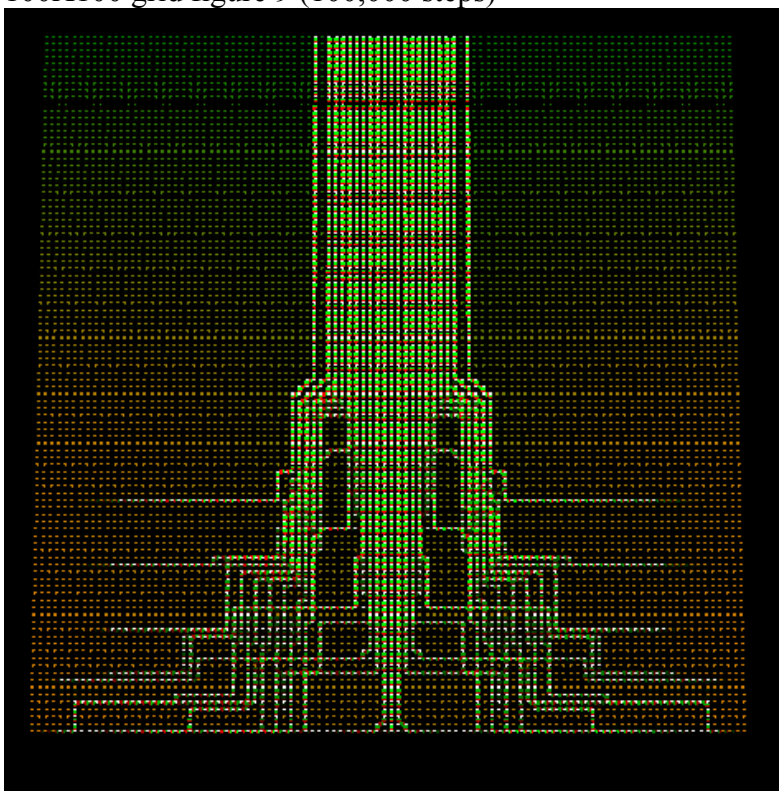




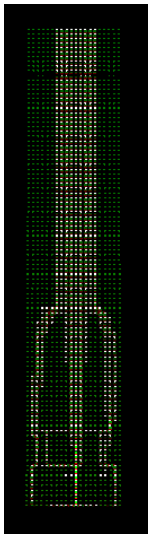
beta = .0005
39000 steps figure 7 | gamma=.5 figure 8 (100,000 steps)

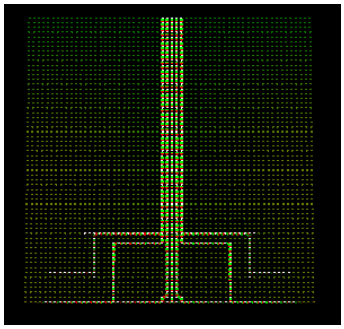


100X100 grid figure 9 (100,000 steps)



20X 100 figure 10 | 60X60 max_D_prime = 30 figure 11
max_D_pirme = 30(100,000 steps) | (100,000 steps)





Appendix B: Code

```
#include <iostream.h>
#include <math.h>
#include <GL/glut.h>
#include <stdlib.h>


//public variables
        //camera variables
static float alpha_camera=0.0, beta_camera=0.0, ratio;
static float zoom=500.0;
static float x=0.0f,y=10.0f,z=0.0f;


//adjust the camera to be centerd
int adjust_x(int i);
int adjust_y(int j);

//leaf variables
static const int number_of_cells_wide=80;    //number of rows and columns of leaf cells
static const int number_of_cells_tall=500;
```

```
static const int number_of_near_cells=4;//number of neighouring cells, standard is 4, can not be
                                                                        //directly altered currently, without some modifcation
                                                                        //to several algorithms, but still makes the procees a bit easier

/* these are the leaf cells that exist,
they are given an i and j coordinate,
and they determine whether a the cells exists or not, by changing these values
we can simulate the leaf being cut after a certain amount of steps or we can
simulate leaf growth by having cells initially non existstent and the
defining them after a certain amoun of steps*/
static bool         defined_cells[number_of_cells_tall][number_of_cells_wide];
/* these are the actual leaf cells,
they are given an i and j coordinate,
and they hold the concetration value of auxin*/
static double leaf_cells[number_of_cells_tall][number_of_cells_wide];
/* this determinses whther it is still a cell or a vein
they are given an i and j coordinate,
if the cell is a vein we don't bother to calculate the diffusion coeficient because we know
it has to be the max, (leaf veins can not go back to regular cells)
*/
static int          leaf_type[number_of_cells_tall][number_of_cells_wide];
/* these are the cell walls,
they are given an i and j coordinate and k(to store wich of the cells wall) coordinate,
and they hold the flux of auxin*/
static double
cell_flux[number_of_cells_tall][number_of_cells_wide][number_of_near_cells];
/* these are the cell walls,
they are given an i and j coordinate and k(to store wich of the cells wall) coordinate,
and they hold the temporary flux so that after the calculations are done everything can be updated
simoultaneuosly so it doesn't matter which way you sweep through the arrays*/
static double
temp_cell_flux[number_of_cells_tall][number_of_cells_wide][number_of_near_cells];
//this holds the diffusion coeficeint for each cell wall
static double
D_prime[number_of_cells_tall][number_of_cells_wide][number_of_near_cells];



static const double step=.01;          //used in euler approximation
static const double D=.325;                    //regulates background diffusion
static const double alpha=.00005;      //regulates production of carriers due to flux
static const double beta=.005;         //background producion of carriers
```

```
static const double gamma=.05;                  //regulates decay of carriers
static const double beta_prime=beta-gamma*D;    //when background diffusion is
introudced this variable is introduced to

        //simplify equation



//this variables are used to set the inital conditions of the leaf, so that the top
//has more auxin
static const double T=D;
static const double F=15;       //determines how much auxin flows into the top row of
cells each step
static const double max_D_prime= 10; //max D_prime, if this is reached then the cell
turns into a vein


//function
//clears array
void clear_near_cells(int near_cells[3]);
//finds all the neighbouring cell walls
void near_cell_finder(int near_cells[3],int i, int j, int k);
//calculates auxin flow
void auxin_flow();
//these are cells that natuarally produce auxin
void auxin_producing_cells();
//caclculates concentration chane of auxin
void calculate_concentration_change(int cell_one_i,int cell_one_j,int cell_one_k,int
cell_two_i,int cell_two_j,int cell_two_k);
//caluclates flux
double calculate_phi();
//caluclates change of diffusiuon coeficcent
double calculate_D_prime(int cell_one_i,int cell_one_j,int cell_one_k);
//this sets the intial leaf concentrationms
double calculate_inital_concentrations(int m);
//this goes through all the flux's of the cell wals and changes the concntration of the call
//accordingly
void update_concentrations();
//this takes all the temp values and stores them into the actual array
void update_flux();
void update_D_prime();
//if the difussion coeficient reaches a creatain limit, it turns into a
//vascular cell and the diffusion coeffienct is clamped off at that point
void clamp_values(int cell_one_i,int cell_one_j,int cell_one_k);
```

```
//this is just in case a vein has negative auxin, this cannot happen in real life
//and therfore this is used for debbuging if it does somehow go below 0
//then this function warns us that something is wrong
void equalizer(int i, int j);

//this is to determine the inital shape
void square();
void circle();

//this is to simulate growth (still glitchy so we didn't use it too much)
void squre_growth();
//window resizing
void changeSize(int w, int h)
{

	// Prevent a divide by zero, when window is too short
	// (you cant make a window of zero width).
	if(h == 0)
		h = 1;

	ratio = 1.0f * w / h;
	// Reset the coordinate system before modifying
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();

	// Set the viewport to be the entire window
	glViewport(0, 0, w, h);

	// Set the clipping volume
	gluPerspective(45,ratio,1,10000);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
	gluLookAt(x, y, z, 0,0,0,0.0f,0.0f,1.0f);
}


//initalize the scene
void initScene() {
   glPointSize(.10);
	glLineWidth(.10);
	glEnable(GL_DEPTH_TEST);
//make a square grid of leaf cells
square();

}
```

```
void renderScene(void) {
	//this renders a grid of leaf cells, and the higher the concentration
	//the more oragne they are the lower the concentration, the greener they are
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
	double color=0;
	double color_1=0;
	double color_2=0;
	int x=0;
	int y=0;
	for(int i=0; i<number_of_cells_tall; i++)
	{
		for(int j=0; j<number_of_cells_wide; j++)
		{		//if the cells is not defined don't bother drawing it
			if(defined_cells[i][j] == 1)
			{	//adjusts the placement of the cells so they are
centered an not overlapping
				x = adjust_x(i);
				y = adjust_y(j);
				if(leaf_type[i][j] == -1){	//if it's not a vein

	color=double(leaf_cells[i][j])/(30*number_of_cells_tall);
					glColor3f(color,0.5, 0.0);}
				else{
					glColor3f(1,1, 1);	//if it is a vein just
makee it white
				}
				//draw the cell
				glBegin(GL_POINTS);
				glVertex3d(x, y,0);
				glEnd();

				//this draws the cell wall
				for(int k=0; k<number_of_near_cells; k++)
				{
					color_1=0;
					color_2=0;
					if(cell_flux[i][j][k] >0){
						color_1=cell_flux[i][j][k]/(150);
					}else{
						color_2=-cell_flux[i][j][k]/150;
					}
					//if the fulx is postive then make the cell
green otherwise make it red

					glColor3f(color_2,color_1, 0.0);
```

```
                                        glBegin(GL_LINES);
                                        if(k==0){
        //bottom
                                                glVertex3d(x+.3, y-.3,0);
                                                glVertex3d(x+.3, y+.3,0);
                                        }else if(k==1){
        //right
                                                glVertex3d(x-.3, y+.3,0);
                                                glVertex3d(x+.3, y+.3,0);
                                        }else if(k==2){
        //top
                                                glVertex3d(x-.3, y-.3,0);
                                                glVertex3d(x-.3, y+.3,0);
                                        }else if(k==3){
        //left
                                                glVertex3d(x-.3, y-.3,0);
                                                glVertex3d(x+.3, y-.3,0);
                                        }
                                        glEnd();
                                }
                        }
                }
        }

        glutSwapBuffers();
}

void rotate_camera(float alpha_camera, float beta_camera) {
        //this controls the camer so you can pan it around the origin,
        //uses spherical coordinates
        //initialy we were going to do a 3-d leaf but we realized it was not practical
        //but the camer code we made worked so we kept it anyway
        alpha_camera=alpha_camera*3.141/180;
        beta_camera=beta_camera*3.141/180;
        x = zoom*sin(beta_camera)*cos(alpha_camera);
        y = zoom*sin(beta_camera)*sin(alpha_camera);
        z = zoom*cos(beta_camera);
        glLoadIdentity();
        gluLookAt(x, y, z, 0,0,0,0.0f,0.0f,1.0f);
}

void move_camera(int direction) {

        glLoadIdentity();
        gluLookAt(x, y, z, 0,0,0, 0.0f,1.0f,0.0f);
```

```
}

void keyboard(int key, int x, int y) {
	//controls the camera panning
	switch (key) {
		case GLUT_KEY_LEFT :
			alpha_camera -= 5.0f;
			rotate_camera(alpha_camera, beta_camera);break;
		case GLUT_KEY_RIGHT :
			alpha_camera +=5.0f;
			rotate_camera(alpha_camera, beta_camera);break;
		case GLUT_KEY_UP :
			beta_camera +=5.0f;
			rotate_camera(alpha_camera, beta_camera);break;
		case GLUT_KEY_DOWN :
			beta_camera -=5.0f;
			rotate_camera(alpha_camera, beta_camera);break;
	}
}

void mouse(int button, int state, int x, int y)
{
	//this initiates the auxin flow, everytime the mouse button is clicked
	//the algorithm is ran several times

static int total_steps=0;
int left_step = 1000;    //determines how many times to run the alogrith if you click the
left mouse button
int right_step = 10000;          //same but for the right mouse button
	if(state==GLUT_DOWN && button == 0)
	{
		cout<<"proccesing..."<<endl;
		for(int i=0; i<left_step; i++)
		{
		auxin_flow();
		}
		glutPostRedisplay();
		total_steps+= left_step;
		cout<<"Done!\n total steps:"<<total_steps<<endl;

	}
	if(state==GLUT_DOWN && button == 2)
	{
		cout<<"proccesing..."<<endl;
		for(int i=0; i<right_step; i++)
		{
```

```
                auxin_flow();
                }
                glutPostRedisplay();
                total_steps+= right_step;
                cout<<"Done!\n total steps:"<<total_steps<<endl;
        }
}

int main(int argc, char **argv)
{
        //Opengl decleration stuff
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
        glutInitWindowPosition(100,100);
        glutInitWindowSize(640,360);
        glutCreateWindow("Leaf Venation");
        initScene();
        glutSpecialFunc(keyboard);
        glutMouseFunc(mouse);
        glutDisplayFunc(renderScene);
        glutIdleFunc(renderScene);
        glutReshapeFunc(changeSize);
        glutMainLoop();
        return 0;
}

double calculate_inital_concentrations(int m)
{
/*this is a fairly simple way to set inital concentrations,
m is the row number and F and T are constant, so basically
the higher the row number(m) the more auxin it will start off with
the auxin decreases from row to row in a linear way, and the bottom
row start off with the most auxin, and top with the least*/
        return F*(m)/T;
}

void auxin_flow()
{
/* This function goes through each cell
then finds the neighbouring cells of the current cell
and then calculates the auxin flow between these two cells*/

int near_cells[3];        //this stores the location of the neighbouring cells
//sweeps throuygh a checkerboard patter, so that doubles are avoided

        for(int i=0; i<number_of_cells_tall; i=i+2)
```

```
	{
		for(int j=0; j<number_of_cells_wide; j= j+2)
		{
			if(defined_cells[i][j] == 1)
			{
				for(int k=0; k<number_of_near_cells; k++)
				{
					//make sure array is cleared before each use
					clear_near_cells(near_cells);
					//find the near cells
					near_cell_finder(near_cells,i,j, k);

						//make sure that the neigghbouring cells
exists, -1 indicates that
						//there is no cell, such as on the corners
where there is only two cells
						//when l is =3 or =4 then the near_cell value
is -1 to indicate they don't exist
						if(near_cells[0] != -1)
						{
							//this calculates the flux of auxin
between the two cell walls
							calculate_concentration_change(i,j,k,
near_cells[0],near_cells[1],near_cells[2]);
						}

				}
			}
		}
	}
	//sweeping through the rest of the cells...
	for(i=1; i<number_of_cells_tall; i= i+2)
	{
		for(int j=1; j<number_of_cells_wide; j=j+2)
		{
			if(defined_cells[i][j] == 1)
			{
				for(int k=0; k<number_of_near_cells; k++)
				{
					//make sure array is cleared before each use
					clear_near_cells(near_cells);
					//find the near cells
					near_cell_finder(near_cells,i,j, k);

						//make sure that the neigghbouring cells
exists, -1 indicates that
```

```
                                                        //there is no cell, such as on the corners
where there is only two cells
                                                        //when l is =3 or =4 then the near_cell value
is -1 to indicate they don't exist
                                                        if(near_cells[0] != -1)
                                                        {
                                                                //this calculates the flux of auxin
between the two cell walls
                                                                calculate_concentration_change(i,j,k,
near_cells[0],near_cells[1],near_cells[2]);
                                                        }

                                        }
                                }
                        }
                }

//these two functions update the variable for each wall
                update_flux();
                update_D_prime();
                //this takes all the flux's and actualy changes the concetration of each cell
                update_concentrations();
//these cells natuarally produce auxin, so that auxin is added on here
auxin_producing_cells();
}



void calculate_concentration_change(int cell_one_i,int cell_one_j,int cell_one_k,int
cell_two_i,int cell_two_j,int cell_two_k)
{
/*this calculater the flux of auxin (cell_flux) beetween cell_one and cell_two
After the flux is calculated D_prime ( the diffusion coeficient) can be recalculated,
because it is dependent on flux.  We use this method to simplify the problem
but we still get fairly accurate results from the differential equation by using this
approximation method
  */
                        temp_cell_flux[cell_one_i][cell_one_j][cell_one_k] = -
(D+D_prime[cell_one_i][cell_one_j][cell_one_k])*(leaf_cells[cell_one_i][cell_one_j]-
leaf_cells[cell_two_i][cell_two_j]);
                        temp_cell_flux[cell_two_i][cell_two_j][cell_two_k] = -
temp_cell_flux[cell_one_i][cell_one_j][cell_one_k];
                        //cell flux is oppotsite for he other wall

                return;
}
```

```
double calculate_D_prime(int cell_one_i,int cell_one_j,int cell_one_k)
{
	/* we use the euler method to approximate the differnetial equation
	by using a small step we can get failry accurate aproximations*/

	double result=0;
	double current_flux = temp_cell_flux[cell_one_i][cell_one_j][cell_one_k];

	D_prime[cell_one_i][cell_one_j][cell_one_k] +=
step*(alpha*pow(current_flux,2)+beta_prime-
gamma*D_prime[cell_one_i][cell_one_j][cell_one_k]);
	clamp_values(cell_one_i, cell_one_j, cell_one_k);
	return D_prime[cell_one_i][cell_one_j][cell_one_k];
}

void clamp_values(int cell_one_i,int cell_one_j,int cell_one_k)
{
//if the diffusion coeficient reache sthe max turn the cell into a vein
	if(D_prime[cell_one_i][cell_one_j][cell_one_k]>max_D_prime)
	{
		leaf_type[cell_one_i][cell_one_j] = cell_one_k;

		D_prime[cell_one_i][cell_one_j][cell_one_k]=max_D_prime;
	}


}



void auxin_producing_cells()
{
	//these cells make auxin
	for(int i=0; i<number_of_cells_wide; i++)
	{
		leaf_cells[0][i] =0;
		leaf_cells[number_of_cells_tall-1][i] +=step*F;	//top row
	}

leaf_cells[number_of_cells_tall-1][number_of_cells_wide/2] +=80*step;


}

void update_D_prime()
```

```
{
	//calculates D_Prime for all the walls
for (int i=0; i<number_of_cells_tall; i++)
{
	for(int j=0; j<number_of_cells_wide;j++)
	{
		if(leaf_type[i][j] == -1 && defined_cells[i][j] == 1)
		{
			for(int k=0; k<number_of_near_cells;k++)
			{
				D_prime[i][j][k] =calculate_D_prime(i,j,k);
			}
		}
	}
}
}

void update_flux()
{
	//tales the flux and updates them to the real flux, to make sure it happens
	//simultaneously
for (int i=0; i<number_of_cells_tall; i++)
{
	for(int j=0; j<number_of_cells_wide;j++)
	{
		if(defined_cells[i][j] == 1)
		{
			for(int k=0; k<number_of_near_cells;k++)
			{
				cell_flux[i][j][k] =temp_cell_flux[i][j][k];
			}
		}
	}
}
}


void update_concentrations()
{
	//takes all the flux's and adds them to the concentration
for (int i=0; i<number_of_cells_tall; i++)
{

	for(int j=0; j<number_of_cells_wide;j++)
	{
		if(defined_cells[i][j] == 1)
```

```
                {
                        for(int k=0; k<number_of_near_cells;k++)
                        {
                                leaf_cells[i][j] += step*cell_flux[i][j][k];
                        }
                        if(leaf_cells[i][j]<0)
                        {       //in case something goes wrong
                                cout<<leaf_cells[i][j]<<endl;
                                equalizer(i ,j);
                        }
                }

        }
}
}

void equalizer(int i, int j)
{       //
        leaf_cells[i][j]=0;

}
void near_cell_finder(int near_cells[3],int i, int j, int k)
{
        //this function finds the fourn neighbouring cells if they exist
        //and returns an array that stores there i and j coordinates, and on which wall
        //they border
        //bottom
        if(k == 0){
                if(i<number_of_cells_tall-1 && defined_cells[i+1][j] == 1){
                        near_cells[0]=i+1;
                        near_cells[1]=j;
                        near_cells[2]=2;
                }
                else{
                        return;
                }
        }else if(k == 1){ //right
                if(j<number_of_cells_wide-1 && defined_cells[i][j+1] == 1){
                        near_cells[0]=i;
                        near_cells[1]=j+1;
                        near_cells[2]=3;
                }
                else{
                        return;
                }
        }else if(k == 2){       //top
```

```
                if(i>0 && defined_cells[i-1][j] == 1){
                        near_cells[0]=i-1;
                        near_cells[1]=j;
                        near_cells[2]=0;
                }
                else{
                        return;
                }
        }else if(k == 3){       //left
                if(j>0 && defined_cells[i][j-1] == 1){
                        near_cells[0]=i;
                        near_cells[1]=j-1;
                        near_cells[2]=1;
                }
                else{
                        return;
                }
        }


}

void clear_near_cells(int near_cells[3])
{
//clears the array
                for(int j=0; j<3; j++)
                {
                        near_cells[j]=-1;
                }

}









int adjust_x(int i)
{
return i-(number_of_cells_tall/2);
}
```

```
int adjust_y(int j)
{
return j-(number_of_cells_wide/2);
}



void square()
{
	double row_value=0; //this determines the concentration of that row

	//clear the arrays
   for(int i=0; i<number_of_cells_tall; i++)
	{
		row_value=calculate_inital_concentrations(i);
		for(int j=0; j<number_of_cells_wide; j++)
		{
			leaf_cells[i][j]=row_value;
			leaf_type[i][j]=-1;
			defined_cells[i][j] =1;
			for(int k=0; k<number_of_near_cells; k++)
			{

				cell_flux[i][j][k]=0;
				temp_cell_flux[i][j][k]=0;
				D_prime[i][j][k]=0;
			}
		}
	}
}

void circle()
{
	double row_value=0; //this determines the concentration of that row
	int x=0;
	int y=0;
	//int i=5;
	int max_theta = 2*3.14159265+1;
	//clear the arrays
   for(int i=1; i<number_of_cells_tall/2; i++)		//in this case this is r
	{
		row_value=calculate_inital_concentrations(i);
		for(double j=.01; j<max_theta; j+=.01)	//in this case this is theta
		{
			//cout<<i<<" "<<j<<endl;
			x= (i*cos(j))+number_of_cells_tall/2;
```

```
                        y= (i*sin(j))+number_of_cells_tall/2;
                        if(y<0){
                        cout<<"X: "<<x<<" Y: "<<y<<"  "<<endl;}
                        leaf_cells[x][y]=row_value;
                        leaf_type[x][y]=-1;
                        defined_cells[x][y] =1;
                        for(int k=0; k<number_of_near_cells; k++)
                        {

                                cell_flux[x][y][k]=0;
                                temp_cell_flux[x][y][k]=0;
                                D_prime[x][y][k]=0;
                        }
                }
        }

        //defined_cells[number_of_cells_tall/4][number_of_cells_tall/4] =0;
}


void squre_growth()
{
//erased because of glitches
}
```