GroIMP Tutorial

Gerhard Buck-Sorlin, Jorad de Vries, Junqi Zhu

The *Italics* text is instructions, the Courier text is source code to write, the normal text is explanations.

**Simple plant, consisting of an internode and an apical meristem**

*Open Example01.gsz, the editor window on the right hand side is still empty.*

*Write source code*:

Axiom ==> Bud

Bud ==> Internode Node Bud

In order for this to work as a program, we have to write a bit more (by wrapping the rules with some method code):

protected void init ()

[

Axiom ==> Bud;

]

public void run ()

[

Bud ==> Internode Node Bud;

]

run() is a public method. public: so that the user (you) can use it to run the model, and a method is a function that is carried out to run what is found inside, in this case the rule(s) are carried out. The name of the method doesn’t have to be run(), by the way: *rename run() into executer(), then rename it back to run()*

*change* public *to* private*, then save the code: The method disappears from the left-hand-side panel, then change it back to* public*.*

*Press run() a few times* and explain what is to be seen*.*

*Move the object, zoom in and out, turn it, using the corresponding tools.*

*Press reset, then Run run, press stop.* Explain the potential danger of Run run for certain quick models (🡪 fills up the memory and crashes the system).

**Introducing leaves into the model**

(*Have Example02.gsz opened in the background, already filled with the source code from example 1*).

Now, we are going to add leaves to the simple stem. Leaves are in some way “branches”, that is they are appendages on the side that do, however, not branch further. Here is how you add leaves:

*Insert the source code between Node and Bud:* [ Leaf ] *then save and rerun by pressing Run a few times. Turn the plant to show the “stuck” leaves.*

Now, the leaves are normally not sticking to the branch, but they are inserted at a certain angle. To tell the programme that it should put the leaf at such an angle, we use this command: *RL(70)*. You already know the little turtle from the lecture: It has three axes around which rotations can take place. The RL-command will perform a rotation (R) around the left or L-Vector. *Draw a turtle with three axes at the blackboard, then show the vectors with the right hand*. *Now save and rerun the model for a few steps*. So what we have just done can also be called “pitching down” (like a landing or sinking plane).

As you can see, the leaves are all sticking to one side, which is ugly and unnatural. Quite more often, leaves are arranged differently, for instance opposite each other. To create an alternation of leaves (i.e. one to the left, then one to the right, then again left, and so on), we could use the RL command again and tell it to pitch up or down alternately. However, this is cumbersome. A much easier way is to do a rotation around the head vector of the turtle by 180 degrees (this is called rolling by the way) before a new bud is inserted. Let’s do this: *Insert* RH(180) *before the* bud *symbol*.

*Save and rerun*. Finally, let’s insert another internode below the bud, because it seems the apical bud is sticking directly to the leaf. *Insert* Internode *before* RH(180) *(doing it the other way around doesn’t make a difference in this case).*

**Introducing branches**

*Maximize window Example03.gsz. The editor should contain the source code of Example 2.*

Now we have a single stem with leaves, which endlessly produces new phytomers at the end. So what? Real plants branch, i.e. they produce new phytomers from lateral buds. So how can we teach the model to do this? First of all, let’s get rid of the leaves for reasons of simplicity, we will put them back in later. *Comment the leaves out using /\* \*/.* So can anybody make a suggestion already what we should do? No? Ok, it is quite simple: Just reusing the bud symbol in a branch does the trick: [ Bud ]. But, of course, we know that we have to add a rotation so that the new branch is growing out at an angle, say 50 degrees, thus RL(50).

*Save and run for a few steps only*. You see that the branched structure is completely flat, i.e. 2D. Let’s modify the RH command to make the plant branch to all sides, i.e. in 3 dimensions:

*Insert* RH(137.5)*, then save and rerun*

137.5 degrees is an angle very often encountered in nature, it is referred to as the Golden Angle. At this (so-called phyllotactic) angle, the least overlap and therefore self-shading of leaves within the same shoot occurs.

Put the leaves back in: [RL(60) Leaf]. As you see, the leaves are just another “branch” inserted at the node, but with a bigger angle, because the lateral bud is in the axil of a leaf and not the other way around. *Run again, this time with Run run.*

You get a real ball of leaves, with just one rule! Why do we get this? Because the bud rule is unrestricted, it can always be applied, it has no condition.

But what we really see in nature is, of course, a constrained growth of an organism, both in number of organs and in the growth of a single organ. We furthermore observe that the number of branching orders, e.g. in trees, is quite small, usually less than three or four:

**Restricting branching order**

*Open Example03a.gsz. The window contains the source code of Example03.gsz*

So, the way to restrict for instance branching order is to give the bud a parameter, let’s call it o for order. Orders are usually counted starting from 0 or 1. Let’s start with 1. A parameter is usually set with a module (this is the name for these organ symbols) by writing its identifier or a value in brackets just behind the name of the module: So the bud should start with order 1: *write* Bud(1) *in the Axiom.* In the bud rule, we say Bud(o), then the programme knows, aha, this is the order, and it currently has the order 1 (from the axiom). On the right-hand side, we want to leave the order of the terminal bud as it is, so we also write Bud(o), but the lateral branches are incremented in order by one, thus Bud(o+1). What is missing now, is a condition: We want branching to stop once order 4 is reached, so we have to write

, (o<4) at the left-hand side.

*Save and rerun using Run only. Click on the terminal buds to show the parameter* order. You can check that the order of the different buds is indeed increasing to 4. And, as predicted, all the lateral buds with order 4 stop growing out.

**Introducing the phyllochron**

*Open Example03b.gsz. The window contains the source code of Example03a.gsz*

Ok, but the shoots that grow, do so at a high speed. This is because at every step, a new phytomer is produced at the tip of the shoot. If we want to change the speed at which new phytomers are formed, we have to introduce a parameter which puts the bud at rest for a certain number of steps before the next phytomer is formed. This time between the appearance of two phytomers or more obviously leaves, is referred to as the phyllochron. We can thus define a constant, call it phyllo and give it a value of, say, 25:

const int phyllo = 25;

int is the type of the constant, an integer number.

Now we have to tell the bud that it has a new parameter, let’s call it p. Here is how it is done: initiate the parameter (i.e. give it a concrete value) the first time the module is declared. We don’t write a number but instead write bud(phyllo, 1), then the programme knows that it has to insert the number 25 here. Later, we can change this number more easily in the declaration of the constant phyllo at the top. What should be done with this number by the model? The idea is that it should count down from 25 (the current value of phyllo) to 0, before producing the next phytomer. The module bud refers to this parameter under a name, so let’s call it p. p can have values between phyllo and 0. If p is larger than 0, it should be reduced by 1. Whenever a new bud is inserted at the tip of the shoot or at the sides, the parameter should be re-set to the value of phyllo, because this new phytomer has to wait. We now have two cases: p>0: wait and p==0: grow. Thus we need to write a new rule, the waiting rule:

Bud(p,o), (p>0) ==> Bud(p-1,o);

And we have to modify the existing rule:

The condition is enlarged: (p==0 && o<=2)

(note that we set the maximum order to 2 now). And:

Bud(phyllo, o+1) resp. Bud(phyllo, o)

*Save and rerun now. Change the value of phyllo and observe the difference. Show the values of p by clicking on different buds.*

**Flowering**

Now, plants don’t stay vegetative forever. In fact, what you observe in many plants is that an apical meristem, after having formed a certain number of phytomers, will decide to form a terminal flower and thereby use itself up, i.e. there will be no further formation of phytomers. Let’s tell the bud to produce a flower once it has formed a given number of phytomers. In order to do so, we have to introduce a new parameter to the bud, r (for rank), which is counted up to, say, 10, and if r == 10, it produces a flower. First, we change the axiom rule, r is initially 1:

**Bud(1,phyllo,1)** (yes, the order of the parameters DOES matter, so please don’t confuse it! I have no time to go into details here)

Then, we need a new rule for the flower:

**Bud(r,p,o), (r==10) ==> Internode Internode Flower;**

Next, we need to modify the first and the second rule of the method run():

**Bud(r,p,o), (r<10 && p==0 && o<=2)**

As long as r is smaller than 10, produce a phytomer.

**Bud(r+1,phyllo,o+1)**

Increment r by 1 when a new phytomer is produced. Do this also on the side branches (because we want flowering to happen everywhere at the same time, not earlier at the main stem).

Now, let’s insert a turtle command which improves realism by bending the shoots slightly upwards:

**RV(-0.1)**

*Insert this command before the first two Internode modules.*

In order to see something you also need to define the new Flower module:

**module Flower ==>**

**RU(180) Cone(0.3, 0.3).(setColor(0x82B417))**

**M(-0.25) RL(90)**

**[**

**for (int i = 1; i <= 5; i++) (**

**[ RU(i\*360/5) RL(20) Parallelogram(2, 1).(setColor(0xFF00FF)) ]**

**)**

**]**

**RU(45)**

**[**

**for (int i = 1; i <= 5; i++) (**

**[ RU(i\*360/5) RL(40) F(0.3, 0.1, 14)**

**RV(-0.3) F(0.3, 0.1, 14) RV(-0.3) F(0.3, 0.1, 14) ]**

**)**

**]**

**RU(-45)**

**[**

**for (int i = 1; i <= 5; i++) (**

**[ RU(i\*360/5) RL(70) Frustum(0.7, 0.2, 0.05).(setColor(0x8DAF58)) ]**

**)**

**]**

**;**

Note that this module is defined like a replacement rule and it also works like one: every time the compiler (the part of GroIMP which translates the code) encounters the symbol “Flower” in the code, it replaces it by all these turtle symbols listed on the right-hand side of the ==> operator. You will also notice that a so-called for-loop has been used three times, e.g.

**for (int i = 1; i <= 5; i++) (** …some turtle commands… **)**

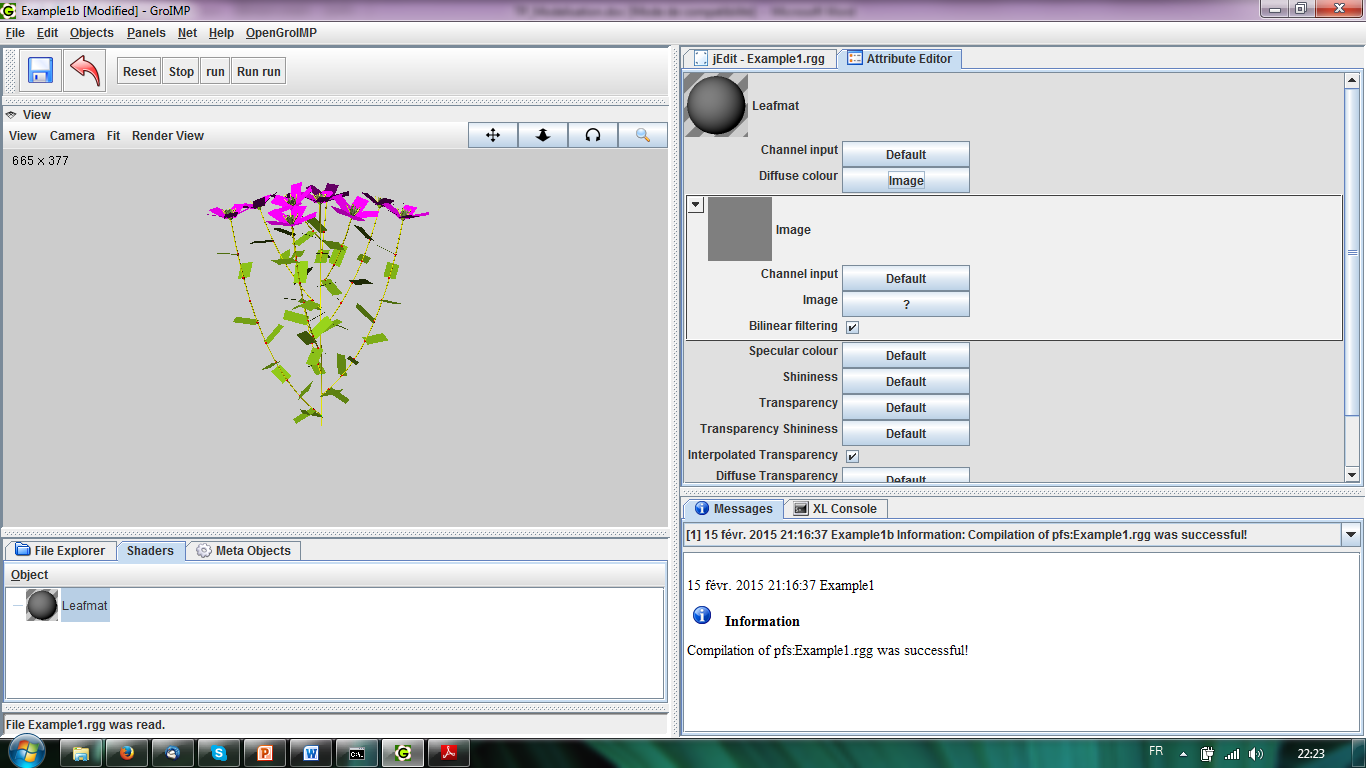
This construction is very useful to repeat the same sequence of turtle commands several times, e.g. draw petals five times, then draw stamens five times, then draw carpels five times…

You will also see that two new geometric objects have been used, **Cone** and **Frustum**. You can try to insert them into the View window to see how they look like (Main menu: *Objects -> Primitives -> Cone or Frustum*). Then you can also modify their parameters or their colour, using the Attribute Editor. In the end please delete the object again (Main menu: *Edit -> Delete*)

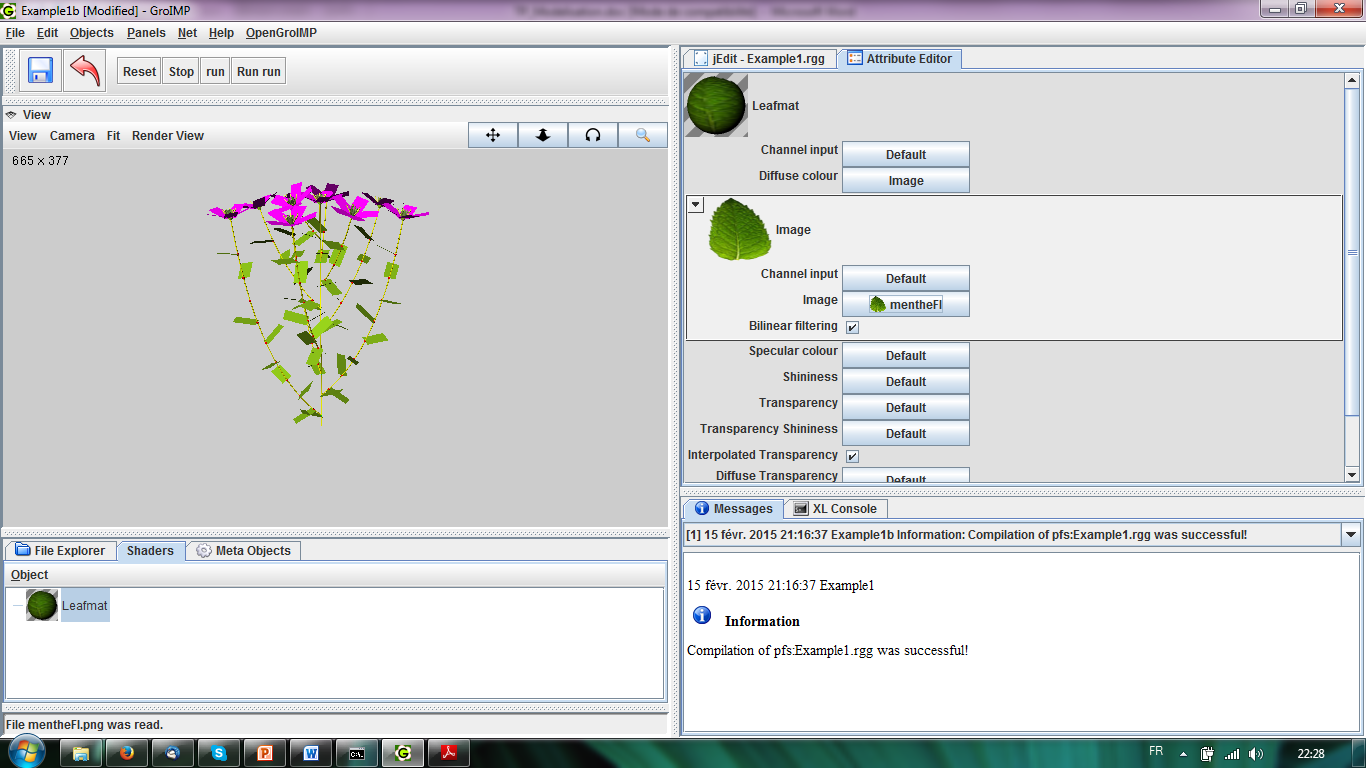
If you want your plant to look a little bit more realistic, then you need to use a shader. A shader is in fact a little computer program that is written to give a color and texture to an object. In fact we already did use shaders previously, every time we used the method setShader somewhere! What we now want to do is to create and then use a shader derived from a photographic image of a plant organ. This is very straightforward as you will see. First you need to prepare an image, using an image processing software, like the Gimp or Photoshop. The image should show the surface of the organ quite representatively. The image of a flattened leaf blade with a white background will be a good choice. Using the Gimp you can easily cut out the background, i.e. make it transparent (as will hopefully be shown in the course). Then you save the image as a png or jpg. Next we go back to GroIMP and do the following: From the Main Menu you choose:

*Panels -> Explorers -> 3D -> Shaders.*

This will open a window called Shaders. As you have already been shown you can anchor this window somewhere in the lower left corner of the GroIMP windows, perhaps beside the MetaObjects window. Then in the Shaders window you go in the Menu to *Object -> New -> Lambert*. A grey filled sphere will appear below with the name *Lambert*. You can click once on it, wait, and then click again (like a very slow double click!) to change its name, to *Leafmat*. Now, when you double-click (quickly!) on the grey sphere you will see it properties in the Attribute Editor window on the right: at least ten different shader parameters could be changed, but don’t worry, we won’t do this. We will just change the diffuse color, by clicking on the second button (which currently reads “RGB”): From the scroll-down menu which appears we will choose **Surface Maps** and then **Image**. If you now see something like this:



…then you are on the right way. If not, please ask…

Next we will click on the “?” button besides Image and from the scroll-down menu choose **From File**. We now have to choose the image file (I hope you remember where you put it!) and choose **Add the file**. The moment you have done this the Attribute Editor should change to something like this:  


The shader is now ready for use, we only need to make a reference to it in our model :

**const ShaderRef leafmat = new ShaderRef("Leafmat");**

The reference leafmat can now be used as a shader in the Leaf module:

**module Leaf extends Parallelogram(2, 1) {**

**{ setShader(leafmat); }**

**}**

*You can now do the same on your own for the other organs, node, internode and petal.*

(The images will be provided, but you can try your own images)

**Light interception of leaves**

Now it’s time to introduce some functionality: One of the first things that come to mind when thinking about plant functions is, of course, photosynthesis. In order that the plant can do photosynthesis, it needs energy, water and CO2. Thus, first of all, some plant parts (normally the leaves), need to intercept and absorb light and transfer part of the energy gained from this process onto specialized molecules. Let’s first look at light capture: There are several ways to model this. And the nice thing about a 3D model is that we have information about the position of each leaf. We further have information about the position of light sources. To actually model the light environment in our 3D scene, we first need to create a light source:

**module MyLamp extends SpotLight {**

**{**

**setPower(200.0); // power in W**

**setAttenuationDistance(50.0); // in m**

**setAttenuationExponent(0.0);**

**setInnerAngle(22.5 \* Math.PI/180.0);**

**setOuterAngle(30.0 \* Math.PI/180.0);**

**}**

**}**

Then, this light source needs to be coupled with a light node to give the emitted light a colour:

**module MyLight extends LightNode(1.0, 1.0, 1.0) {**

**{ setLight(new MyLamp()); }**

**}**

We insert the lamp into the scene using a very strange command in the init method:

**==>> ^ M(50) RU(180) MyLight;**

This rule has no left-hand-side, and the operator looks weird, it has a second “>”! But don’t worry, this is alright. The lamps is now placed 50 units above the plant and then turned downwards (RU(180)) to make sure it shines on the plant.

Now let’s invoke the radiation model of GroIMP:

**LightModel lm = new LightModel(100000,5);**

Furthermore, our **Leaf** module gets a parameter named al that stores the value of absorbed light after each run of the light model:

**module Leaf (float al)**

**extends Parallelogram(2,1).(setShader(leafmat));**

In the run method, the Leaf is initiated with al = 0, thus **Leaf(0)**.

Furthermore, we introduce a new method, absorb(), which will invoke a method of the radiation model that measures the amount of absorbed light for each leaf and writes the result to the parameter al:

**protected void absorb ()**

**[**

**lf:Leaf ::> {**

**lf[al] = lm.getAbsorbedPower3d(lf).integrate()\*2.25;**

**//println(lf[al]);**

**}**

**]**

Also we made this method protected (as with run()), because we are not going to invoke it directly but from another method that we will call grow():

**public void grow ()**

**{**

**run();**

**lm.compute();**

**absorb();**

**}**

This method invokes run () once, then runs the light model (lm.compute()), then the rule absorb() which measures light.

Next, we would like to see from the colour of the leaf how much light it has absorbed. For instance, it could be dark green if it has absorbed almost nothing and very bright green or yellow if it has absorbed a lot. On the other hand, we don’t want the light model to get confused with the changing colour (in reality the color doesn’t change, at least not by absorption of light). So we want the light model to work on the same colour but at the same time visualize the absorbed light. In order to achieve this we need a special shader called AlgorithmSwitchShader. This shader has two component shaders, one shader for the light model, and one shader for the visualization. For the light model we will use the simple GREEN shader, whereas for the visualization we are going to use an RGBAShader in which the RED, GREEN and BLUE parameters will be reset at every step as a function of the absorbed light al. Thus, first of all we need to rewrite the Leaf module by changing the setting for the shader:

…

**setShader(new AlgorithmSwitchShader(new RGBAShader(0, 1, 0), GREEN));**  
…

Next, we will add in the absorb() method a command that will reset the shader of the leaf:

**lf.(setShader(new AlgorithmSwitchShader(new RGBAShader(lf[al]/5.0, lf[al]\*2, lf[al]/100.0),GREEN)));**

Make sure this command is within the execution rule of the Leaf.

Note that the shader of each leaf is now reset using the value of al to paint the leaf, thereby indicating the amount of absorbed light, a gradient of green.

**Plotting the absorbed light in a chart**

Maybe by now you are a bit fed up of just looking at the plant and what it does (or not…). Therefore, we might want to see model output as a chart, as we might know from other models. Let’s output the total light intercepted by all leaves, and this at every step of the simulation.

First of all, we need to declare a kind of data table in which the output is stored:

**const DatasetRef lightdata = new DatasetRef("Light intercepted by canopy");**

This command reminds a bit of the command we used earlier to define a shader reference. In fact it is the same principle: lightdata is the name of the dataset reference (type DatasetRef), and this is invoked with a so-called constructor: the key word new and then DatasetRef("Light intercepted by canopy").

Next, we need to initiate the chart in the init method. For this, we will use another method that we are going to define later:

**protected void init()**

**[**

**{ initChart(); }**

**…**

**]**

Also, we want to update the chart at every step, so we will add a new line to the grow() method:

**updateChart();**

The latter is another new method we still need to define.

Now, let’s define the two new methods, first initChart():

**protected void initChart() {**

**lightdata.clear();**

**chart(lightdata, XY\_PLOT);**

**}**

This method does something with the data table lightdata, in fact it empties it at each new run. The chart command does the actual plotting: It creates an x and y plot with the data table, using as x the time (simulation steps) and as y the sum of light absorbed by all leaves. The table is filled up with data by the updateChart() method:

**protected void updateChart() {**

**lightdata.addRow().set(0, sum((\* Leaf \*)[al]));**

**}**

It is the addRow() method of DatasetRef which does the actual job, more specifically its submethod set(), which adds the data into the 0th column of the table: The data are produced by the sum method, which searches for all leaves (\* Leaf \*) that are already produced and then sums up their parameter al. You can also have a look at the data table itself:

Main Menu: *Panels -> Explorers -> Datasets*

**Leaf growth**

Currently, the leaves and internodes of our plant just “pop up” with their final size, although in a real plant they would, of course, grow. Growth in length can be modelled fairly easily: first, we need to give both Leaf and Internode a parameter length. Since both organs are extending objects that do already have a parameter length (Box and F), we just declare super.length to inherit that parameter. For the leaves we do the same with width (super.width). Then we initialize the length (and width) with a small value (0.1 and 0.07) in the init() method.

Next, we need to find an appropriate function that describes growth. The logistic function is actually quite accurate when it comes to description of organ length. It is sigmoid. Its derivative can be used to describe the grow rate as a function of organ age. We declare it as an external function in the code:

**public float logistic (float maxdim, int time, float phylloM, float slope)**

**{**

**return (slope\*maxdim\*Math.exp(-slope\*(time-phylloM)))/**

**((Math.exp(-slope\*(time-phylloM))+1)\*\*2);**

**}**

The function has three parameters, for now we are only interested in organ age. This we declare as a further (integer) parameter in Leaf and Internode: **int age**

Next, we need to initialize age with 1 in the run() method. Then, we extend the Leaf rule in the method that we now rename to absorbAndGrow() to model leaf ageing and extension in length and width:

**lf[age]++;**

**lf[length] += logistic(2,lf[age],10,0.5);**

**lf[width] = lf[length]\*0.7;**

Finally we add another execution rule to model internode extension, and this you can do yourself!

**Absorbing tiles**

We also want to visualize how much light is reaching the ground. To do that, we will cover the soil surface around the plant with light-absorbing tiles:

**module Tile(float len, float wid) extends Parallelogram(len,wid)**

**{**

**float al;**

**};**

In the Axiom rule, preceding the Bud, we will place 1600 little tiles:

**[**

**RL(90) M(4) RU(90) M(-4)**

**for ((1:40)) (**

**for ((1:40)) (**

**Tile(0.25, 0.25).(setShader(new RGBAShader(0.6, 0.3, 0.1)))**

**)**

**M(-10) RU(90) M(0.25) RU(-90)**

**)**

**]**

Like the leaves the tiles will become brighter when they absorb light. Update is done in the absorbAndGrow() method:

**p:Tile ::> { p[al] = lm.getAbsorbedPower3d(p).integrate();**

**println(p[al]);**

**p.(setShader(new AlgorithmSwitchShader(new RGBAShader(p[al]\*300,p[al]\*200,p[al]),new RGBAShader(0.6,0.3,0.1))));**

**}**

**Simple linear leaf photosynthesis**

Next we want to have some rudimentary photosynthesis. In its simplest version, the model of photosynthesis is simply a linear function of absorbed radiation. This approach is called Radiation Use Efficiency and works well as a rough tool for many crops.

We first add another parameter, as, to the leaf module. In this parameter the output of our primitive photosynthesis model will be stored:

**module Leaf(super.length, super.width, float al, int age, float as)**

**extends Box(length, width, 0.01) {**

**{**

**setShader(new AlgorithmSwitchShader(**

**new RGBAShader(0, 1, 0), GREEN));**

**}**

**}**

Then we define a simple conversion factor that will serve to calculate the amount of assimilates in one step from the quantity of absorbed light:

**const float CONV\_FACTOR = 0.2;**

Don’t forget that the Leaf now has a further parameter, which needs to be initiated when the Leaf is first invoked:

… **Leaf(0.1, 0.07, 0, 1, 0)** …

The method absorbAndGrow() is modified accordingly:

**protected void absorbAndGrow()**

**[**

**lf:Leaf ::> {**

**…**

**// amount of assimilates**

**lf[as] = lf[al] \* CONV\_FACTOR;**

**// amount of assimilates of all leaves**

**float lfas = sum((\* Leaf \*)[as]);**

**// dependency of growth on availability of assimilates**

**if (lfas > 0) {**

**lf[length] += logistic(2, lf[age], 10, 0.5);**

**}**

**lf[width] = lf[length] \* 0.7;**

**}**

**]**

Note that we also introduced a new condition to leaf growth: it is now dependent on the global availability of assimilates. Before leaf growth can occur it is checked whether the sum of assimilates of all leaves is not zero.

**Aging of flowers**

As you know flowers don’t stay on the plant for a long time after blooming; in fact they perish rather quickly and are either dropped altogether or are replaced by a growing fruit after successful pollination. In our model we can very simply introduce aging and dropping of flowers. First, the Flower requires two new parameters, age and max\_age:

**module Flower(int age, int max\_age) ==> … ;**

When the Flower is introduced in the run method for the first time it is initiated like this:

**Bud(r, p, o), (r == 10) ==> … Flower(1, irandom(10, 15));**

The initial age is thus 1, whereas the maximum age that the flower can attain is between 10 and 15 days, as determined by this command, irandom(10, 15), which will draw a number between 10 and 15, with equal probability.

As long as age has not attained max\_age, age is incremented by one, i.e. the Flower ages:

**Flower(t, m), (t < m) ==> Flower(t+1, m);**

When age is equal to max\_age the flower is dropped:

**Flower(t, m), (t >= m) ==> ;**

Try this yourself: Introduce leaf shedding after 20 steps !

**More realistic non-linear photosynthesis**

While Radiation Use Efficiency is fine as an approximation for rough calculations, it is not suitable for calculating leaf photosynthesis. The relationship between incident light and photosynthesis rate is called the *light response curve*, and it is non-linear. In fact, photosynthetic rate becomes saturated at some high light intensities, as it also depends on the utilization rate of fixed carbon as well as on the regeneration of ribulose 1,5-bisphosphate (RuBP), a key enzyme of the light-independent reaction of photosynthesis (Calvin cycle).

In this exercise we want to further inspect two rather simple photosynthesis models that are both based on non-linear relationships between incident light and photosynthesis rate (difference between incident and intercepted: incident is the amount of light *available before* it hits the leaf; intercepted is the amount of light that is *captured* by the two photosystems inside the chloroplasts).

In this more realistic photosynthesis model, the rate of carbon fixation or carbon exchange is not directly linearly related to the light quantity but becomes saturated with increasing photosynthetic photon flux density:

(to be completed)

**List of useful turtle commands:**

**F0, F, F(***l***), F(***l,d***), F(***l,f,c***), F(***l,d***).(setShader(***COLOR***))**: draw a line (cylinder) of standard length, of length *l*, of diameter *d*, of color *c*, or set shader of color *COLOR*

**M, M(***x***)**: move *x* steps in the HEAD direction

**RU(***a***), RL(***a***), RH(***a***)**: rotate around the UP, LEFT or HEAD vector by *a* degrees (*a* can be positive or negative).

**RG, RV(***b***)**: special orientation commands. **RG** = turn head of turtle strictly downwards (gravitropism); **RV(***b***)**: bent head of turtle upwards (*b* negative) or downwards (*b* positive). Useful range of *b*: <0 to -1 for upbending, >0 to +1 for downbending.

**[** , **]**: branch (open **[**, and close **]**)

**Declaring a module:**

**module** *moduleName* (list of parameters: *type name1*, *type name2*, …) **extends** *knownObject***();**

Examples:

module Leaf; //invisible

module Leaf extends Parallelogram; //visible as a plane surface (parallelogram) with standard length and width

module Leaf extends Parallelogram (10,7).(setShader(GREEN)); // visible as a green parallelogram with length 10 and width 7

module Leaf (super.length, float width) extends Parallelogram (length, width).(setShader(GREEN)); // visible as a green parallelogram with length and width specified during the simulation run. Each new Leaf needs to be initialized with the values for length (inherited from Parallelogram, thus “super”) and width, e.g. “Leaf(10,7)”.

Alternative (object instantiation):

**module** *moduleName* (list of parameters: *type name1*, *type name2*, …) **==>** *String of turtle commands*

Example:

module Leaf(length,width) ==> RL(20) F(10,1) Parallelogram(length, width) .(setShader(GREEN));

// The Leaf object is interpreted as a string of turtle commands: a rotation, a cylinder, and a parallelogram.

**Declaring a variable or constant:**

*(***const***) type varName* **=** *varValue***;**

Example:

Variable:

float initialIntLength = 10.5; //the value is assigned here but may change during the simulation!

Constant:

const float myFixedFinalLength = 20.6; //declared once as a global constant, cannot change!

Both variables and constants can be arrays of values (e.g. from measurements in time or along a gradient)

const float[] myMeasuredInternLength = {1.0,2.1,3.2,4.5,5.8,6.1,7.6,8.6,9.5,10.5}; //measured internode lengths of a shoot consisting of ten internodes, from first to tenth rank.

A particular length from the array is then chosen with an index, from 0 to 9

MyShoot ==> for(int **i**:0:9) (F(**myMeasuredInternLength[i]**));

**Declaring a method:**

**protected** | **private** | **public** **void | int | float | double | Boolean** *methodName* (type paramName1, type paramName2, …)

[

//on or more rules, e.g. Axiom ==> A(1);

]  
  
or  
  
{

//one or more imperative statements, e.g. println(myOutput);

}

Example:

protected void init() //note that this function has NO output, it is thus void!

[

Axiom ==> RU(90) A(1) B(2); //A and B are modules that need to be defined before!

]

Another example:

public float myFunction(int time) //this function requires as input an integer value (time) and returns a //float value. It is used as “myFunction(20)” where 20 is the value for time.

{

float p18430 = 25;

float p2 = 50;

return (p1\*p2)/time;

}