**Appendix I**

**Summary of GroIMP programming tips for plant modelling**

Junqi Zhu

**I.1 Model structure**

**Inheritance structure** using module inheritance can dramatically improve the structure of model, and reducing the length of the code. For instance, a hierarchical structure like: organ (base of all organs) – growing organ – visible organ can improve the readability and efficiency of the program.

Module organ contains the general organ properties such as species, plant number, rank and age, and abstract method; growing organ and visible organ can overwrite the abstract method. When we directly call the method in organ, it will update the value of all its subclasses as well.

Module growing organ is the parent module for all organs that grow. It contains the method for calculating the organ sink strength and assimilates allocation.

Module visible organ is designed for organs that are doing photosynthesis. Thus it contains the method for calculating photosynthesis.

For realization, we can make Apex extends organ, root and flower extend growing organ, leaf, internode and petiole or sheath extend visible organ.

**File separation** separate the files for different growth and development processes, and others like model initialization, parameters and environment conditions can greatly improve the readability of the model.

**Unify growth functions** using the same growth function for all the organ growth or extension can greatly reducing the length of code, and can unifies the parameter input. For instance, we use the beta growth function for calculating the sink strength of all organs with specified parameter set for different organs. When it is impossible to use the same function for all organs, e.g. calculate the dimension of different organs, an empty method can be defined in the parent module. This empty method can be override in subclass modules. By this way, we still can call the method in the parent module to update all subclass modules. For instance, organ.calcDimension() will update the dimension of all growing organs which have overwrite this method in their module definition.

**Group** An abstract module can be built up for collecting information for organs either on the same phytomer, branch, plant, row or field. Those abstract modules can facilitate the output at different levels, e.g. plant level and field level. They are also very useful for solving inter-dependent growth conditions, e.g. growth of an individual leaf depends on the availability of assimilates of the whole plant. The abstract modules can be inserted at organ initialization, e.g. a:Apex, (a.shouldProduceVegetative() == true) ==> phytomerBase Internode [Sheat Leaf]

**Encapsulation** In object-oriented programming, information or data of a certain class are often written within the class. This protects the information from direct access from the outside and makes the information consistent throughout the whole program, see the main text. Based on this principle, it is a good practice to define functions within the module for giving values to the parameters, especially when the organ is first initiated. For example, when a new organ is initiated, we can call organ.initiated() to give all the initial conditions to this organ.

**Practical tips**

**I.2 Model syntax**

For Java syntax, the for loop use {}

for (int a = 1; a < 11; a++) {

System.out.println("Count is: " + a)**;**

//Within java syntax, you can use square brackets [ ] to use turtle commands

// put semi comma after each expression in Java syntax

}

For turtle syntax, the for loop use ()

for (1 : nrRows) ( //row configuration

Translate(plantDistance,0,0) // no semi comma needed

//Within turtle syntax, square brackets are interpreted as branch; here each row is one branch.

//Note after the operation of branch, it goes back to the original point where the branch starts.

for (1:nrPlants) (

) )

For L system syntax, use square bracket [ ]

**I.3 Visualization method**

**Instantiation rule** links the module directly with the desired geometry. Of course, it is better to make the property of this geometry linked to a certain parameter of the module, e.g. length. Note within the instantiation rule, we can also use for or if else to control the visualization. For instance,

module Y(int n) ==> for(1 : n) (Cylinder.(setShader(RED)) Cylinder.(setShader(WHITE)));

Furthermore, by using conditions in the instantiation rule, we can just show the structure that we want and make others invisible. This can be realized by **setLayer**. **s**etLayer specifies the layer for visualizing the object in GroIMP. In the view window (view - visible layer), we can select the layers to visualize.

Modular Internode extends visible organ ==> if (border == false) (Cylinder (length,width). (setShader (internodeTexture), setLayer(1))) else (Cylinder (length,width). (setShader (internodeTexture), setLayer(1)));

**Shader** links object surface with the real object texture, e.g. image we took.

Modular Internode extends visible organ ==> F(length,width).(setShader(internodeTexture));

**AlgorithmSwitchShader** chooses among three shaders, one for the visualization in the user interface, one for the use in ray tracing, one for the radiation model.

Modular Internode extends visible organ ==> F(length,width). (setShader(new AlgorithmSwitchShader(new RGBAShader(R,G,B), GREEN, LightShader)));

The LightShader defines the reflectance, and transmittance of this object to different wavelengths.

const Phong LightShader = new Phong().(setDiffuse(new RGBColor(reflectancePAR, reflectanceRed, reflectanceFarRed)), setDiffuseTransparency(new RGBColor(transmittancePAR, transmittanceRed, transmittanceFarRed)));

**SideSwitchShader** chooses among two shaders, one for the front side of a surface, the other for the back side.

**Examples Micha.**

**Repaint** can update the 3D view when the model stops. By using this method, we can switch the visualizing the 3D view based on the properties that we are interested, e.g. the distribution of PAR absorption, red to far red ratio or water potential within the canopy. The following code can switch the view between the default view and the distribution of red to far red ratio within the canopy, given the condition that there is a corresponding shader for ‘sRFR == true’ in the instantiation rule.

public void swColor() {

de.grogra.imp3d.View3D view = de.grogra.imp3d.View3D.getDefaultView(workbench());

if (sRFR == true) {sRFR = false;} else {sRFR = true;}

view.repaint();

}

**disablePaint and enablePaint** can turn off and turn on the 3D view, respectively. Turn off the 3D view can accelerate the simulation speed. This is useful when we only want to see the views of certain time steps, e.g. when we did the measurement.

**Examples Micha.**

**makeSnapshot** captures the 3D view and outputs it to the path we define. The outputted pictures can be further used to make animations.

makeSnapshot(path + time+".png"); // implement in the model running control function

**TextLabel** can give some interpretation and label at certain objects, e.g. TextLabel(plantNumber)

**I.4 Data input, output and visualization**

**readFile** can load external files into the model, and give the value to an array. The code of this function can be downloaded at:

<http://wwwuser.gwdg.de/~groimp/grogra.de/gallery/Technics/OpenFileDemo.zip>

**export** can export the dataset into a desired path.

const DatasetRef laidata = new DatasetRef("Leaf area index"); //define a dataset

laidata.(setColumnKey(0,"LAI")); //set the column key

laidata.getRow(time).set(0,lai); //give the value, in the place where you can access leaf area index

import java.io.FileWriter;

laidata.export (new FileWriter("/home/Downloads/test.txt"), ",");

**Chart** visualizes the dataset that we built up in the model. Just add chart(laidata, XY\_PLOT) at the model initialization, a chart panel will appears. XY\_PLOT means plot the values against the simulation steps with lines. More plot types, e.g. scatter, histogram can be downloaded at <http://wwwuser.gwdg.de/~groimp/grogra.de/gallery/Technics/ChartsDemo2.gsz>

Further information about the Chart can be found at JFreeChart <http://www.jfree.org/>.

**Process control** you can schedule the running of model whatever you want. You can make it run continuously, stop at certain time, or repeat the whole simulation for several times.

public void run()

{ update(); //update organ information

develop(); //organ initiation or removing

makeSnapshot(path +dayTotal+".png");

if (dayTotal > end) {stop(); System.gc();}

}

Furthermore, using apply() can accelerate the simulation speed as it does not updating the views.

public void runEnd() {

end = 100; //time steps

for (apply (end)) run(); exportData(); System.gc();

}

**I.5 Aggregate methods**

Aggregate methods are very useful for summarizing certain properties, e.g. number of leaves, whole plant light absorption. The following methods can be applied for a sequence of values:

* **array** array converts this into an array;
* **count** counts its number of values, long n = count( (\* pb:PlantBase, (pb.border == false) \*)); Note (\*…\*) is the syntax for queries, within which you can specify the conditions for finding a certain object.
* **empty** tests if it is empty, e.g. empty( (\* t:Seedling, ( (t != s) && (distance(s, t) <= 10) ) \*) );
* **first** and **last** return the first or last value,
* **max** and **min** the maximum or minimum value,
* **prod** and **sum** the product or the sum,
* **mean** the mean value for numeric types,
* **string** a string which contains a comma-separated list of the values, enclosed

in brackets.

Furthermore, first and last can be used to find objects.

Internode i = first((\* this (<--)+ Internode \*)); //this rule can implement within a module, e.g. leaf module for finding the closest internode

In addition, four selection operations are provided in GroIMP:

* **selectRandomly** selects one value out of all values based on a pseudorandom number generator. It chooses the value using a uniform distribution. Furthermore, one can also specify the relative probability of choosing the corresponding value (e. g. selectRandomly(x = nodes(), prob(x))
* **selectWhere** can specify the object to select, e.g. GridPoint A = selectWhere((\* y:gridPoint\*), location(y).z == 0);
* **selectWhereMax** and **selectWhereMin** can select the object which has the maximum value or minimum value of a given condition, e.g. GridPoint B = selectWhereMin((\* y:gridPoint\*), distance(this, y));

**I.6 List of most used turtle commands and simple math:**

Classical turtle commands from RGG tutorial chapter 2

|  |  |  |
| --- | --- | --- |
| **Turtle command** | **Meaning** | **GROGRA counterpart** |
| F(x) | Draw a cylindrical segment of length x. The cylinder is oriented along the local z-axis (the turtle's head axis). | F(x) |
| M(x) | Move along the local z-axis (the turtle's head axis) by a distance x. | f(x) |
| RL(a) | Rotate about the local x-axis (the turtle's left axis) by a degrees. | RL(a) |
| RU(a) | Rotate about the local y-axis (the turtle's up axis) by a degrees. | RU(a) |
| RH(a) | Rotate about the local z-axis (the turtle's head axis) by a degrees. | RH(a) |
| RG | Rotate towards the negative global z-direction such that subsequent turtle movement will be strictly downwards. | RG |
| RV(x) | Rotate towards the negative global z-direction by an amount which depends on the current orientation and the value x. This implements gravitropism, its strength being given by x. | RV(x) |
| Translate(x, y, z) | translation by (x, y, z) | Translate(x, y, z) |
| Rotate(x, y, z) | rotation by (x, y, z) degrees (local coordinate system is  rotated around the x-axis first, then around the y-axis  and finally around the z-axis) | Rotate(x, y, z) |
| RD(v,e) | directional tropism towards direction v with strength e | RD(v,e) |
| RN(n, e) | positional tropism towards location of node n with  strength e | RN(n, e) |

Math.min(), Math.max(), Math.exp(), Math.pow(), Math.sqrt(), random(min, max), normal(mean, sigma standard deviation), boolean probability(),

:’= ordinary differential equation solve notation, e.g. s[concentration] :’= -r;

**I.7 Java naming conventions**

The main goal of the conventions is to improve readability and thereby the understanding and the maintainability and general quality of the code. It is impossible to cover all the specific cases in a general guide and the programmer should be flexible.

1. Class (Module) names start with upper case

Module Internode extends visible organ

2. Variable names must be in mixed case starting with lower case.

line, audioSystem

3. Names representing constants (final variables) must be all uppercase using underscore to separate words.

MAX\_ITERATIONS, COLOR\_RED

In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice:

int getMaxIterations() // NOT: MAX\_ITERATIONS = 25

{ return 25; }

This form is both easier to read, and it ensures a uniform interface towards class values.

4. Names representing methods must be verbs and written in mixed case starting with lower case.

getName(), computeTotalWidth()

5. Abbreviations and acronyms should not be uppercase when used as name.

exportHtmlSource(); // NOT: exportHTMLSource();

openDvdPlayer(); // NOT: openDVDPlayer();

6. Generic variables should have the same name as their type.

void connect(Database database) // NOT: void connect(Database db)

Reduce complexity by reducing the number of terms and names used. Also makes it easy to deduce the type given a variable name only.

7. The terms get/set must be used where an attribute is accessed directly.

employee.getName();

matrix.getElement(2, 4);

8. is prefix should be used for boolean variables and methods.

isSet, isVisible, isFinished, isFound, isOpen

There are a few alternatives to the is prefix that fits better in some situations. These are has, can and should prefixes:

boolean hasLicense();

boolean canEvaluate();

boolean shouldAbort = false;

9. The term compute or calc can be used in methods where something is computed.

valueSet.calcAverage();

matrix.computeInverse()

10. The term find can be used in methods where something is looked up.

vertex.findNearestVertex();

matrix.findSmallestElement();

Give the reader the immediate clue that this is a simple look up method with a minimum of computations involved. Consistent use of the term enhances readability.

See more styles at <https://google-styleguide.googlecode.com/svn/trunk/javaguide.html>