Teaching Mathematics and Statistics in Sciences - IPA HU-SRB/0901/221/088
Interesting Mathematical Problems in Sciences and Everyday Life - 2011

Maths in modeling astronomical phenomena

Mirjana Rakić

Department of Mathematics and Informatics, University of Novi Sad, Serbia
Trg Dositeja Obradovića 4, 21000 Novi Sad
mirjana.rakic@dmi.uns.ac.rs
http://sites.dmi.rs/personal/rakicm/

Solar system has been widely studied over the past centuries and even more with the development of new technologies. It is so, since astronomical phenomena are present in everyday life of every individual. Therefore, we have developed a model of a solar system written in Java 3D to represent tits behavior. Mathematics has an extremely important role in modeling translation, rotation, revolution as well as zooming the whole system.In this paper it will be shown how this can be done and will be given the examples.

1. Introduction

Computer graphics has been greatly influenced by the work and results of mathematicians and artists during the years. One of the first founder is considered to be Euclid (around 300 BC) whose formulation of geometry represents the basis of graphics concepts. Also, an Italian architect and a sculptor, Filippo Brunelleschi (1377 - 1446), is famous for his usage of perspective, which is an important part of modeling.

Figure 1: Euclid and Brunelleschi [8], [9]

Moreover, Rene Descartés (1596 - 1650) should be mentioned here for his development of analytic geometry, especially coordinate system which is extremely important when describing the location and the shape of objects in space.

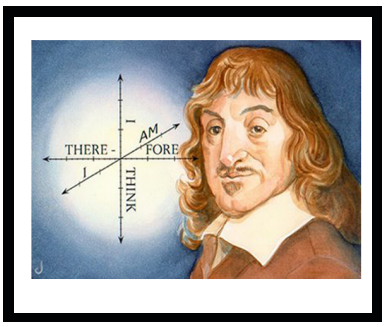

Figure 2: Rene Descartés [10]

Gottfried Wilhelm Leibniz (1646 - 1716) and Sir Isaac Newton (1642 - 1727) have co - invented calculus that allow us to describe dynamical systems.

Figure 3: Wilhelm Leibniz and Sir Isaac Newton [12], [11]

Very important role is of James Joseph Sylvester (1814 - 1897) who invented matrix notation. Matrices are widely used in various transformations of objects in graphics. Lastly, Isaac Schoenberg is famous for discovering splines, a fundamental type of curve.

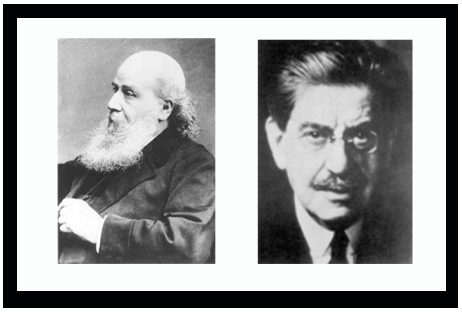

Figure 4: James Joseph Sylvester and Isaac Schoenberg [8], [13]

In this chapter, attention is put on the modeling an application that simulates behavior of the solar system and the role that mathematical concepts play in this modeling. This chapter is divided into 8 sections, each describing a different aspect of the model.

2. Java 3D

Java 3D is an **API** (**A**pplication **P**rogramming **I**nterface) that allows using object oriented approach in creating 3D models. It is an extension of Java standard libraries, to support 3D programming. There are around 29 standard packages with loads of classes in them for enabling the wide variety of possibilities for programmers.

Java platform is both platform and a programming language. Java is so special because it uses a virtual machine to interpret its code, so it is completely independent of operating system. A programme written in Java is compiled into *.class* files and can be interpreted on any virtual machine. This implies that a virtual machine is crucial for Java programmes. The advantage of this lies in the fact that a programmer does not have to worry about the specific hardware on which the application will be used.

As mentioned before, Java enables object oriented approach to programming. It enables encapsulation of the object, inheritance and polymorphism. This is very useful, since a programmer can create a class with some properties and methods and then use it again with various extensions, without having to write the same code again. This is widely used in this application.

3. The model of the solar system

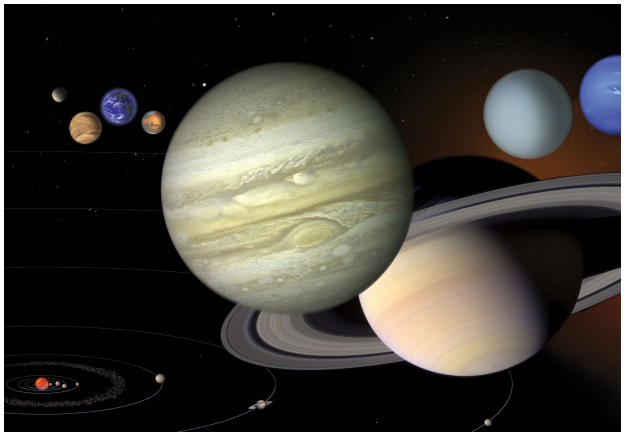

Figure 5: Our Solar System [5]

The solar system has been modeled in Java 3D in such a manner that it resembles the real one. That in particular means that the real data has been collected from the NASA web site, including radii of the planets, distances, texture, length of the day, length of the year, number of satellites etc., and used to get the real picture. The data gathered was the basis of the model. The Sun is in the center of the system and eight planets of the solar system together with a dwarf planet, Pluto, are around it. Some of them have been created together with their satellites, although not all of them. Each planet has accompanying text stating its name. It is 3D text that follows planets on their paths. The planets visually look the same as they can be seen via telescope. This is possible because the texture is applied to the planets.

Planets are in the basis spheres that have some certain properties. They can rotate around themselves, as well as around the Sun itself. The speed of the rotation depends on the length of the day, and the length of the year, respectively. The planets are rotating around the Sun via colored paths made especially for that purpose. Moreover, the inclination is modeled for the planets that are inclined.

The model has possibility to be zoomed in and out. Also, one can select a particular planet and zoom just that planet, which brings it in the center of observation. Also, the whole system can be accelerated. Furthermore, the model allows itself to be moved upside down. One can simply select an object by using left mouse button, and then rotate the system. Or, if one selects an object in the system using right mouse button, the whole system can be translated.

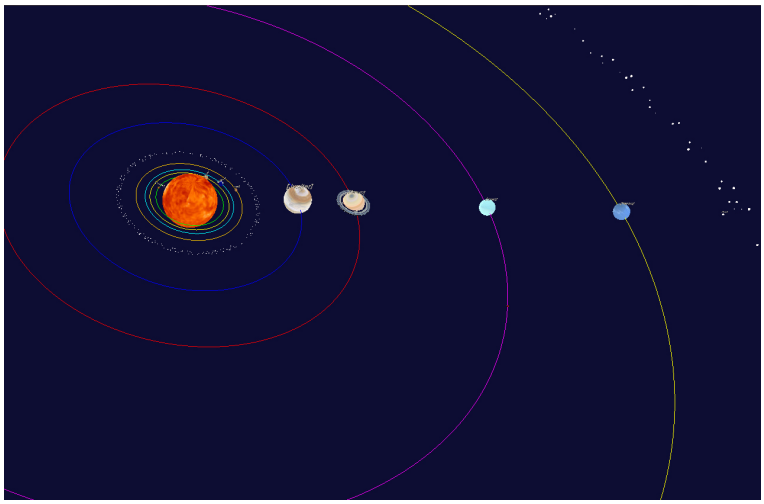

Figure 6: The model of the Solar System in Java 3D

4. Inside the solar system

In this section, we'll describe how the model of the solar system is made. This includes mentioning some of the Java classes used for modeling, as well as explaining how mathematics is connected with the whole model. The model of the solar system has been created on a 3D canvas within a class *Simple Universe*.

3D canvas is created based on a preferred configuration obtained from the user system. It is used as a "drawing panel" for the virtual objects. This canvas plays an important role in rendering. Namely, it is possible to do both *on-screen* and *off-screen* rendering. During on-screen rendering, Java constantly renders all existing on-screen canvases. It is useful in models like solar system, where such a rendering doesn't use too much memory and other resources, because in this model there is only one canvas. When there are many canvases and also many different perspectives of observation like, for example, two or multi player game, where each player has a different appearance, off-screen rendering is a much better choice. During off-screen rendering, the process of rendering is done before running the application. This saves resources to the great extend.

4.1. Virtual Universe

When the class *SimpleUniverse* is created everything except the canvas is set on the default values. One of the parameters is *ViewingPlatform*, which enables user to observe the system. The object of this class enables multiple transformations to be linked together. For this purpose, *ViewingPlatform* is set to its nominal transformation, which means that the objects are moved along z - axis allowing objects that are placed along x - axis in the interval between -1.0 and 1.0 to be visible on the screen. This is possible by setting a translation of $\frac{1}{\tan(fieldOfView/2)}$. The field *fieldOfView* by default gets the value $\frac{\pi}{4}$.

When the *SimpleUniverse* is created and the *ViewingPlatform* is set, it remains to say something more about how model is made in Java 3D. All the objects of the model, that are created, are added to *scene graph*. The objects are called *nodes*. Scene graph is a rooted tree in graph theoretical sense. No cycles are allowed, which means that the scene graph is a directed acyclic graph. There can be more scene graphs in one model, however, in the model of the solar system, there is only one scene graph. When a scene graph is created, it is added to a virtual universe, in our case, to the object of class *SimpleUniverse*. When added to virtual universe, the scene graph is said to be alive. This implies that the whole graph will be rendered together with all containing nodes. Nodes are divided into two classes - group and leaf nodes. Group nodes have a parent and can have none, one or more child nodes, whilst leaf nodes have only a parent, but have no children.

Nodes are very important because they set the bounds of activities of the objects. Significant is the class *Bounds* that inherits *Node* for this reason, since it defines a volume in space. It actually specifies the region of some action. For example, when the rotation of an object is modeled, then we have to define the region, or the space of rotation. This precisely is done using bounds. The bounds of activity are necessary as a frame within which the scene graph would be constantly refreshed, so that rotation would be visible. The bounds of an object can be in the shape of a sphere - *BoundingSphere*, a box - *BoundingBox* or a set of planes enclosing a space - *BoundingPolytope*. In this model, *BoundingSphere* is used. The reason for that lies in the fact that all the planets, and Sun as well, are represented as spheres, naturally implying the usage of a *BoundingSphere* as bound for their actions. Also appearance and lightning can be applied within the bounds of an object.
Mathematics is pretty involved when it comes to bounds. It is very important to distinguish between different objects, as well as to know the regions of actions of objects and whether or not there is some interference between objects. Class *BoundingSphere* has predefined methods for determining intersection of given objects, based on calculating distances between given objects. It is enough just to call one method of the object to obtain logical result stating whether or not two bounding spheres intersect.

4.2. Grouping

None of the objects in any world, and also in virtual, have any meaning if it is considered alone. Objects have some meaning only when they are considered in interaction with other objects from that world and as a part of it. That is the reason why grouping is so important. Group nodes are very significant in this model because of their role in managing objects. There are many types of groups, but for our model, the most important ones are *BranchGroup* and *TransformGroup*.

BranchGroup plays a significant role in virtual universe. The object representing the root of the scene graph must be object from this class. The reason for that lies in the fact that object of this kind of group can be easily detached from the parent at any point dynamically. Moreover, the objects of this class are the only ones that are the members of the set of objects of the *Locale*, whose origin defines the coordinate system in the virtual world of that *Locale* given in double precision coordinates. An object of the *BranchGroup* can be easily extended with object of other groups.

TransformGroup enables wide variety of transformations to be done with objects. In order to make transformations of object in space possible, a knowledge of mathematics is required. Transformations set within an object of this group can be also transferred to all the children of that object. Without *TransformGroup* all the object of the scene graph would be static, placed at point (0,0,0) of the coordinate system, without any possibility to rotate, or to be scaled. The transformations specified by this group are mostly affine, in order to define rotation, translation and uniform scaling which is used in the model of the solar system. They are used since they preserve ratios of distances along the line and collinearity of the points. It is not difficult to verify whether or not a transformation is affine. To be able to create a transformation group, one has to set the transformation first. This is done by defining an object of *Transform3D* class, which creates a real 4 - by - 4 matrix, representing the transformation. By default, the matrix created is identity matrix, and the object created in that way is placed in the center of the universe, at point (0,0,0). The object of this class can be translated by translation vector to change its position. Also, it can be rotated about x, y or z - axis. In the model of the solar system, as well as in the real one, some of the planets

have inclined orbit around the Sun, because they have angle greater than zero to the ecliptic plane. For example, the Earth is inclined by the angle of 23° 45'. This is modeled as follows: Firstly, a transformation representing the Earth is created, then it was moved from the center of coordinate system, using translation vector, to the position where it should be in the solar system. Afterwards, the transformation is rotated around z - axis to be inclined.

For this purpose, it has been necessary to implement some basic calculations to make aforementioned actions possible. It is useful to mention here how the angle of rotation is calculated. Since the data about the Sun and each of the planets are stored in separate java file, since each represent an independent object, the angle is stored in degrees, but Java works with radians. Thus, the degrees were to be recalculated in radians. The code snippet representing the set of actions necessary to simulate translation and inclination is given here:

```
t3d = new Transform3D();
t3d.setTranslation(vektorPolozaja);
// a vector for the translation
tg = new TransformGroup(t3d);
Transform3D multiplier = new Transform3D();
if (equat != 0.0){
double deo = 180.0 / equat;
double rot = Math.PI / deo;
multiplier.rotZ(rot);
}
t3d.mul(multiplier);
tg.setTransform(t3d);
```

The previous code also shows how transformations and transformation groups are created and how they are connected. When the code is applied to the Earth and the Saturn, it results in the following.

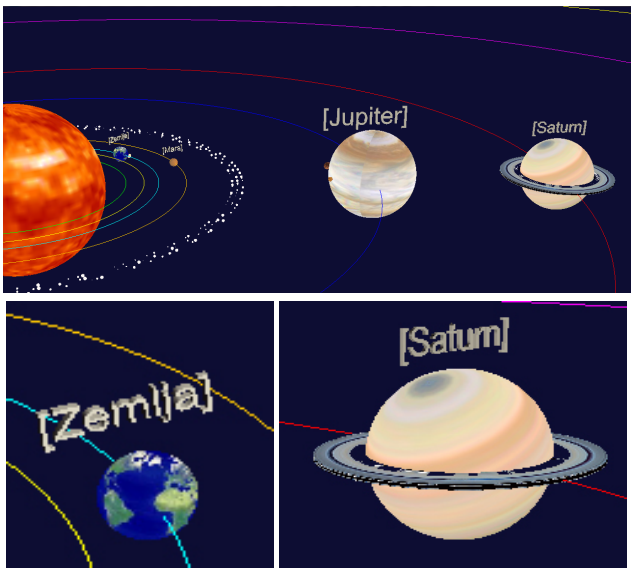

Figure 7: The Earth's and the Saturn's translations and inclinations

4.3. Transformations and maths

Let us say something more about what was used from the predefined methods in the code and what is in the background of it. The *t3d* transformation was firstly translated by a 3 - dimensional vector, which in mathematical

sense means that first, second and third row of the last column of matrix representing *t3d* was replaced by x, y and z coordinates of the vector. The *multiplier* is a *Transform3D* object that rotates around z - axis. The method *rotZ* that is called in the code to enable rotation around z - axis, changes some entries in the matrix that represents *multiplier*. Namely, rotation is simulated by taking sine and cosine of the angle around which we want to rotate the object and these values are inserted in the matrix on specific positions. So, after the rotation of *multiplier* around z - axis, an affine transformation is obtained. It is necessary to make the composition of two transformations to obtain a transformation that is both translated and rotated. This is achieved by multiplying matrices that represent transformations *t3d* and *multiplier* in the classical way. When a transformation is multiplied with an affine transformation the result is an affine transformation as well. After making all the necessary changes, transformation *t3d* is set to be the transformation of a *TransformGroup tg*.

It is of great importance that a transformation group has an ability to read and write transformation while simulating the operation of the solar system. The reason for that lies in the fact that without constant refreshment of the parameters necessary for some transformation, it wouldn't be possible to make changes to it. This, in particular, means that rotation of the planet around itself and around the Sun, and rotation in general wouldn't be possible in this model if transformations wouldn't have this ability. While rotating, parameters of matrix that represent a transformation have to change, since the position of the planet (its coordinates) has to be changed. It was already mentioned in the previous paragraph how translation and rotation of the object were made. When constantly rotating, the aforementioned actions have to be repeated, and writing new entries in the matrix and reading previous state is important.

Beside multiplication of transformations, as well as the ability to read and write transformations during the simulation, an important thing is also adding a transformation into another transformation. This very possibility allows creating the dynamics of the system, and adding additional data to the planets, e.g. text over the planet stating its name. Moreover, one of the most important properties of the model of the solar system, its movement, is obtained by creating transformations and adding one into another. In order to be able to set that every planet rotates around the Sun, the transformation, representing it, is created and set to be in the center of the system. Since it has the property to rotate around itself, this property is created. More about rotation, and how it is implemented, will be said in the next sections. Now, the accent is put on the combination of the groups of transformation. Each planet has its own properties, text stating its name, the path of rotation and maybe one or more satellites following it on its path. Having all that in mind, natural question is how to combine all that into one connected system, and the natural answer is to combine transformations.

The following figure shows the close view of the Sun and some of the neighboring planets.

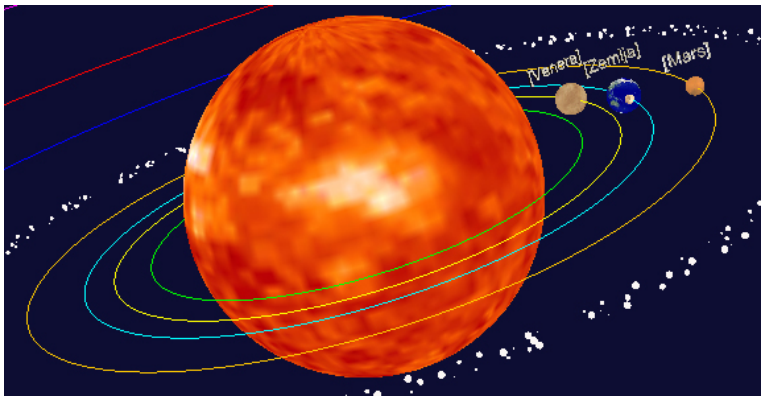

Figure 8: Rotation of the Sun and the neighboring planets

It has been already spoken about position of the planet in solar system and its inclination. The 3D text is added to the transformation representing a planet just as a transformation. And a satellite of a planet is modeled in the same

way a planet is created and then added to the planet by adding the whole *TransformGroup* of the satellite to the *TransformGroup* of the planet. The same procedure is connecting the planet to the Sun, to enable rotation around it. The *TransformGroup* of the planet is added to the *TransformGroup* of the Sun.

The model of the solar system can be represented graphically, as mentioned before. In it, the *BranchGroup*, which is the root of the tree, is extended with the object of transform group containing the background, lights, and many other objects of transform groups representing the Sun and all the planets etc. Also, some other objects can be seen in the picture, e.g. *Interpolators*, but it will be spoken about them in the next sections. Since many objects exist in the system, only one part of it is depicted.

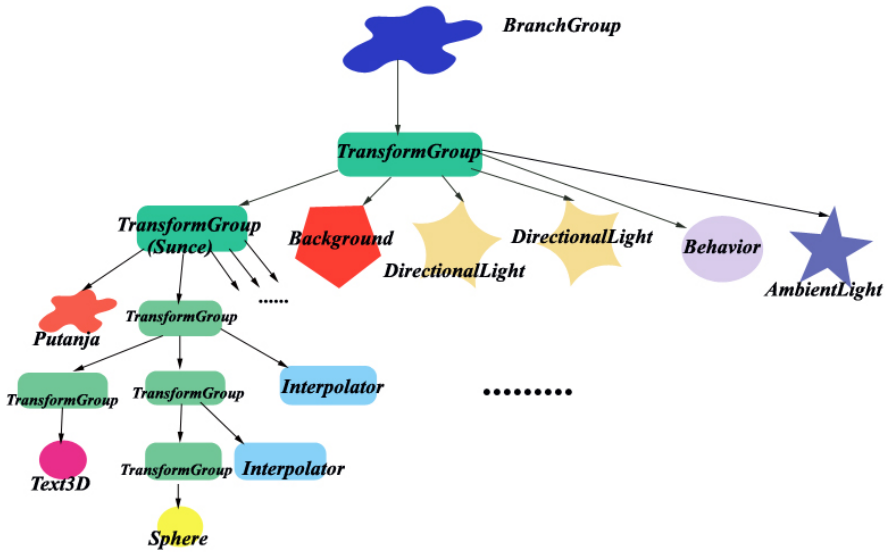

Figure 9: Graphical representation of a part of the model of the solar system

5. Planets and their paths

It can be seen in the model that the planets are moving around the Sun following a specific path. In the real Solar system, paths of rotations are elliptical, but for simplicity reasons, they were modeled as cyclic. The paths are also colored.

In order to model this paths, some knowledge of analysis is required. Namely, the paths are drawn, point by point, using trigonometry. Actually, a line consisting of these points represents a path. Every point is given in 3 dimensional space, since the model as a whole is 3 dimensional, by setting its coordinates. The y - coordinate is fixed, it is 0.0, since all the point of the circle are given in the plane where $y = 0$. Now, the question remains how to calculate the 360 points of the circle representing the path of one planet. There are 360 different points all together, for a circle has 360 degrees. Each planet and its path are set on the same distance from the Sun (or the center of coordinate system). So, having this in mind, it is necessary to specify the way of calculating the points of the circle with given radius.

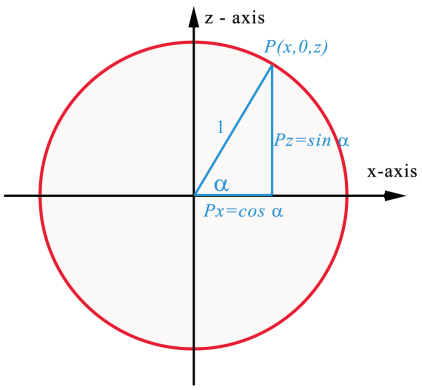

Figure 10: Trigonometric functions on unit circle

The x - coordinate of every point p of the circle is calculated as the product of radius and the cosine of the angle between the points $p1 = (radius, 0.0, 0.0)$ and p. Analogously, the z - coordinate of some point p is calculated as the product of radius and the sine of the angle between the points $p1 = (radius, 0.0, 0.0)$ and p. Note that the first and the last point are the same, meaning that there are 361 points. This is because we need to close the line and not to leave any free space. Given in the code, it looks as following.

```
LineStripArray lsa = new LineStripArray(length,
GeometryArray.COORDINATES,
new int[] {length});


// first and last points
Point3f pt0 = new Point3f(radius, 0.0f, 0.0f);
lsa.setCoordinate(0, pt0);
// computed points
Point3f pt = new Point3f();
for (int i = 1; i < 360; i++) {
pt.x = (float) (radius * Math.cos(i * Math.PI / 180));
pt.z = (float) (radius * Math.sin(i * Math.PI / 180));
lsa.setCoordinate(i, pt);
}
lsa.setCoordinate(360, pt0);
circle = new Shape3D(lsa);
```

In order to create circle it was necessary to use *Shape3D* object. Object of this class is convenient for this purpose, because it can contain geometry. *LineStripArray* is an array of coordinates constituting a line. Since this class is an ancestor of *GeometryArray*, it can be easily added to the object of class *Shape3D*.

6. Rotation

It was already mentioned before, that without having the ability to rotate, the Sun, all the planets and their followers would just stand in one position and could not move. Thus, it is of utmost importance for the model to enable movement.

Rotation is a specific behavior. Thus, we should mention some of the properties of a class *Behavior* which is the base class for representing all types of behavior. First of all, behavior has to start at some point, which is

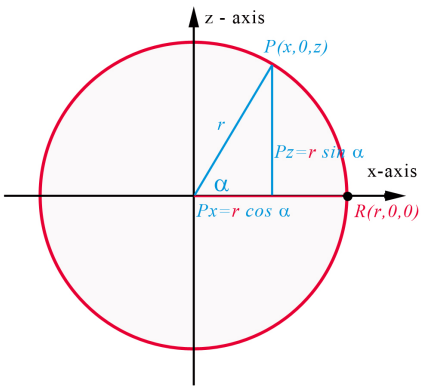

Figure 11: Trigonometric functions on a circle of radius r

called initialization. At this point, a *WakeupCondition* is set. A *WakeupCondition* consists of a set of conditions that are significant for the dynamics of the system. These can be seen as signals for making some changes in the system. Initialization is done only once, when the object of *BranchGroup* is added to virtual universe. Apart from initialization of a behavior, there is a so called process stimulus, that makes the changes to the system, when some wake up criterion is satisfied. This can be, for example, when a mouse key is pressed. In our model this action exactly is used for obtaining the data about a planet on which the user has clicked, or for zooming, about which it will be spoken later. Furthermore, the behavior has bounds within some object is active. In our model, it is a *BoundingSphere* because we rotate spheres. The object is only active when its bounds are within the bounds of the behavior set for this object. This implies that during the behavior, the intersection of the aforementioned bounds is checked on regular basis. The interesection is calculated as described in section 4.

The most important role in the movement plays the class *RotationInterpolator*. The object of this class is a linear interpolator between two given endpoints of interpolation. The endpoints are angles given in radians. If they are not defined, minimal and maximal angles get the default values of 0 and 2π, respectively. This interpolation simulates rotational behavior about y - axis. It is used in the model for simulations of rotation of a planet or a satellite about itself and about some other objects. Time, as a parameter, is used to control the interpolation. In Java 3D, this time parameter is given by the class *Alpha*.

Alpha is a function that transforms time into an alpha object which can take values from the interval $[0.0, 1.0]$. It is given by $f(t) \rightarrow [0.0, 1.0]$. This object can be called finitely or infinitely many times, depending on the requirements of the model. Since in our model rotations are constantly performed, alpha is run infinitely many times. Because we simulate rotation, we want alpha to go from 0.0 to 1.0. This would resemble a period of rotation. So, we have to set the trigger time (in milliseconds) when alpha is about to start again. In our model, when it is about rotation of a planet around itself, a perimeter of a planet determines when alpha is about to start again.

7. Zooming

Zooming, translation and rotation of the virtual universe as a whole is possible thanks to the class *OrbitBehavior*. Orbit behavior is a behavior that is applied to View platform that was mentioned at the beginning of the chapter.

What is more, the zooming that exists in this model of the solar system is modified in order to enable placing the object that is zoomed in or out in the center of the observation. The motivation for that lies in the fact that is is more interesting to see the object that is zoomed in as a central figure on the canvas, than to search for it all over the screen. This is done by setting the object in the center of observation when it was clicked on it by mouse. Note

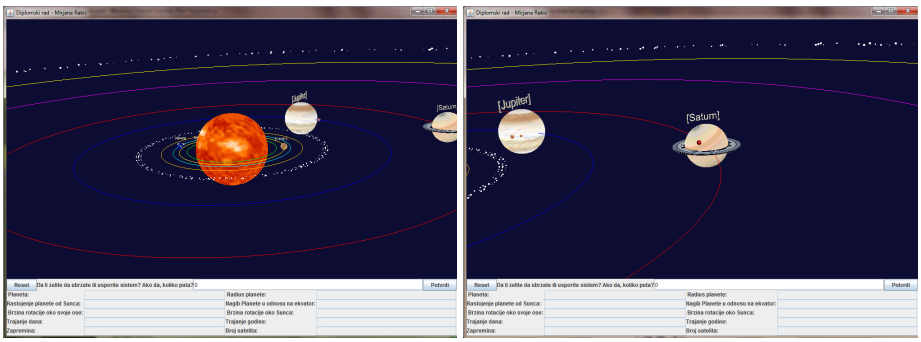

Figure 12: Zooming in the Saturn

that this is done only once, since there is no need to constantly move the center of the system during the process of zooming.

The method that does zooming in the class *OrbitBehavior* is modified to support this requirement. First, the x and y - coordinates of the center of the model are found. The translation of the system will be made according to the difference between the these two coordinates and x and y - coordinates of the selected point. In the code, the selection points are named *tackaX* (the x - coordinate) and *tackaY* (the y - coordinate). The code is given below:

```
if(zoom(mouseevent)){
if (mozeTrans){ //if we have placed an object
//into center of the system,
//we won't move the picture any more
distx = (int) (centarX - tackaX);//the distance by x-axis
disty = (int) (centarY - tackaY);//the distance by y-axis
xtrans -= (double) distx * transXMul * 0.1;
ytrans += (double) disty * transYMul * 0.1;
mozeTrans = false;
}
}
```

What is done in the code is the following: the distances by x and y - axis are found, marked with $distx$ and $disty$ respectively. After that the variables $xtrans$ and $ytrans$, representing translation by x and y - axis, are decreased (and increased) by the product of the distances, multiplier for the translation and factor 0.1 for the accuracy of translation. The variable *mozetrans* is the one enabling centering of the selected object only once.

8. Conclusion

Computer graphics is closely related to and influenced by mathematics. Mathematics is in the basis of every model in virtual world. Although there are many methods and calls enabling an application of built - in mathematical formulae, sometimes it is also necessary to implement something new that will satisfy our needs.

The application representing the model of the solar system is created in educational and scientific purposes. In the model mathematics is mostly used for rotations, translations and zooming, and also for some other transformations. In this chapter is given the application of maths to the model in combination with explanations how it is used in

the world of programming. Also, some examples of the code in Java3D are given in text, illustrating application of maths in the model.

References

[1] Rakić, M., **Modelling astronomical phenomena using Java 3D**, graduation thesis, 2007.

[2] Selman, D., **Java 3D Programming**, Manning Publications, Reprint Edition, February 2002.

[3] Zhang, H., Liang, Y. D.,**Computer Graphics Using Java 2D and 3D**, Prentice Hall, December 06, 2006.

[4] http://cs.fit.edu/ wds/classes/graphics/History/history/history.html

[5] http://sse.jpl.nasa.gov/planets

[6] Java 3D API - documentation

[7] www.java3d.org

[8] www.nndb.com/

[9] http://en.structurae.de/

[10] http://tetrahedral.blogspot.com/2010/12/rutgers-phil-proves-descartes-wrong.html

[11] www.brisbanetimes.com.au

[12] http://fermatslasttheorem.blogspot.com/2007/10/gottfried-wilhelm-leibniz.html

[13] http://www.diac.upm.es/accesoprofesores/asignaturas/television/stv/historiatv/dvbtv-history.htm

Contents

1. Introduction **1**

2. Java 3D **3**

3. The model of the solar system **3**

4. Inside the solar system **4**

4.1. Virtual Universe 4

4.2. Grouping 5

4.3. Transformations and maths 6

5. Planets and their paths **8**

6. Rotation **9**

7. Zooming **10**

8. Conclusion **11**