Università La Sapienza

Interactive Graphics

Final Project

# Solar System Simulation

*Authors:*
Luca Di Giammarino
Alessio Tomaselli
Alessandro Santopaolo

*Supervisor:*
Prof. Marco Shaerf

October 2, 2018

# Contents

# 1 Introduction

This project is a real simulation of a Solar System. It underlines the importance of a wide range of techniques used in Computer Graphics and the optimization solution offered by C/C++ and OpenGL 2.1 which allow to emulates 200 planets of small masses orbiting a star of a bigger mass.

Being a real simulation it focuses more on the motion of the planet that on the rendering itself. In fact, graphically speaking only GL primitives have been used. It has been decided OpenGL in place of WebGL, first for the computational efficiency of C/C++ and second to learn a more complete and hard graphical environment.

# 2 Development Environment

Few tools have been used for the development of the project. The language preferred has been C/C++ with Visual Studio 2017 as interactive development environment. The OpenGL version used is the 2.1. From this perspective the project is very simple in the sense that all the implementation it is related on the basics of OpenGL and its primitives.

## 2.1 External libraries

The external libraries used to deal with *algebraic* and *mathematical* operations are the following:

- raaMaths
- raaMatrix
- raaVector

An additional fundamental library used to deal with all the *camera* operations in OpenGL was:

- raaCamera

Moreover, to help including a texture it has been used the following library:

- imageLoader

# 3 Technical Aspects

This project uses the basis concepts of OpenGL and the optimized data structures implemented in C/C++ in order to build a complicated simulation of a Solar System. This section list the technical design and ideas used in the project.

## 3.1 Data structures

The simulation includes a star of a bigger mass situated in the origin of the space and 200 planets of smaller mass which orbit the star.

In order to store the information for each planet it has been fundamental to define an extended generic single structure to encapsulate all possible parameters for rendering and simulation with additional functionality for occupancy of an advanced dynamic data structure and support for redefined rendering solutions. The data structure designed for this purpose is a **Doubly-linked List** [Figure 1]. Being a dynamic data structure this allowed to add and remove bodies during the simulation.

```
Node *createNewElement(float *m_position, float m_mass, float
    *m_pcolor, Data *registerPosition, int m_camera)
```

This constructor call represents the creation of a new node to be added in the linked list. Every node represents a planet (a sphere in OpenGL) and it is stored as Node in the structure. In the fragment of code are highlighted the information stored for each planet.

The **Doubly-linked List** allows to write comfortably the information for each planet to a *text file*. This feature allows to store the Solar System state that can be restored by reading the text file. However, this last restoring feature it has not been implemented due to lack of time.

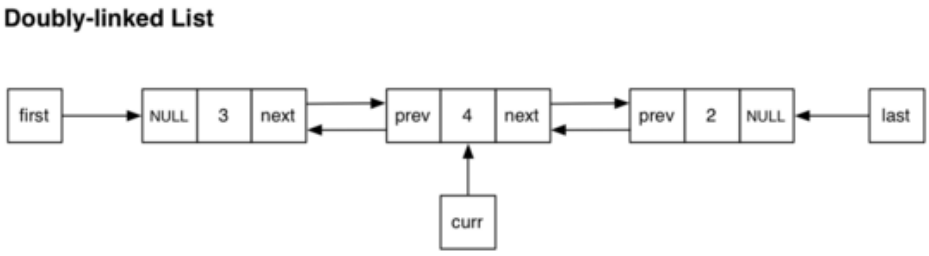


Figure 1: Doubly-linked list

On the other hand to draw the orbits using OpenGL an additional **Singly-linked List** [Figure 2] has been used. This has been important to store the history of the positions for each planet, allowing to draw perfect curved orbits. It is passed through the constructor when the element is created under the name of:

```
Data *registerPosition
```


Figure 2: Singly-linked list

## 3.2 Simulation components

In the simulation all the parameters of the planets have been initialized randomly. In order for the simulation to work, these parameters have been tuned and optimal ranges have been found. This implies less hard coded variables and a more flexible and readable code. For instance, the planet *position* ($x$, $z$ coordinates) [Figure 3] can be a value in the range showed by [Eq. 1] where $p$ is the planet position and $M$ is the star mass.

$$-30M^{1/3} < p < 30M^{1/3} \tag{1}$$

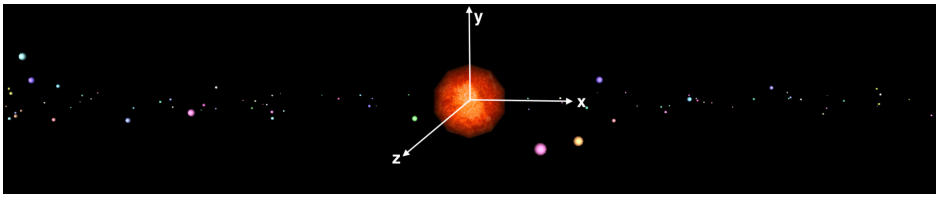


Figure 3: OpenGL Coordinate System

The same technique has been used to generate random *colors* and *mass*. The ideal mass parameter (in reality is the size of the sphere) oscillates between 1 and 1000 *Kg*. The star conversely needs to have a mass 800 to 80000 times bigger. Hence, this has been set to 80000 *Kg*. Moreover, another parameter initialized randomly has been the *velocity*, this oscillates between 1 *m/s* and 100 *m/s*. Once the simulation is started this parameter is updated on-line through Eq. 2, where $v$ is the velocity, $s$ is the updated position, $p$ is the previous position and $t$ is the time.

$$v = \frac{(s-p)}{t} \tag{2}$$

Another important feature designed is the addition of bodies (planets) at runtime in response to user instruction, in order to maintain a preferred set of population and/or *performance*.

As stated previously, being a simulation, is based on the real dynamics forces and laws. First of all, the *Newton's First Law of Motion* saying that an object that is in motion will not change its velocity unless a force acts upon it. This is for each planet with each planet (*nested for loop*) and it is expressed through Eq. 3. In the equation $F$, is the force calculated for the first planet, *m1* is the mass of first planet, *m2* is the mass of the second planet and $l$ is their normalized distance.

$$F = 9.8\frac{(m1)(m2)}{l^2} \tag{3}$$

It is important to realize that $F$ becomes after, the sum of the forces for the first planet with each other planets $F_{tot} = \sum_{n=1}^{200} F_n$. It is noticeable how many times this computation needs to be done, being 200 the number

of planets, $200 \times 200$. Hence, 40000 each rendering function call. Since it is a very expensive process in terms of computation, it can be noticed that increasing the number of sides of the sphere or by replacing this object with a more complex one, the simulation slow down drastically. Therefore, it is fundamental to keep this kind of parameters.

After, using *Newton's Second Law of Motion* the acceleration $a$ can be calculated as in Eq. 4.

$$a = \frac{F_{tot}}{m} \tag{4}$$

Once this is computed the position for each planet can be updated using simply Eq. 2, with the updated position $s$ calculated from the *Equation of Motion* [Eq. 5]:

$$s = p + vt + \frac{at^2}{2} \tag{5}$$

In addition, a damping coefficient is added to take more control of the simulation and that the system becomes unstable. This can be controlled by the user. It is simply a small index that is multiplied with the velocity $v$. Thus, the user can decrease its size. By doing this stable energy level are achieved.

Furthermore, to make the simulation more realistic and advanced a collision detection system has been designed. By checking the coordinates of the planets, this can be simply achieved. If two planets collide, the major body is kept while the other is removed from the simulation (popped from the Doubly-linked list). More interesting, is the functionality behind a collision planet/star. In this case, the planet is removed and the star incorporate the mass of the planet.

## 3.3 Rendering and display

From the graphics perspective, the simulation have been kept simple in order to enhance the optimization of the simulation. In fact the only complicated object is a *car model* drawn with a mixture of OpenGL primitives. The rest

of the object is kept simple with *solid spheres* and *line strips* using retained *material* values and position to draw the bodies/orbits.

A *texture* has been applied to the star, the bigger element of the simulation. This is a simple sun texture and makes the simulation more realistic [Figure 4].

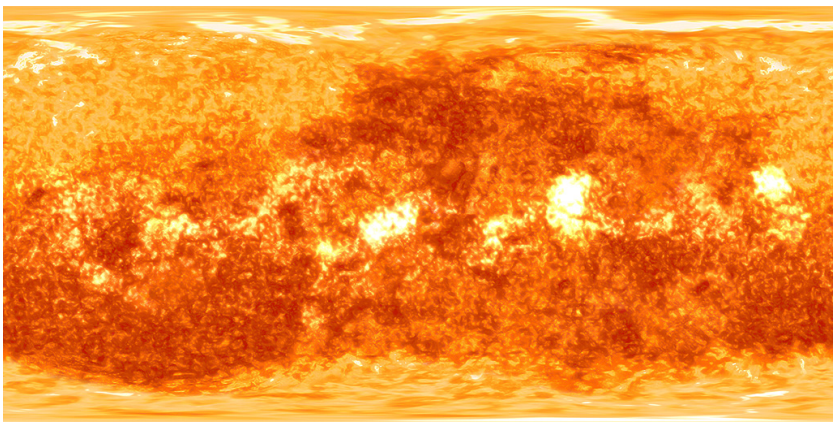

Figure 4: Sun-like texture

The light used is a *point light* in this way, the simulation seems more understandable since every single body get the same amount of lighting.

Additional rendering function have been implemented such as *fog*, this can be switched on and off simply by the user.

It is important to point out how the orbits are generated, this, as discussed before, encloses the history of the position for each planet and are store into the dynamic data structures. Hence, the rendering function needs to query the data structure to draw the lines using queried positional information.

Finally, as mentioned before the only complicated rendered object is a car model. This has been drawn from OpenGL primitives and it is in the simulation to underline the importance of the *hierarchical model*. This model, allows to control the car as a whole object, emulating for example the movement of the vehicle and fragment objects like the ignition system that is part

of the car, but does not represent the complete model. The ignition system in this case can be called *child* and in the simulation it is translated first independently by a little displacement and finally it is moved with the car when the user interact with it.

### 3.4 User interaction

In terms of user interaction, rendering window does not provide an interface and the user can interact only with the keyboard and the mouse.

First of all the car is completely controlled by the user, with this object that can rotate and translate in the space, he can explore the simulation. In fact, the camera can be locked to the car in order to change the point of view.

In addition, the user can switch to multiple modes of camera navigation. One of the most important thing is the exploration mode where the camera can be locked to a planet through mouse position detection. The detection through mouse pointer has been one of the most challenging features to implement and it is discussed in details in the implementation section.

As described in previous subsections the user can control other minor simulation parameters such as switch the fog on/off, increasing the fog density, decreasing the damping coefficient to stabilize the energy of the system, add and remove planets, etc etc.

## 4 Implementation

In this section the implementation details are discussed more in details.

### 4.1 Doubly-inked list

As discussed in previous sections, the Doubly-linked list has been implemented to benefit of a dynamic and efficient data structure. This implementation has the following interface:

```
struct Node {
   Node *m_pNext;
   Node *m_pLast;
```

```
    float m_position[4];
    float m_mass;
    float *m_pcolor;
    float m_velocity[4];
    Data *registerPosition;
    float m_camera;
};



extern Node *g_pHead;
extern Node *g_pTail;

Node *createNewElement(float *m_position, float m_mass, float
    *m_pcolor, Data *registerPosition, int m_camera);
Node *popHead();
Node *popTail();
Node *searchColor(float *m_pcolor);
Node *searchSize(float m_mass);

void pushToHead(Node *pElement);
void pushToTail(Node *pElement);
void insertBefore(Node *pElement, Node *pTarget);
void insertAfter(Node *pElement, Node *pTarget);
void printList();

bool destroy(Node *pElement);
bool remove(Node *pElement);
```

This data structure offers a very detailed interface, with functions to search planet details as well as basic functions to pop and push after and before a certain Node (planet). It is important to point out that C/C++ allows the user to manage the memory, thus the remove functionality implemented in the data structure are fundamental to keep the simulation stable, avoiding problems of slowness. This problem cannot be faced in Javascript and WebGL where objects created are deleted only when they get out from the scope. This process is called *garbage collection* and it is not really efficient. The field *Data *registerPosition* is the Singly-linked list stored for each planet in its own Doubly-linked list.

## 4.2 Simulation loop

The most important implementation detail to point out is the simulation main function, where forces are calculated and the position for each planet is updated. This is described below in terms of pseudo-code.

```
for each PLANET:
   if PLANET is not the star:
      initialize all the variables and arrays
         for each planet:
            calculate displacement between the two planets
            normalize displacement vector
            calculate scalar force from Newton I law of motion
            calculate vector force using scalar product
            add PLANET force to sum of forces


         calculate acceleration vector from Newton II law of motion
         updating position using the equation of motion
         updating velocity
         apply dumping coefficient to make simulation stable
         storing history position
```

As stated in previous section in this main function it is present a nested for loop, therefore each planet iterate through each other planet. This difference has been highlighted in the pseudo-code with the word *PLANET* (outer loop) and the word *planet* (inner loop)

### 4.2.1 Collision detection

The collision detection has been implemented in a way that detects collision through the comparison of coordinates of the planets. This works surprisingly well. The nested for loop below includes the collision detection logic:

```
for each PLANET:
   initialize vector collision and distance
   for each planet:
      if element and are different:
         calculate the distance between planet and PLANET
         calculate length of distance (normalize)
         if distance is smaller than planet or PLANET radius:
```

```
if body is not the sun:
    add mass to PLANET
else
    add mass to star
```

Figure 5: Car, star and planets in static conditions

## 4.3 Rendering with OpenGL

In this subsection are listed the main implementation rendering details using OpenGL.

### 4.3.1 Drawing bodies

As stated before, the bodies apart from the car are drawn using random materials and colors and random positions. These are all spheres composed by 10 slices each. More slices would increase the computational resources needed and would lead to slowing down the simulation. The following pseudo-code explains how the planets are rendered looping through the doubly-linked list.

```
for each planet
   push a new matrix
     if planet is not the star
         rotate planet
         translate planet
         give a color to the planet
         draw the planet through glutSolidSphere
     else planet is the star
     texture the star (see texture section)
```

```
      pop the matrix
```

An additional implementation detail that can be discussed is how the orbits (*lines*) are drawn. Being more technical this needs to be discussed more in depth. As it is visible from the code:

```
glPushAttrib(GL_ALL_ATTRIB_BITS);
glDisable(GL_LIGHTING);
glLineWidth(1.0f);
glColor3fv(pElement->m_pcolor);
glBegin(GL_LINE_STRIP);
for (Data *historyPosition = pElement->registerPosition;
    historyPosition; historyPosition =
    historyPosition->m_pNextData) {
   glVertex3fv(historyPosition->m_data_position);
}

glEnd();
glPopAttrib();
```

It needs to be said that the following code is included inside the for each planet loop, here every planet is called *pElement*. The first fragment is only the initialization of the line features through OpenGL. The loop in the code indicates simply the iteration through all the position history of every planet. Hence, the orbits are simply drawn in this way.

### 4.3.2 Texture

According to the project specification and to make the simulation more realistic, a texture has been added to the star. With the help of the imageLoader file, a library downloaded from the Internet the a sun-like texture has been wrapped on the star, the sphere with the bigger mass. The following fragment of code explains the OpenGL process:

```
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, _textureId);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
gluQuadricTexture(quad, 1);
gluSphere(quad, powf(pElement->m_mass, 0.33f), 10, 10);
```

This texture has obviously an impact in the GPU power required for the simulation. If all the planets were wrapped with a texture probably the simulation would not even run.

### 4.3.3 Drawing the car

The car is composed simply by primitives body in OpenGL and takes the advantages to be composed in hierarchical structure way. Hence it has been drawn with a mixture of objects translated, rotated and scaled in the space. These are:

- *gl quads* for the chassis and windows of the car
- *gl triangles* for the poster windows
- *gluCylinder* for the ignition system
- *gl line strip* for the wheels' rims
- *glutSolidTorus* for the tires

By using consistent colors the result is as like in Figure 6.

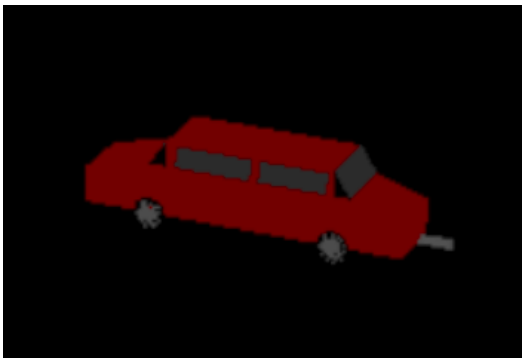

Figure 6: Hierarchical Car Model with OpenGL

## 4.4 User interaction

The user interaction implementation is discussed in the sections below. As mentioned previously an interface has not been developed since the goal of the project was to implement a simulation rather than focus on a more comfortable interaction from the user perspective.

### 4.4.1 Mouse interaction

The mouse interaction allows to select a planet and to lock the camera to it. This has been done converting the coordinates from the screen ($x$, $y$) to GL ($x$, $y$, $z$). This has been achieved through the following function that takes the mouse input coordinates in the screen (right click) and returns a vector with the OpenGL coordinates in 3D.

```
float *translatePositionFromScreenToGL(int x, int y)
{
   float vecPosition[3];
   //Viewport values holder
   int viewport[4];
   //Modelview matrix holder
   double modelview[16];
   //Projection matrix holder
   double projection[16];

   float winX, winY, winZ;
   double posX, posY, posZ;

   //retrieve the Modelview Matrix
   glGetDoublev(GL_MODELVIEW_MATRIX, modelview);
   //retrieve the Projection Matrix
   glGetDoublev(GL_PROJECTION_MATRIX, projection);
   //retrieve the Viewport Values (X, Y, Width, Height)
   glGetIntegerv(GL_VIEWPORT, viewport);

   //holds X mouse coordinates
   winX = (float)x;
   //subtract the corrent mouse Y coordinates from the screen
   winY = (float)viewport[3] - (float)y;
   //get Z coordinates
   glReadPixels(x, int(winY), 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT,
      &winZ);
   //converting screen coordinates to OpenGL coordinates
   gluUnProject(winX, winY, winZ, modelview, projection, viewport,
      &posX, &posY, &posZ);
   //set up vector to be returned
   vecPosition[0] = (float)posX;
```

```
   vecPosition[1] = (float)posY;
   vecPosition[2] = (float)posZ;
   //return vector having position X,Y,Z stored
   return vecPosition;
}
```

After, iterating through every planet and finding the corresponding position with

```
camExploreUpdateTarget(g_Camera, pElement->m_position);
```

it is possible to lock the camera relative to the selected planet. In addition, by using the mouse it is possible to set the move the camera position (always locked to the origin) and the use flying mode, where the camera is move by the user in any direction.

### 4.4.2 Keyboard interaction

The keyboard allows the user to interact in deep with the simulation. From the implementation perspective this was simply a *switch case* and for every letter corresponds an developed feature. This is highlighted in the next session (User Manual).

# 5 User Manual

## 5.1 Executing the application

The submitted folder *Interactive_Graphics_Project* includes:

- *Instruction_Document.txt* including the application instruction (also listed below)
- *Project_Solution* including the whole Visual Studio solution
- *Project_Executable* including the executable file

Therefore to **run** the application is enough to go *Interactive_Graphics_Project/Project_Executable* and double-click on **InteractiveGraphicsAssignment.exe**

**Instructions** (lower-case):
press **y** -> to start and stop simulation
press **w** -> to move close the camera to target
press **s** -> to move far the camera to target
press **c** -> to switch collision on or off
press **a** -> to add planets to the simulation
press **x** -> to write solar system state to text file
press **d** -> to decrease dumping coefficient
press **v** -> to disable or enable fog
press **f** -> to increase density of fog
press **r** -> reset the camera to original position
press **i** -> to move the car forward
press **j** -> to move the car on the right
press **l** -> to move the car on the left
press **o** -> to rotate the car clockwise
press **u** -> to rotate the car counter-clockwise
press **n** -> to lock the camera on the car

(upper-case):
press **w** -> to move close the camera to target
press **s** -> to move far the camera to target
press **f** -> to activate fly mode

(arrows):
press **arrow-up** -> to move close the camera to target
press **arrow-down** -> to move far the camera to target

(mouse):
press **left-button** and move the mouse -> to move in the space relative to the target
press **right-button** and point a planet -> to lock the camera focus to the planet selected

# 6 Conclusion

This project has been a great opportunity to boost advanced programming knowledge in C/C++ using a complete graphical environment provided by OpenGL. In addition, it has been really interesting developing the actual simulation using real motion laws of physics.

Being the simulation autonomous it has been decided to add an more concrete user interaction by allowing the user to control a car that can float around the solar system while the planets orbits around the star.

OpenGL it has been used in place of WebGL because it is much more efficient and offer a more complete and stable graphical environment. From the programming side C/C++ is much more well defined and readable compared to WebGL.

Future work may involve the creation of a well rendered simulation, using more complex objects in place of primitives spheres. However, this would require a more powerful GPU.