A Simulation of Solar System in OpenGl

Submitted by:

Hiranava Das

Shiwam Mittal

Aditya Singh Raghav

In partial fulfillment for the graduate level course titled

Of

CAP5705 Computer Graphics

FALL 2015

At

Department of Computer and Information Science and Engineering

12-12-2015

ACKNOWLEDGEMENT

We are deeply thankful to our professor, Dr. Corey Toler-Franklin, for her constant motivation and guidance towards the completion of this project without which this endeavor would not have been successful. We are also very thankful to Kai Zhang, the teaching assistant for the course. His constant encouragement and appraisal helped in the early completion of the project. Finally, I would also like to thank all my fellow students for their constant cooperation and motivation.

ABSTRACT

In this project we have implemented a simulation our solar system which shows the
8 planets, Earth's satellite Moon and the asteroid belt revolving around the Sun. The moon also revolves around the Earth as the Earth itself revolves around the Sun. Each of the planets also be rotate about their own axes and the elliptical orbits for each planet are traced as they revolve around the sun.

The planets and their orbits are rendered to scale, however as the sun is much larger than all other bodies, its size has been decreased in the simulation to make it visually affective.

The Sun in our simulation also act as a point light source, therefore cast shadows on the planets depending upon their relative position.

The users can interact with the simulation by moving the camera around to look at the solar system from various positions, so as so as to observe the various lighting effects. The users can also increase or decrease the speed of the simulation, add remove special effects like skybox, asteroids etc.

Table of Contents

Motivation

We chose this project as it allowed us to learn and implement various concepts of OpenGL programming. The simulation of solar system in particular, also helped us better our understanding of the concepts of computer graphics. Through this project we were able to integrate and implement a wide array of concepts that we covered in the class and the programming assignments. Some of them are as follows:

- Mesh manipulation
- Shading
- Lighting
- Texture Mapping
- Animation
- Camera manipulation
- User interaction
- Skybox
- Texture mapping
- Particle system

Our project can also be used for educational purposes by simulating how various events like the solar and lunar eclipse happen, how the planets are oriented with respect to each other, their rotation and revolution relative to each other etc.

Background

We have used actual data for the distance of the celestial objects from the sun, their rotation and revolution period from the sun etc. to make our simulation realistic.

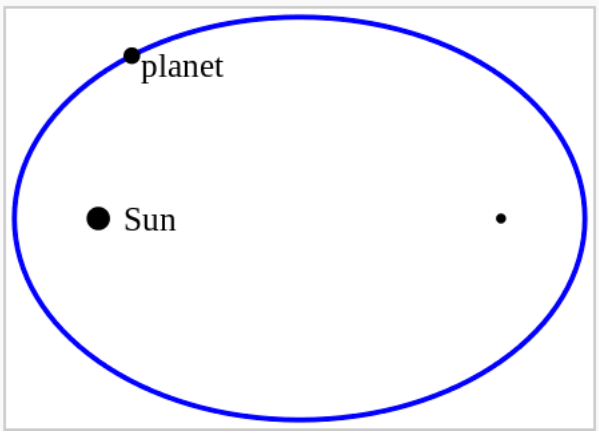
	Distance from Sun (miles)	Revolution (Days)	Rotation (Days)	Radius (miles)
Sun	-	-		432474
Mercury	39,580,000	88	58.6	1516
Venus	67,240,000	225	243	3760
Earth	92,960,000	365	1	3959
Mars	140,000,000	687	1.01	2106
Jupiter	483,800,000	4380	0.42	43441
Saturn	890,700,000	10585	0.44	36184
Uranus	1,783,939,400	30660	0.71	15759
Neptune	2,795,084,800	60225	0.66	15299
Pluto	3,670,050,000	90520	0.66	736.9
Moon	238,900	27.32	27.32	1079

Table 1: Statistics of Celestial Bodies

We have also tried to follow the physical laws of the solar system some of which are as follow:

- Kepler's 3 laws of planetary motion
  1. The path of the planets about the sun is elliptical in shape, with the center of the sun being located at one focus
  2. A line segment joining a planet and the Sun sweeps out equal areas during equal intervals of time
  3. Motion of the innermost planets is much faster than that of the outermost

- Revolution of Planets in elliptical orbit



$$r = \frac{p}{1 + \varepsilon \cos\theta},$$

Figure 1: Elliptical Orbit

R = Distance of the planet from the Sun

P = Semi-latus rectum

ε is the eccentricity = 0.7453

Θ is the angle to the planet's current position from the Sun

At angle θ = 0°, perihelion, the distance is minimum (Our starting point)

At θ = 180°, aphelion the distance is maximum

We have used the programming assignments as framework to model camera, implement shading, lighting and texturing, add user interactions etc.

Methodology

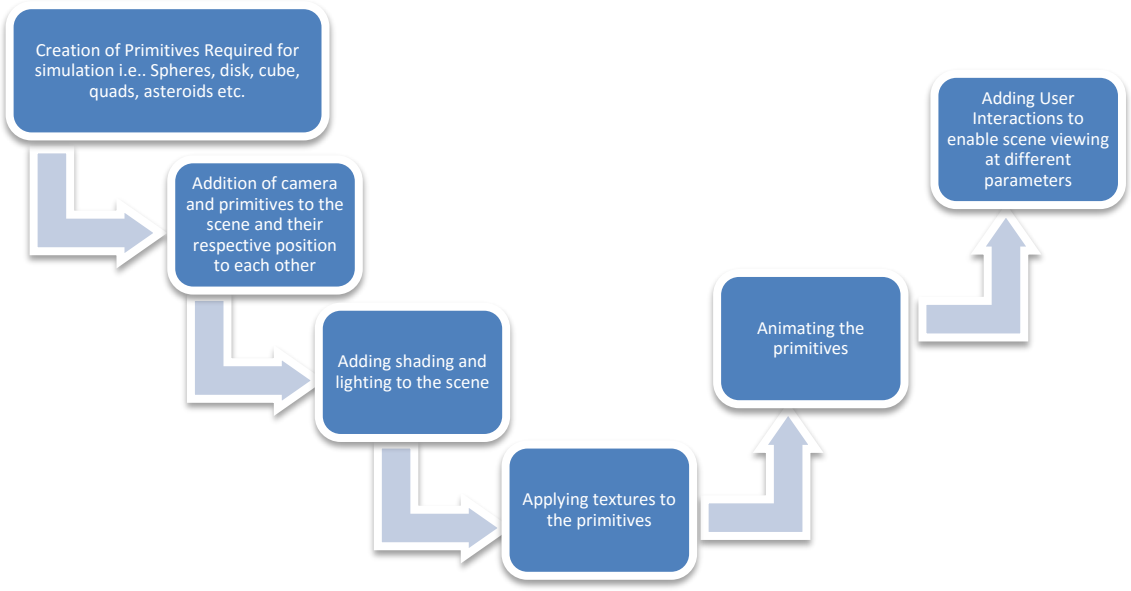


Figure 2: Flow of the Project

Implementation

Spheres:

We have modeled the sphere in our simulation using OpenGl function gluSphere() which has the following parameters:

- *Quad* :Specifies the quadrics object
- *Radius:* Specifies the radius of the sphere.
- *Slices*: Specifies the number of subdivisions around the *z* axis (similar to lines of longitude).
- *Stacks*: Specifies the number of subdivisions along the *z* axis (similar to lines of latitude).

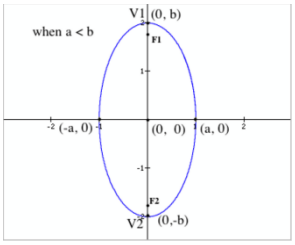
Orbits:

We render orbits using the following OpenGl functions:

- *glBegin()* :
    - Takes in the shape of the object we want to create

- glVertex3f(GLfloat x, GLfloat y, GLfloat z):
    - Specifies the coordinates of a vertex.
- GL_LINE_STRIP *:*
    - Draws a connected group of line segments from the first vertex to the last.
    - Vertices n and n + 1 define line n.
    - N - 1 lines are drawn.

We have used parametric equation to represent the ellipse to draw the orbits:



$$\cos t = x/b \qquad \sin t = y/a$$

$$\cos^2 t + \sin^2 t = 1$$

$$(x/b)^2 + (y/a)^2 = 1$$

Figure 3: Parametric Equations of Ellipse

Lighting and Shading:

In our implementation we have set the sun as a point light source and the light emitted from the sun acts lights and shadows on the planets depending upon their relative position.

With OpenGL, you need to explicitly enable (or disable) lighting. If lighting isn't enabled, the current color is simply mapped onto the current vertex, and no calculations concerning normals, light sources, the lighting model, and material properties are performed. Here's how to enable lighting:

glEnable(GL_LIGHTING);

We have explicitly enabled the sun as a light source:

glEnable(GL_LIGHT0);

Light 0: white light (1.0, 1.0, 1.0, 1.0) in RGBA diffuse and specular components

We have used the following OpenGl function for our implementation to add an ambient light source to our screen so that the object on which no light falls don't look completely dark in the scene.

void **glLightfv** (GLenum *light*, GLenum *pname*, const GLfloat * *params*);

Light: Specifies a light. The number of lights depends on the implementation, but at least eight lights are supported.

Pname: Specifies a light source parameter for light.

Params: Specifies a pointer to the value or values that parameter pname of light source light will be set to.

Texture Mapping:

First we have created a class texture which loads the texture file in BMP format to the program. To do this we have used the following OpenGl functions:

- glGenTextures(GLsizei *n*, GLuint * *textures*) :

  N Specifies the number of texture names to be generated.

  Textures Specifies an array in which the generated texture names are stored.

- glBindTexture(GLenum *target*, GLuint *texture*);

  Target Specifies the target of the active texture unit to which the texture is bound. Must be either GL_TEXTURE_2D or GL_TEXTURE_CUBE_MAP.

  Texture Specifies the name of a texture.

- glTexParameteri(GLenum *target*, GLenum *pname*, GLint *param*);

  Target Specifies the target texture of the active texture unit, which must be either GL_TEXTURE_2D or GL_TEXTURE_CUBE_MAP.

  Pname Specifies the symbolic name of a single-valued texture parameter.

  Param Specifies the value of *pname*.

- gluBuild2DMipmaps(GL_TEXTURE_2D, 3, texture1->sizeX, texture1->sizeY, GL_RGB, GL_UNSIGNED_BYTE, texture1->data);

  It builds a series of prefiltered two-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture-mapped primitives.

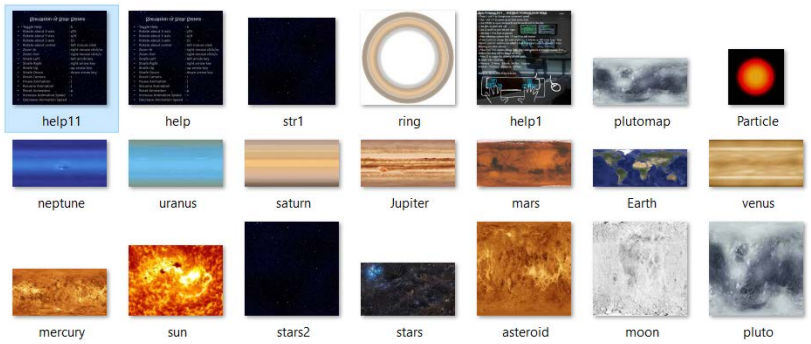


Figure 4: Textures

Since are texture class allows to load only textures in bmp format we create all the textures in bmp format and initialize their paths in the program.

Since we are using gluSphere we have applied textures to the sphere using a quadrics object. They are routines to model and render tessellated, polygonal approximations of spheres, cylinders, disks and parts of disks

$$a_1x^2 + a_2y^2 + a_3z^2 + a_4xy + a_5yz + a_6zx + a_7x + a_8y + a_9z + a_{10} = 0$$

We have specified the following parameters for our quadrics object:

- GLU_SMOOTH One normal is generated for every vertex of a quadric.
- gluQuadricTexture specifies if texture coordinates should be generated for quadrics rendered with *quad*. If the value of *texture* is GLU_TRUE, then texture coordinates are generated, and if *texture* is GLU_FALSE, they are not. The initial value is GLU_FALSE.

The manner in which texture coordinates are generated depends upon the specific quadric rendered.

Animation:

To rotate and revolve the planets around sun in elliptical orbits as explained in background we use the following OpenGL functions:

- glRotatef(GLfloat *angle*, GLfloat *x*, Glfloat *y*, Glfloat *z*)

  Parameters:

  - *angle*: Specifies the angle of rotation, in degrees.
  - GLfloat *x:* Specify the *x*, *y*, and *z* coordinates of a vector, respectively.
- glTranslatef(GLfloat *x*, Glfloat *y*, Glfloat z*)*

  Parameters:

  - GLfloat *x:* Specify the *x*, *y*, and *z* coordinates of a vector, respectively.

Moon:

The moon class is very similar to the planet class but as the moon revolves around the earth as well as around the sun with the earth, we need to translate it twice once with respect to the earth and then translate it center with respect to the sun

Skybox:

To implement the skybox we use glBegin(GL_QUADS) to draw the six faces of the cube. GL_QUADS requires the four vertices of a quadrilateral as input to draw a quadrilateral. We then use glVertex3f(x, y, z) to specify the each point of our cube and then use glTexture3f(x, y, z) to map the corresponding image to the vertex.

We disable the depth writing so that Skybox is behind all the objects of our scene and close glBegin with glEnd().

Saturn Rings:

We use gluDisk() function which creates a disc on the z=0 plane. It takes input the outer and inner radius of the disk along with the number of slices and rings around the z axis.

Then we use a quadrics object to map the texture onto the ring.

Asteroids using Particle System:

Each particle is created with OpenGl function gluSphere and has its own position, speed, lifetime and size. When generated they are given a random angular speed of the same order as the planets and a random size of the order if 1000 – 2000 miles. A particles dies when its lifetime becomes zero or size reduces to 10 miles.

These particles are initially created in a straight line between Mars and Jupiter and then slowly spreads to form a belt between them. The particles speed and size changes with each collision and expires after its lifetime. New particles are created when old ones die out.

Collision Detection:

When two objects are within a fixed radius a collision happens. In our implementation we have taken a difference of 2000000 miles as a limit for collision. The final velocity of object is 10% of the original velocity and in the opposite direction.

Each collision reduces the size of the particle with smaller momentum to half of its original size. The collision between particles obey the rules of an elastic collision that is the velocity of the particle is reversed and is reduced by a factor k = 0.5. As the size reduces to below 10, the particles dies out.

Camera and User Interactions:

We have used the camera class provided in the programming assignments by Dr. Corey Toler-Franklin to model the camera.

We allow the user to rotate the screen in two modes for a better view of the scene from different angles.

1. Rotation about the primary axis i.e. X,Y,Z.
2. Rotation in orbit mode where the look at remains the same while the camera can move anywhere in the scene

Similarly we have added other camera manipulations.

1. Strafe up/down/right/left : We have added different keys to allow this at different parameters
2. Zooming in/Zooming out: We allow the user to zoom both using keyboard suitable for precise changes and using mouse for large changes.

We have allowed the user to increase or decrease the speed of animation. We do this by starting at a time factor of one and at each keypress event we add/subtract 0.1 from the factor.

The user can also pause resume and reset the animation.

The user can also add or remove the features like skybox and asteroids etc. using keypress events

Rendering the Menu:

To render the menu on the scene we have used GL_QUADS to draw a rectangle on the top right hand corner of the scene. Then we create an image for the menu and use texture mapping to map that image on the quad.

Figure 5: Menu

Results

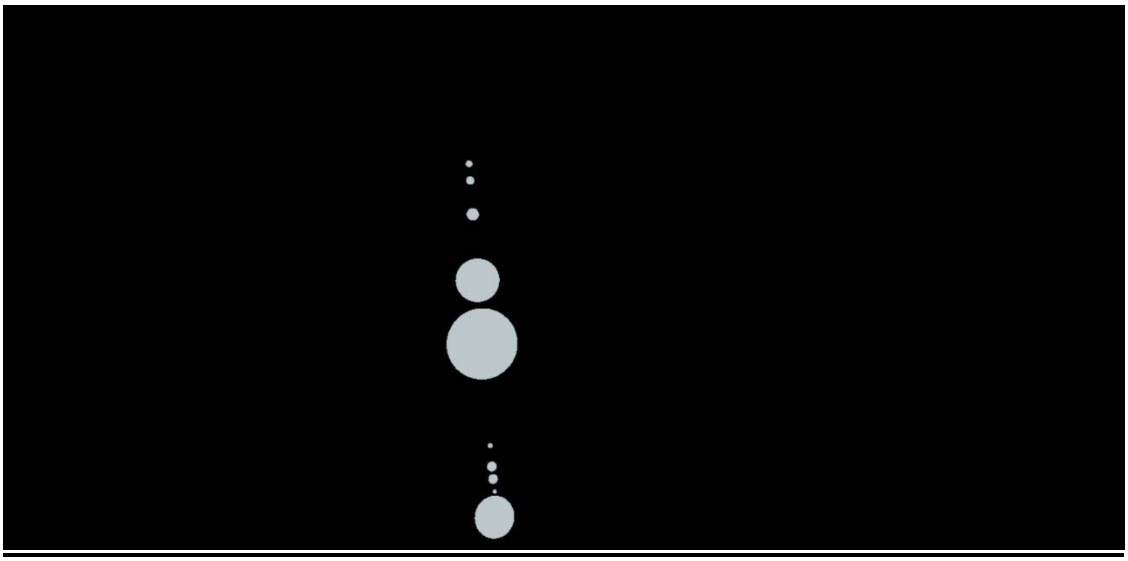


Figure 6: Spheres for Sun and the Planets

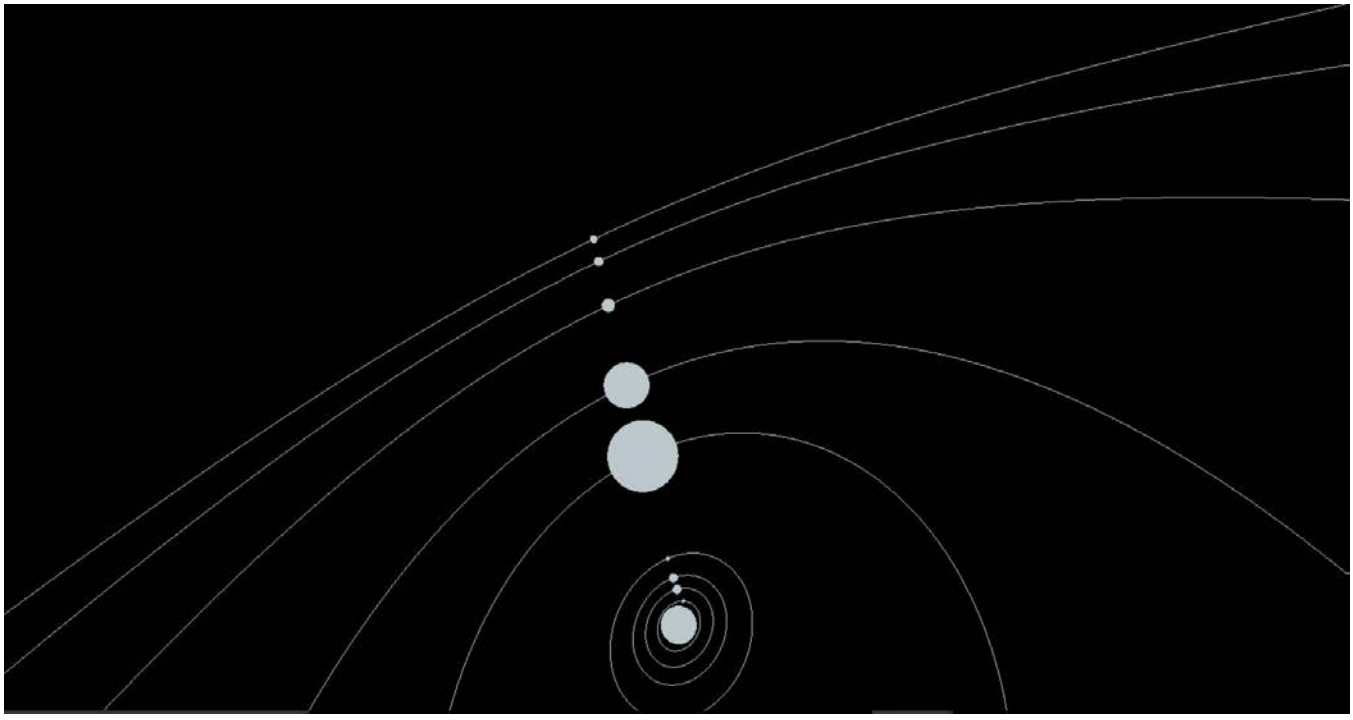


Figure 7: Planets with Orbits

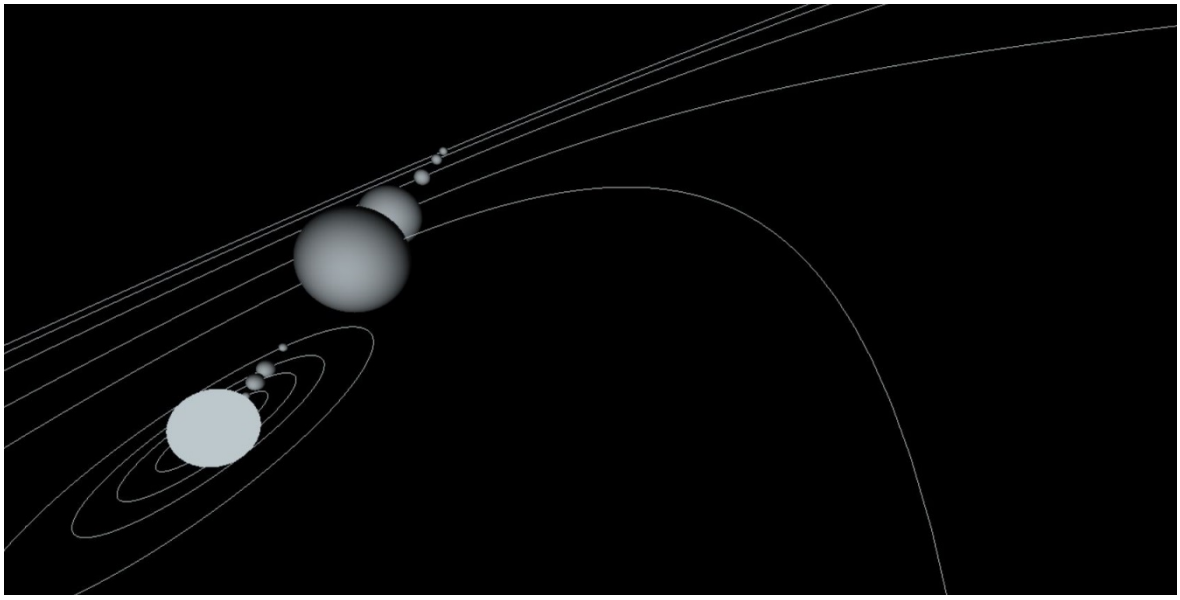


Figure 8: Planets with Lighting and Shading Enabled

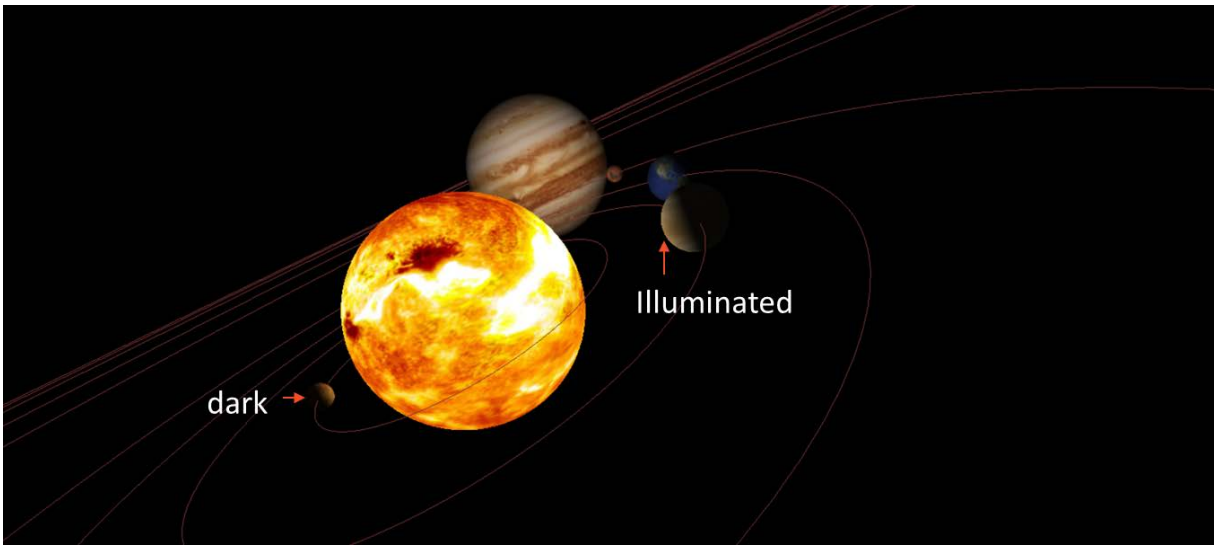


Figure 9: Planets with Lighting, Shading and Textures

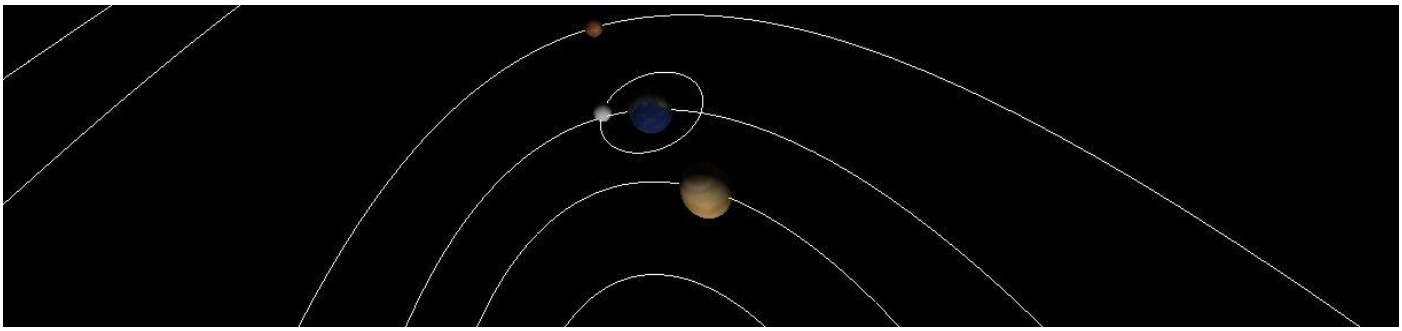


Figure 10: Moon Revolving around the Earth

Figure 11: Saturn and its Rings

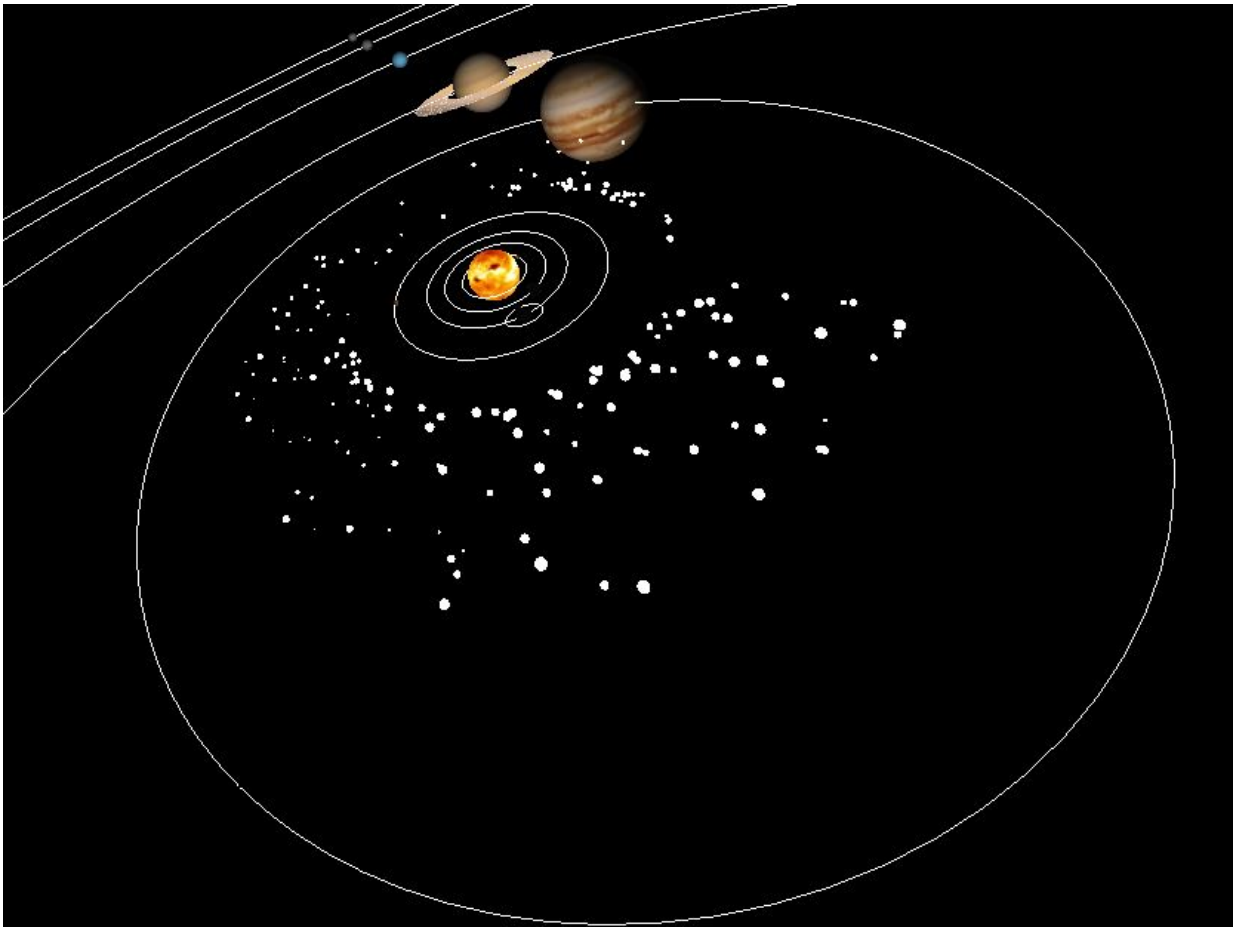


Figure 12: Asteroid Belt

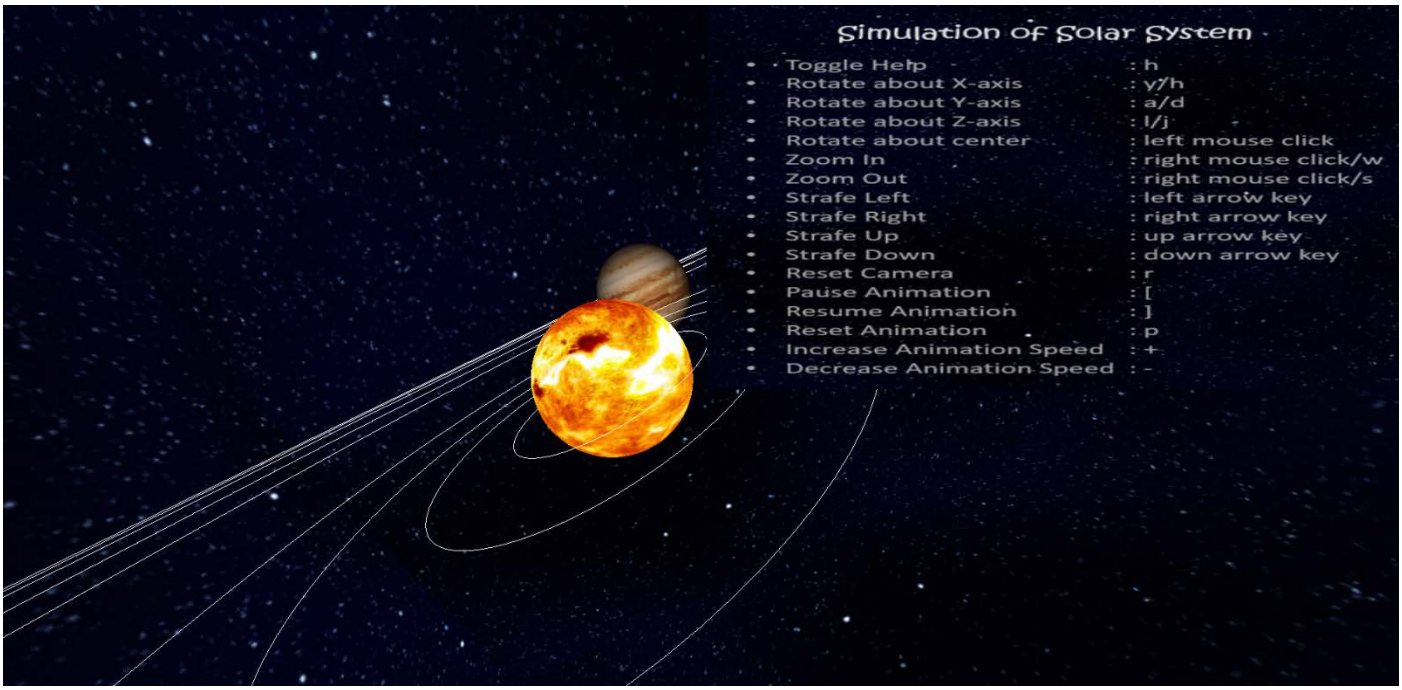


Figure 13: Final Result

Challenges

- Adding texture on Saturn's Ring: When we use the quadrics object to map the texture on the rings of Saturn only a texture oriented in a specific way works.
- Formation of Elliptical orbits keeping revolution time and rotation period of all the planets in agreement with Kepler's second law
- To model sphere's we were initially using glutSolidSphere function, however it does not allow texturing using a quadrics object therefore making the texturing process more computationally expensive, and hence using Glu Primitives seems like a better idea for modelling primitives.
- In particle system it is difficult to understand and implement collision detection between two particles and their following behavior.

Future Scope

- We can enhance the user interaction by allowing the user to click on various planetary bodies to display their information
- We could also allow the user to orient the planet and camera in such a way that phenomena's like solar eclipse, lunar eclipse etc. can be observed.
- Optimization of animation.
- Adding bump mapping, displacement mapping etc. to make the textures look more realistic.

References:

- https://www.opengl.org/
- http://www.glprogramming.com/red/
- https://www.opengl.org/sdk/docs/man2/xhtml/
- http://natureofcode.com/book/chapter-4-particle-systems/
- http://www.openglprojects.in/2015/02/journey-of-space-shuttle-or-rocket.html#gsc.tab=0
- https://github.com/harry1357931/SpaceShuttle-3DAnimation-OpenGL
- http://www.math.ucsd.edu/~sbuss/MathCG/OpenGLsoft/Solar/Solar.html
- https://github.com/dmitriibarillo/OpenGL-Solar-System
- http://csclab.murraystate.edu/bob.pilgrim/515/orbit_demo.html
- http://www.codemiles.com/c-opengl-examples/solar-system-transformations-t7292.html
- https://github.com/digwiz/solarsystem