Project TSBK07
Solar System
13/05/2019

Roberto Corrochano Piqueras
robco726@student.liu.se

Sergio García Prados
serga155@student.liu.se

Linköping University
Sweden

Index

1. Introduction. Describe the problem, basically the specification you started from. What features were mandatory and optional?

This project is about an interactive solar system, including the planets that belong to our solar system spinning around the sun. Each planet has one satellite spinning around. There are 2 different views for this project to move by, a general view where the user can move as he wants through the solar system and another one where the user controls one rocket as a way of transport in 3rd person to move between the different planets and satellites. The rocket may interact with the spheres and they might have lighting methods applied.

Mandatory features:

1. The center of the solar system shall be the Sun (a spherical object acting as a light source in every direction)

2. The solar system should be inside a skybox with "space texture" (appearance of space with distant stars

3. At least 2 planets must be implemented (spherical objects with some kind of terrain and texture that reflect light and create shadows)

4. The planets spin around the sun in their own orbits

5. Each planet must have at least one satellite (light sources). The satellites should spin around the planets.

6. A spaceship should be implemented (try to use Blender to do so a model)

7. The spacecraft should be able to steer forward, backward, up, down and rotate around one shoulder.

8. The spaceship should be able to land on a planet. Collisions should be handled so that the spaceship does not go through planets, suns, satellites or anything.

9. The camera should follow the spaceship.

Firstly we started implementing the skybox.
Secondly we worked with the planets, satellites and the general view. We positioned the sun and all the planets and their respective satellites in their own orbits. Then, we added textures to the spheres and also we configured the view to be controlled via keyboard.

Finally, we implemented the rocket, the view through it and the interaction with the spheres.

The “might do” features could be many possible options like:

- Showing written information about the planet when the spaceship lands on it
- Performing graphics
- Sound effects
- Including space rings to go through and to accumulate points
- Including enemy spaceships to attack and getting points
- Including the rest of the known satellites, planetoids, comets, asteroids of the Solar System with a bigger size version of the system than the current project
- Taking off animation of the spaceship from the planets

2. Background information. Any information about the kind of problem you solved that is needed to follow the rest of the report.

We had to redefine the function SetMat4 in VectorUtils3 since it did not recognize us the function

3. Implementation. Tools used, program structure.

We built our project above the operative system Linux. The code was wrote in C so we used the GNU compiler for C programs, gcc.

In terms of the libraries used it, we benefited from the source codes which was given to us for the labs.

To compile and run the program we have used a makefile so you just have to type “make all” in the commad line.

The C program itself is divides in several methods:

1. **SetMatrix4:** creates a 4 by 4 matrix with the parameters passed
2. **loadTextures**: load all the textures for the cubemap (6 images of the space), planets, satellite and rocket

3. **RadiusObj**: calculates the radius of an object depending of a scalar value which is applied to the object

4. **CollisionDetection**: creates an interaction between the rocket and the spheres if they collide between them. It needs the radius of the sphere to calculate the minimum distance to the rocket so that if is less than it, they will collide

5. **keyboard**: interaction between the keyboard and the world

6. **timer**: orders a new call to the display function plus starts the next round with the timer

7. **init**: function called once which initialize all the global variables

8. **display**: executed for every new image rendered

9. **main**: main function which is executed the first.

Due to the complexity of some methods, the followings sections will explain in much more detail the implementation of them:

1. **loadTextures**: the aim is to match the six images (cubemapTextureFileName), top, bottom, right, left, front and back, on each side of a cubic skybox. Each one is identified by the target GL_TEXTURE_CUBE_MAP_POSITIVE_X plus an index. For the planets and satellites, the textures are loaded in the variable satellites_textures. Same for the rocket with rocket_texture

2. **RadiusObj**: to calculate the radius, we take one vertex of the model, we scale it depending on the parameter and we normalize it to know the length

3. **CollsionDetection**: to make an interaction between the rocket and the spheres, we just calculate the length between the rocket and the sphere and in case that length is less than the sum of both radius, there is a collision. In that case, we spli the forwardVector taking the normal vector as (0, 1, 0) and we calculate the new forwardVector as the subtract of the nortmal and tangent. This make seems the rocket to be landing on the surface of the planet or satellite.

4. **keyboard**: this method control the movement of the camera and the rocket and change the camera view. Lets start talking about the general view and then with the rocket view.

   General view: modify the variable which save the position of the camera (cameraVectorGeneral), the postion where the camera should view (viewVectorGeneral) and the rotation degree variable (angle).

   1. <u>Go straight</u>: add to the x and z component the sine and cosine respectively of the angle rotated

   2. <u>Go backwards</u>: subtract to the x and z component the sine and cosine respectively of the angle rotated

   3. <u>Turn left</u>: add 0.03 to the rotation angle variable

4. Turn right: subtract 0.03 to the rotation angle variable
   The component x and z of the view vector position is initialized respectively yo the x plus the sine of the angle and the z plus the cosine of the angle of the position camera vector so that it always look at the front

Rocket view: modify the 3 vectors: forward, right and up (always normalize after modifying them), the rocket and camera position and the direction vector from the rocket to the camera (cameraDirectionVector). The latter is also always normalized.

1. Go straight: add the forward vector to the rocket position vector

2. Go backwards: subtract the forward vector to the rocket position vector

3. Turn left: to rotate left the forward vector, we first scale the right vector and then we sum both of them. The right vector is calculated as the cross product between the forward and up vector

4. Turn right: the same as turning left but scaling the right vector in reverse

5. Rotate up: to rotate up the forward vector, we first scale the up vector and then we sum both of them. The up vector is calculated as the cross product between the right and forward vector

6. Rotate down: the same as turning up but scaling the up vector in reverse
   The direction vector from the rocket to the camera is calculated as the sum of the forward vector in reverse and the up vector. Continually, the vector is scale by 3 to have a proper view of the rocket.
   After that, the position of the camera will be position rocket plus the direction vector from the rocket to the camera.
   Finally, we calculate the rotation matrix calling the SetMatrix4 with the 3 main vectors: right, up and forward. It is transposed.

5. **init**: we load the models, initialized and send the projection matrix to the vertex shader and initialize the global variables. At the beginning, the rocket is on the position (50, 0, 0) looking to the positive x axis and without any rotation so for that reason we initialized to the forward vector to (1, 0, 0), the right vector to (0, 0, 1) and the up vector to (0, 1, 0)

6. **display**: we just initialize and send the world to view martix depending on the variable “cam”, we draw the skybox and the objects with their respective textures.

The shaders is structured as follows:

1. **Vertex shader**: First we need to transform the normal vector in order to make the normal vectors follow the rotation of the model. We did that by removing the translation, which is equivalent to casting the 4x4 matrix to a 3x3 one. We also pass through the vertex position (inPosition) and the vector (exSurface), cast to vec3, formed by the product of the 3 main matrices (world-to-view, model-to-world and projection) and the inPosition vector cast to a vec4.

2. **Fragment shader**: We have implemented an extra method, arctan, to do the spherical mapping. Firstly, we initialized the s and t values. Then, depending on the variable drawSun, we apply lighting or not since the sun works as a light source. In case we draw a sphere different from the sun, we apply diffuse shading and add them to the texture.

4. Problems. Did you run into any particular problems during the work?

We had some problems in:

1. **Spherical mapping** (Figure 1). The problem was that we were applying mistakenly cylindrical mapping instead of sherical mapping.

2. **Shading** (Figure 2). The problem was that we were taking also negative specular values in the fragment shader.

3. **Collision detection** (Figure 3). The problem was that we were calculating mistakenly the position of the spheres. Hence, the length between the rocket and the sphere was wrong.

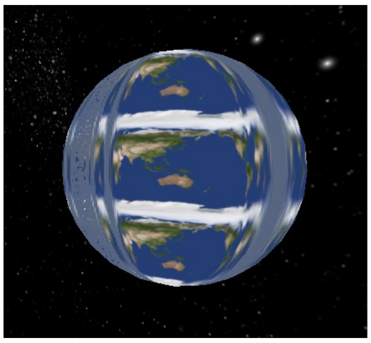
Figure 1: Bad spherical mapping

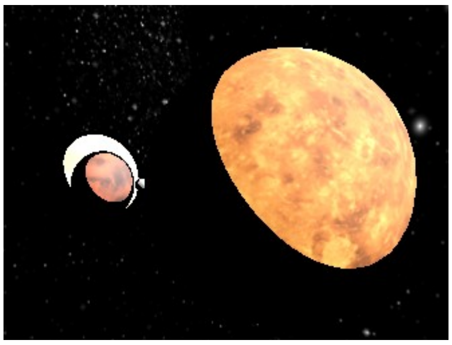
Figure 2: Bad diffuse shading

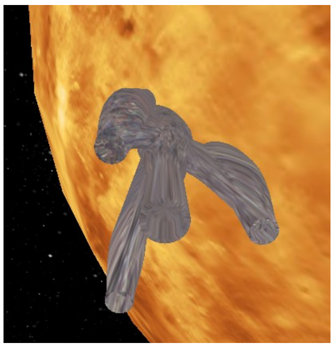
Figure 3: Bad collision detection

5. Conclusions. How did it come out? How could it have been done better?

As exchange students and coming from different fields of Engineering, we have some difficulties to organize ourselves and we decided on implementing this project because it was simpler than others (and even this one was a challenge for us), but also because it seemed funny and nice, although it sounds absurd. We had never created any animation and starting with planets in the middle of the space felt really good.

We know this kind of project could become much bigger and have several extra functions to implement as a video game, because everyone likes to drive a spaceship, so it can bring many ideas to improve it, as it has been explained in the introduction.