Celestial Mechanics Simulator

Computer Graphics TSBK07

Alexander Sjöholm Martin Svensson

2013-05-21

Table of contents

Introduction

The goal of the project was to implement an interactive and customizable solar system with shading, gravity and collision detection.

Data Structures

The figure below show the most important data structures along with their methods. They are described in further detail on the following pages.

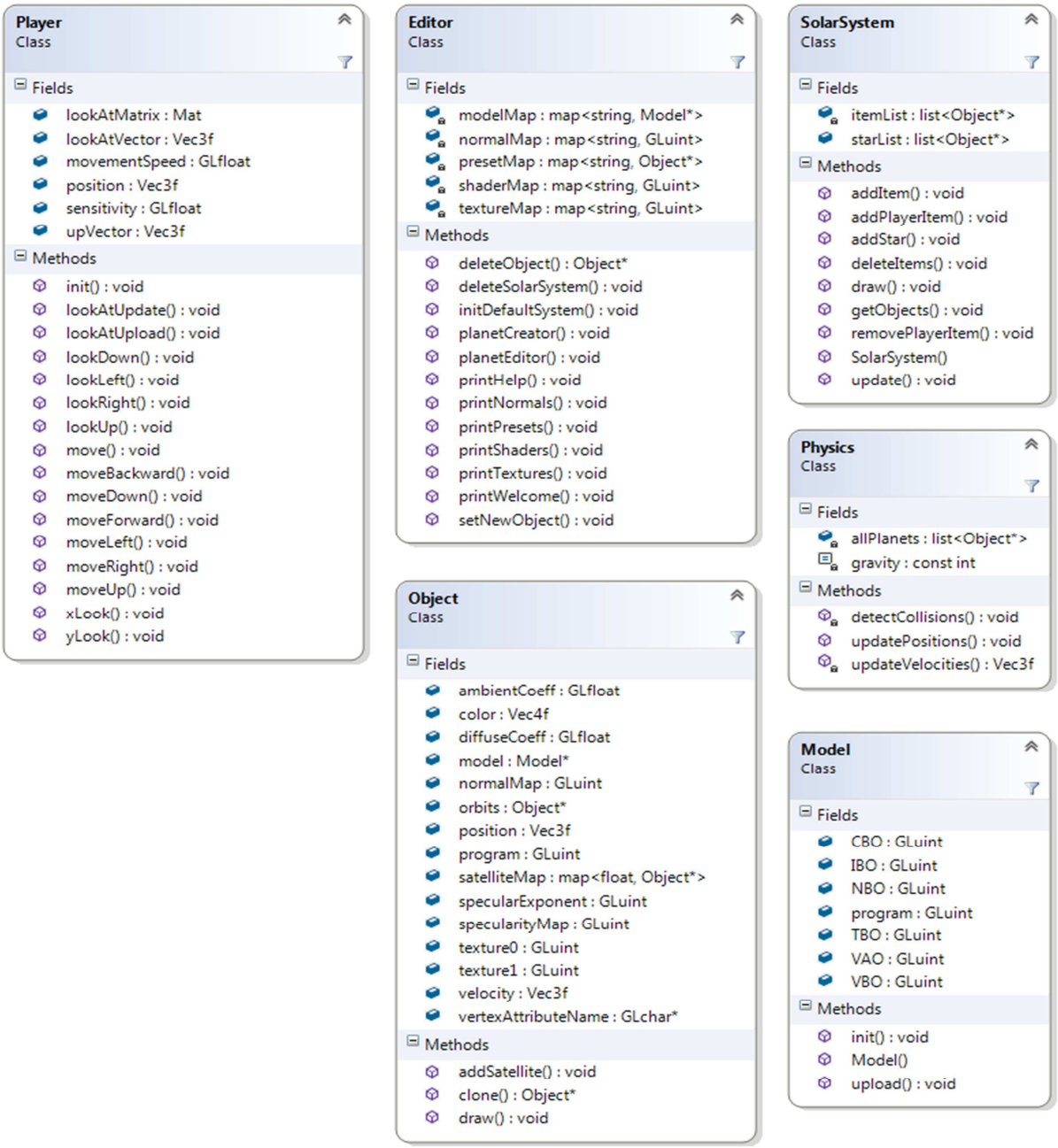

Figure 1 The above figure shows all implemented classes along with their most important methods and variables.

Solar System

The Solar System class is an overhead containing the functions needed to update positions, delete, select or draw all objects in the simulator, as well as lists containing all planets and items currently active.

Player

The Player class is essentially a class representing the camera, containing parameters such as movement speed and the lookAt matrix, and methods such as move left, move right etc.

Physics Engine

The physics engine contains all the functions required for the movement of both planets and items, and given the time passed since the last update, it goes through all the planets and objects currently active and updates their positions and velocities. The collision detection also takes place in this module.

Gravity

For convenience reasons only the items thrown away by the user are affected by gravitational pull, while all the planets and stars have their satellites locked in orbit. First all planets have their positions updated according to their orbital velocities and secondly, using the positions from the previous update, all items have their velocities updated according to:

$$F = G\frac{m_{object}m_{planet}}{d^2} \Rightarrow \left[F_{object} = a_{object}m_{object}, m_{planet} \gg m_{object}\right] \Rightarrow a_{object} \approx G\frac{m_{planet}}{d^2}$$

Where $m_1 \ll m_2$ and m_2 is the mass of the planet, and d is the distance from the planet to the object. This acceleration is then multiplied with dt and a normalized vector from the object to the planet, and finally added to the object velocity. The final new velocity is the given as the sum of the current velocity and the acceleration contributions from all planets.

Collision Detection

During the planet position update, pointers to all planets are stored in a list which is used for the collision detection. The collision detection is purely spherical and all items are checked against both each other and the objects. Collision detection between planets is not implemented since these are locked in orbit at all times.

Editor

All external manipulation of the solar system is performed from the Editor class. The editor consists of a command window from which it is possible to create new planets delete planets, edit a selected planet and so forth. The editor had access to all available shader programs, textures and bump maps, making all planets fully customizable. The planet selection function is implemented using a Plücker-coordinate based ray caster just like the one in the fragment shader.

Models

In the entire project there is mainly one model that is being used, namely the sphere. All planets and also the skybox, that actually is a sphere, originate from the same model. This model is later turned into different objects with different properties such as texture and structure.

There is a model loader implemented that loads a file of sort .obj. There is also a sphere generator that generates a sphere of given vertex resolution with normals and texture coordinates.

Objects

Objects are used to save resources by using the same model for several similarly looking objects. Each object has a list of properties illustrated in: UML.

Texture

To texture a sphere is a known problem. In the stitch there will be one vertex that needs two different texture coordinates depending on from what side the texture is being interpolated. Otherwise one will get one big interpolation over the entire texture in the stich.

In this project it is solved in the sphere generator. It actually inserts two vertices at the same location, one with the texture coordinate zero and one with the texture coordinate one. These are then combined correctly to their neighbours but never with each other.

Bump map

To add a feeling of structure to for instance the earth a bump/normal map is used. If one where to add enough vertices to add real height differences on the sphere the models would become too large. The normal map is an RGB image where the three different color channels represents x, y and z. To add these to a plane is not a big deal. Then you simply change the given normals to the ones in your normal image; not very different from reading a texture. The real difficulties begin when you want to do this to a sphere. The normal map contains normals expressed in local model coordinates, which for a plane are simply x, y and z, but for a sphere this becomes the local normal, a tangent and a bitangent. These need to be aligned for all positions on the sphere for the mapping to look good. The normal is given in the model. To find the first tangent the cross product between the current position and the top of the sphere is used. To avoid problems at the top of the sphere the point to cross multiply with is located just slightly above the top. This way the cross product is always valid. To find the bitangent the cross product between the the normal and the first tangent is calculated.

Specularity map

To add the effect of water being more specular than land on the earth a specularity map is used. This is used to adjust the specularity coefficient in the Phong shader.

Shadows

By implementing a Plücker based ray caster in the fragment shader shadows where easily achieved. The computational complexity grows fast though. For every pixel you need to cast a ray from every light source to the pixel and then check the distance for every planet to this line. If the distance is smaller than the planet radius it generates a shadow in the current pixel. This does not give any soft shadows.

There exists different methods to get rid of the sharp shadow edges. The one mentioned in this course is to randomly alter the position of the light source and sum the contribution from every random position for every pixel. This method quickly becomes very computational heavy.

Another method that we invented on our own is to use an activation function of the light source position and radius and the blocking objects position and radius. The activation function could be for instance a sigmoid function.

Shaders

To avoid unnecessary branches in the shader several different shaders are used designed specifically for their purpose. The earth is the object with the most texture data available and therefore it got a dedicated shader to handle texture blending, bump mapping, specularity mapping and shadows. The more common planets do not have a night texture, bump map or specularity map and hence need none of these features in their shader.