CO1301: Games Concepts
Lab 11 - Solar System Prototype

1. Introduction

1. In this lab, you will first set up the earth and the moon, and then extend your solar system prototype to include other planets and the sun.
2. You are not expected to implement anything remotely realistic. If you attempted to create the solar system to scale them it would be impossible to see and boring to use.
3. The primary purpose of this lab is to introduce the idea of a model hierarchy, the dummy model and to practice implementing collision detection.

2. Setting Up

1. Create a new TL-Engine project called "Lab11_SolarSystem_Project".
2. Use the "Stars.x" mesh to create a model at the origin that will serve as a skybox. It's radius is 100, and when placed in your scene it will create a background of stars. The model also has some alternative skins:
   - “StarsHi.jpg”
   - “StarClouds.jpg”
   - “StarCloudsHi.jpg”
3. The "Planet.x" mesh can be used create spheres with a radius of 1.0 and has several basic skins to create all the main bodies in the solar system:

   “Sun.jpg” “Mercury.jpg” “Venus.jpg” “Earth.jpg”

   “Moon.jpg” “Mars.jpg” “Jupiter.jpg” “Saturn.jpg”

   “Uranus.jpg” “Neptune.jpg” “Pluto.jpg”

   There are also some additional and hi-resolution skins:

   “EarthHi.jpg” “EarthPlain.jpg”

   “EarthNight.jpg” “EarthPlainHi.jpg”

   “EarthNightHi.jpg” “MoonHi.jpg”

4. You can use TL-Engine scaling functions (Scale()) to make models larger or smaller. This will allow you to use the same planet mesh to create models for the Sun, the Earth and all the other planets - you scale each one differently to create appropriate sizes.
5. There is also another engine function SetFarClip that changes how far away you can see. In a space game you can see very far so this is needed, use it when objects are “clipped” in the distance:

```
myCamera->SetFarClip( 100000 ); // Now can see up to 100k distant
```

6. Begin by creating the Earth (at the origin) and the moon (to the Earth's side) uisng the "Planet.x" mesh and the appropriate texture for each body. Place them close enough together so that you can see them both comfortably (remember, this is not meant to be realistic). You will have to scale the model down for the moon.

   You may want to position your models at 10.0 along the y-axis, so that they are almost centered within the camera's view.

7. In your game-loop add code that rotates the Earth about its local y-axis.
8. Then make the Moon revolve around the Earth by attaching it to Earth as a child.

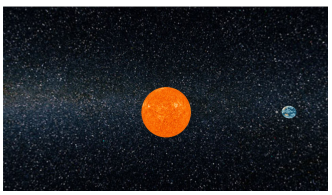
```
moon->AttachToParent( earth );
```

   Remember, this means that the moon's position is now relative to the position of earth. Therefore, you may need to adjust the coordinates used to position it.

9. This is all well and good, but what would happen if you wanted the Earth and the Moon to rotate at a different speeds? The way things are set up at the moment the Earth must rotate around its axis at the same speed that the Moon rotates around the Earth. It is possible to set up the scene so that the Earth and the Moon rotate at different speeds and this is by introducing a dummy model using the "Dummy.x" mesh. You load and use the dummy model in exactly the same way as any other mesh and model. The difference is that the dummy is never rendered: it is just an invisible point in space.
10. Create a dummy model at the same location as the Earth. Attach the Moon to the dummy instead of the Earth, and in the game loop add code that rotates the dummy about its local y-axis. Now you can rotate the Earth separately from the dummy (and hence the Moon).

3. The Solar System

1. Let us expand what we have so far to a mini solar system.
2. Create a model for the sun using the Planet.x mesh, scale to show the size of the Sun compared to the Earth (again it should just be bigger, not realistic), and use the appropriate skin.
3. Change the position of your Earth, Moon and dummy models with respect to the position of the Sun (remember the moon is a child of the dummy, therefore you will need to position it relative to the position of the dummy).



4. Add a few other planets around the sun and scale them to different sizes. You don’t need to worry about getting the sizes of the planets correct, just try to make it look interesting.
5. Make the planets orbit in a realistic way (they rotate around the sun at different speeds and spin on their own axes). You will need to think about using parenting and dummy objects.
6. You will need to think carefully about the scales and distances that you use. Try to make a your scene look good but feel suitably large without being impossible to navigate.
7. For future reference. You are formally building an animation hierarchy. Your program uses the same concepts and general form that you would to build the an animation hierarchy for a person.

4. Space Collisions

1. Load up and create a spaceship. The model can be found inside "spaceship.zip" on Blackboard. You may have to scale this up or down depending on the size of your planets.
2. Implement movement control for the spaceship. Remember to set this up local to the spaceship.
3. Attach a chase cam for the spaceship. You can refer back to the airplane exercise for how to do this.
4. Implement collision detection and an appropriate collision resolution for the spaceship and the Sun only. Use a sphere for the spaceship and check for sphere-to-sphere collisions. Remember, the steps are:

    1. Calculate the vector between the two sphere centres (model positions)
    2. Get the length of this vector (the distance between the models)
    3. If this distance is less than the sum of the sphere radii then there is a collision

   You can use Week 8's lecture slides or lab sheet as reference for implementing collision detection and resolution.

/

5. Using Functions for Collision Detection

1. Right now our program contains sphere to sphere collision detection between the sun and the spaceship. If we were to implement sphere to sphere collision between the spaceship and all the other bodies without using functions, our game-loop could get messy quickly. Let us, therefore, create a sphere to sphere collision detection function that we can call anytime we want to check for a collision between the spacehsip and any of the other bodies in our program.

2. The sphere to sphere collision dectection function should check if there is a collision between two objects bounded by spheres. Therefore it needs to know the position of the objects and their radii of the spheres. If there is a collision it should return True, and return False is there is no collision. Use the code below to declare the function, the declaration should be placed before the start of the program's main function

```
bool sphere2Sphere(float s1XPos,  float s1ZPos, float s1Radius, float s2XPos, float s2ZPos, float s2Radius);
```

3. Next, define the function below the closing brace of the main function.

```
bool sphere2Sphere(float s1XPos, float s1ZPos, float s1Radius, float s2XPos, float s2ZPos, float s2Radius) {
        float distX = s2XPos - s1XPos;
        float distZ = s2ZPos - s1ZPos;
        float distance = sqrt(distX*distX + distZ*distZ);
        bool collision;
        if (distance < (s1Radius + s2Radius)) {
            collision = true;
        }
        else {
            collision = false;
        }

        return collision;
}
```

   The function takes in the x and z positions of two sphere volumes (why do you think the Y position is ignored in this case?) and their radii, calculates the distance between the spheres, compare the distance with the sum of their radii, returns true to show there is a collision if the distance is less than the radii sum, or returns false to show there is no collision if the distance is greater than the radii sum.

4. You can now call this function inside your gameloop to check for collisions. Use the GetX() and GetZ() functions to get the position of an object, and make sure you take scaling into consideration when providng the radius of an object (e.g. a planet that is not scaled has radius 1, a planet scaled up to 2 has radius 2, while a planet scaled down to 0.5 has radius 0.5). An example of the use of the function is shown below:

```
if(sphere2Sphere(earth->GetX(), earth->GetZ(), 1.0f, spaceship->GetX(), spaceship->GetZ(), 4.0f)){
    myEngine->Stop();
}
```

   The spaceship in its default scale has a radius of about 4.0f, you can get a more exact value using its bouding box coordinates found within it's mesh file.

5. Using the sphere2sphere function, update your gameloop to check for collision between the spacehsip and all the bodies in your solar system.

6. Try and optimise the collision detection function based on what whas covered in the lecture this week.

7. You can reuse this function in other programs. You should also try defining functions for other collision detection approaches so you can easily reuse them.