Professor: Alireza Tavakkoli

CS 480 Term Project: Solar System

December 15, 2022

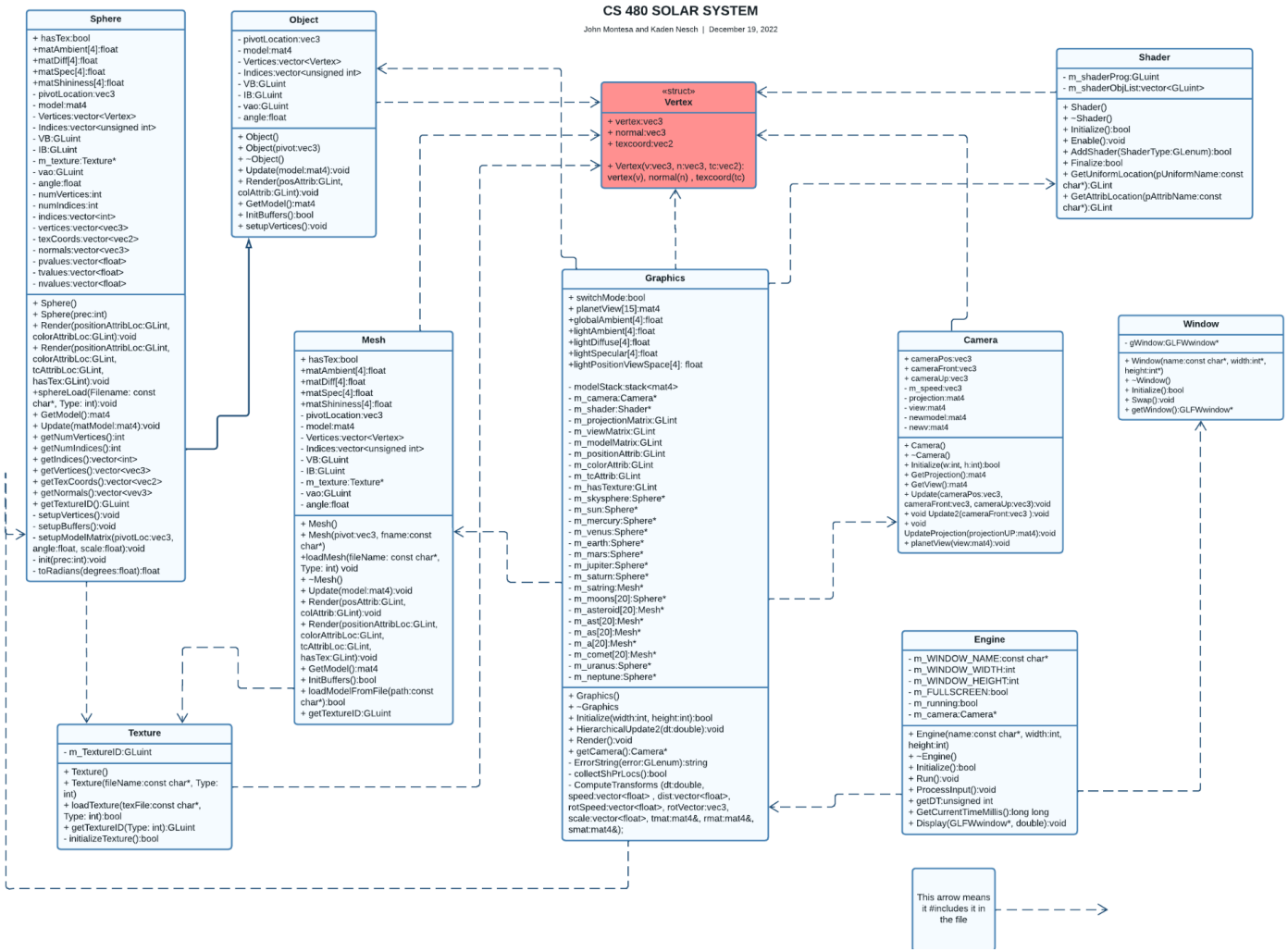
John Montesa

Kaden Nesch

Cover Page:

John Montesa	Kaden Nesch
Accurate animation of the solar system, including rotational axis, orbital speed around sun, distance from the sun, sizes, rotational speed, and elliptical orbit for each planet.	Generating and rendering each planet and providing them with their corresponding textures from the included asset folder.
Creating the skysphere to create a galaxy in the program.	Including the addition of diffuse textures and normal textures to each planet, moon, comet, and asteroid.
Keyboard Implementations	Keyboard Implementations
Camera movements	Camera Movements
Third person view of the spaceship (Offsetting the camera), as well as developing the spaceship movements when traveling such as speeding up, slowing down, and braking.	Including the addition of a lightsource(sun), which emits light to all other objects around the solar system. Also included the addition of ambient lighting from the galaxy that lights up all objects.
Generating and rendering the corresponding moons that come with each planet that has them, as well as animating them accurately	Generating and loading inner and outer asteroid belts in proper locations, between mars and jupiter, and past neptune. Also included the addition of Ceres inside of the first asteroid belt.
The Planetary Observation Mode: including updating the camera matrix to show each planet in orbit and keyboard implementations. Implementations include toggle switch to different mode, and different key bind actions.	Included the generations of specific objects that correlate to each specific element, such as the comet, asteroid, and the starship. Developed normal maps for those objects.
Documentation of code	Documentation of Code
General Shaders, Textures, Models, Meshes, Camera, Engine, and Graphic Coding	General Shaders, Textures, Models, Meshes, Camera, Engine, and Graphic Coding
TOTAL CONTRIBUTION: 50%	**TOTAL CONTRIBUTION: 50%**

UML (FULL SIZED PDF IS INCLUDED)



Implementation Section

Mesh Generation and Loading:

- *Starship*
    - The starship is a mesh loaded using the "assimp" library and the Mesh class. It has its own texture and normal map as well. The user interacts with the starship, controlling it and using it to explore the solar system. It is connected to the camera matrix to give it a third person quality, where the camera is offset at the back of the starship. It is rendered and generated in the "graphics.cpp." It draws from the assets folder and loads the .obj file and its corresponding texture file into the program. The animation is handled in the "HierarchicalUpdate" method, where every time the camera moves, the starship moves with it.
- *Sun and Planets*
    - The sun and the planets are generated and rendered in the "graphics.cpp." Using the "Initialize" function, the sun and each planet are created by coding new objects and passing them through the constructor of the sphere class, taking the amount of vertices and the texture of each celestial body. Each planet has a unique texture and normal map, including the sun. After generating the sun and the planets, the "HierarchicalUpdate" method is used to animate the spheres. The sun is first animated, placing it in the middle of the solar system. It is resized and rotates counter clockwise. This is unique to the sun, as every planet orbits the sun. Once the sun's animations are calculated, it is pushed onto an animation stack of matrices. Using a 'localTransform' variable, the animations can be calculated and kept track of. Calling the "Update" method of the sphere class updates the matrix of the object to become animated. The localTransform variable is passed through. This will be the base of each of the planet's animations, as every planet will orbit this body. The planet's use the localTransform variable to receive the top of the stack, which is the sun's animations. Each planet's animations are unique to every planet, with different rotations, orbit speeds, orbital axis, sizes, and distances from the sun, however each planet orbits in the shape of an ellipses. The planet's animation calculations follow the same steps as the sun, where once an animation is calculated, it calls the "Update" method of the Sphere class. Certain planets have moons, so their animation must be pushed onto the stack before animating the moons, then must be popped off to preserve the sun's animation.
- *Moons*
    - The moons are generated with the use of the sphere class. It is generated in the "graphics.cpp" with creating new Sphere objects in the "Initialize" function. It passes through the amount of vertices rendered and the texture of the moons from the assets folder by calling the constructor of the Sphere class. Once the model is

generated and rendered, the animation is needed. Using the “HierarchicalUpdate” method, a stack is used to create a sophisticated animation dependent on the specific planet it will orbit. The planets orbit the sun as the sun’s animations are at the top of the stack, and once the planet’s animations are set based on the sun, they are pushed onto the stack above the sun’s animations. With the top of the stack now being the planet’s animations, the moon animations can now be associated with the planet’s animations, allowing the moon to orbit the planet, instead of the sun. Different animations are applied to each different moon to account for different moon speeds, rotations, orbits, and axis rotations. Using a ‘localTransform’ variable, the animations can be calculated and kept track of. After each animation is calculated, the function calls the”Update” method of the sphere class to update the model matrix of the object. The localTransform variable is passed through. Once the desired amount of moons are added and animated to a certain planet, the planet's animations are popped off the stack and this process is repeated if subsequent planets have moons.

- *Comets*
    - The comet behaves like most of the other planets as it is generated using the mesh class. It is also then rendered and generated inside of our “graphics.cpp” with creating a new Mesh object in the “Initialize” function. It passed through the amount of vertices rendered and the normal/diffuse texture to load it in properly. Once the model is generated and rendered, the animation is needed. Using the “HierarchicalUpdate” method, a stack is used to create a sophisticated animation dependent on the specific location of the comet. This current comet will orbit the solar system in an elliptical orbit outside of neptune. We will access that position by going back to the sun’s coordinates in the stack and allowing the comment to behave as a planet. A rotation, translation, and scaling transformation is applied to the comet so that it will orbit, rotate (in this case no rotation), and scale properly inside of the solar system. Once the transformation is complete, the animation will be finished and we will call the “Update” method inside of the mesh class to update the model matrix of the object. The element will be popped off the stack and we will go back to no coordinates, as were in the sun’s coordinates beforehand.
    -
- *Asteroid Belt*
    - To load in the asteroid belt, instancing could be used but due the nature of performance being optimal with the amount of asteroids that are currently in our simulation, the use of instancing was not deemed necessary as the asteroid belt looked realistic without it. To begin the “instancing” for the asteroid belt, we needed to find where each asteroid belt lies. One of the belts lies between Jupiter and mars, while the other lies after neptune. To get the accurate placement of the

asteroid belts, we need to manipulate the placement of the belts inside of the sun;s coordinates. To begin, we will generate our asteroids inside of the mesh class which we will later generate and render inside of “grahpics.cpp”. Then, we will need to create an array of however many asteroids we need. These arrays will then be looped inside the creation of the object, during its transformations inside of “hierarchical update”, and inside of the rendering function. However many objects we have is how many times we need to loop through. Inside of “hierarchical update” we will place the first asteroid belt in a distance that is between Mars and Jupiter in the sun’s coordinates. Then we will place the second belt a solid distance after Neptune, also in the sun’s coordinates. Then we will pop the stack so that we are left in nothing’s coordinates.

Texturing:

- To correctly texture each element in oru solar system, we will need to accurately find and store diffuse textures and normal textures of each of our planets, the sun, the asteroids, and the comet. Once we have done that, we need to modify how we utilize the texture class so that we can deal with normal textures and diffuse textures. Our texture class will see which texture it is requesting, and if it is requesting a diffuse texture, it will access the DIFFUSE_TEXTURE element and locate the diffuse texture for that certain element in our folder. Each of our objects will have a normal map and a diffuse texture, besides our skybox which cannot have a normal mapping. Our mesh and our sphere classes will now remove parts of our function that asks for textures and binds them there. Now, we will include another method inside both classes that will load our textures. It will ask for the file name and the type of texture. Depending on which texture, it will go to our texture class, select the type of texture, and then load our texture into our objects. In the “graphics.cpp”, we will then render our objects in a completely new way, asking the system if it is a diffuse or normal texture. Depending on which texture it is, we will call the specific ActiveTexture that is associated with it. We will then ask if it has a normal map, if not, the diffuse will be loaded, and if it does, the normal map will be loaded. Our creation of the objects is also different now, as we will create our sphere or mesh object, then load in both the normal and diffuse texture into it. Each specific element inside of the solar system should then have the accurate normal map and diffuse texture associated with it.

Lighting:

- To correctly implement our lighting into the program, we must first begin with changing our fragment and vertex shader so that it can deal with our now new diffuse and normal textures inside of our program. The new fragment and vertex shader will take in position. Texture, and normal coordinates and do math so that it will correctly apply them with our

new lighting. Once the fragment and vertex shader are complete, we must alter the mesh and sphere classes so that their VBO and VAO will correctly showcase our new lighting. Once these changes have been made. We will then need to establish our light properties and location our light property locations inside of our shaders using our GetUniformLocation function. Once all the locations have been obtained from the shader, we will need to store that location using our ProgramUniform4fv function which will connect each location to its specific element that will be used in our program. For example, our material.spec item from shader will be located and then it will be used as matShininess that will alter a specific object;s material shininess. Once this is done for all elements, we can then utilize them to change the lighting of our solar system. We will then select the object (our sun) to be our light source. Its coordinates are 0, 0, 0, so our point light will be located at those coordinates and all surrounding objects will receive light from this object. The ambient light attribute will also affect our solar system as there will be ambient light emitted from the galaxy onto each of our planets and objects.

User Interaction:

- *Exploration Game Mode Interactions*
    - User interaction of the exploration game mode is processed through the "engine.cpp." Using the "ProcessInput" method, this method is called whilst the window of the program is open. In the "ProcessInput" method, various "GLFW" functions are utilized to register multiple key inputs from the user. Different keys change the camera matrix, changing the position, viewpoint, fov, and other features of the camera. Using global camera variables, they are updated through calling the "getCamera" method from the graphics class and getting the camera, and then calling the update method of the camera class to change the perspective and viewpoint of the camera. Returning to the "ProcessInput"method in the engine class, each key has a unique interaction with the camera to create directional changes, for example, 'W' key makes the camera move forward, and in turn the spaceship as the spaceship is dependent on the camera's matrix to move. Extra implementations include 'LEFT_SHIFT' to speed up the spaceship by increasing the cameraSpeed variable, and 'LEFT_CONTROL' to decrease the speed. The spacebar is implemented to bring the spaceship to a halt. The mouse also has interactions with the spaceship. When moving the mouse around, the spaceship will change its direction, allowing the spaceship to steer left and right. The mouse callback function is used to implement this, also using a variety of global variables like yaw and rotation. The scroll wheel also has an implementation, using the scroll callback function. When scrolling up or down, the camera will zoom in and out respectively. Global variables are used again, and this is updated by calling the graphics, then getting the camera, and calling the

update method of the camera to update the projection/perspective and field of view to create the zooming effect.

- *Observation Game Mode Interactions*
    - A function is made in the engine class to implement the observation mode. Once the key ‘M’ is pressed, the function “observationMode” is called, constituting the start of the observation mode game mechanic. Once this is toggled, the spaceship’s location is saved and the viewpoint is changed to observe the sun. Each planet is bound to a number key, starting with Mercury at ‘1’ and Neptune at ‘8’. Mouse functions and movements are preserved throughout this mode, however the original key bindings to move the camera up/down/left/right are forbidden. Pressing ‘R’ in this mode will reset any mouse movements made.

Shader Programs and Set Up:

- To allow our new program to run properly with lighting, diffuse textures, and normal textures, we must implement a new vertex and fragment shader. Our new vertex shader will have two structs that will deal with PositionalLight, and our Materials. Our positional light will affect each object in our solar system, and our Material Struct will deal with what materials will be present in each specific item. Then, we will change the layout location of our position, normal, and texture coordinates to that of numbers 0, 1 , and 2. We will use these to describe our VBO’s inside of our sphere and mesh classes. Then, we will have 2 layout bindings that in our code will deal with our diffuse and normal textures. Samp will allow us to create diffuse textures, while samp1 will deal with our normal textures. For the rest of the vertex shader, there will be math to calculate our projection, view, and model matrix positions. Next, in our fragment shaders we will reintroduce our structs and specify them again. We will also create the bindings of 0 and 1 for the respective samp and samp1. Then we will assign our hasNormalMap variable that deals with whether or not the program has detected an object is dealing with a normal texture. If it is it will call the normalize function and create the varNorm+texture(samp1, tc).xyz*2-1, otherwise it will just normalize varNorm. For the rest of the fragment shader, it will deal with specific lighting that will alter the way the materials and planets are impacted by lighting.

Graphics Pipeline:

- The pipeline begins with the window of the program, created using the GLFW library. It is created in the main and engine class. OpenGL is then initialized as well. The objects are created and rendered by getting various vertex and index data, depending on the object. The data is stored in corresponding arrays, or loaded through a model library called Assimp. For example, the asteroids and the starship are loaded through Assimp. With these objects, transformation matrices are set up using glm::translate to get the

position, glm::rotate to change its rotation, and glm::scale to change its size. It has to be in this order of translation, rotation, then scale. Shaders are then set up to calculate color and depth values of each pixel. They are responsible for performing tasks such as lighting calculations and texture sampling. Each model in the scene will have its own vertex array object (VAO) and vertex buffer object (VBO) that will hold your vertex data and index data. A VAO is a container object that holds VBOs and other states needed to draw your models. Next, the frame buffer object (FBO) is created to hold the final image, as it stores the colors and depth values of the pixels in the scene. The rendering process includes drawing the objects into the scene using various functions like glDrawElements and glDrawArrays. This will start the pipeline process starting with the geometry, as stated at the beginning. The front and back buffers of the window are swapped to display the image the user sees in the final program.