Computer Graphics (CSCI-GA 2270-001) - Final Project

Avadesh Meduri (N10537558)

December 22, 2020

1 Common Information

The code was written in Ubuntu 18.04, with a cmake version 3.10.2 and C++ version 7.5.0. I have implemented all the algorithms in the project using using standard C++ libraries. I have commented the code so as to explain what I am trying to do in each block of the function.

2 Introduction

In recent years, there has been a significant increase in the use of machine and reinforcement learning methods to solve sequential decision making problems in robotics. These learning based algorithms are often data driven, which means they improve their performance by trying different things and learning from this experience. While this capability of these algorithms is quite remarkable as they have the potential to solve complex problems which are difficult to model and solve analytically, these algorithms often do not make smart choices during the initial stages of learning. Which means that directly training these algorithms on real robot becomes infeasible as these algorithms would choose actions which could damage the robot. Consequently the importance of physics simulators has increased even more than before because they provide an artificial environment that closely mimics the real behaviour of the robot, for these learning algorithms to try different things and learn while at the same time not damage the real robot.

In this project, a simple physics simulator is developed to enhance my understanding on the topic. The developed physics engine is based of the rasterizer implemented as a part of Assignment 5. The engine can load and render any number of objects in the scene automatically with the help of a json file. It also can resize objects and translate them based on user given inputs. The physics engine is capable of automatically computing the camera gaze direction based on the desired location of the camera provided by the user. Further, the engine can render a scene as the camera is moved constantly in 3D space.

The physics simulator uses a first order euler integration scheme to integrate the dynamics of the objects in the scene. Currently, it can detect collisions between a sphere and cube. In the event that collision occurs, the physics engine uses a spring damper model to compute contact forces for the objects. The parameters of the spring damper model, the mass and size of the objects, number of objects in the scene etc.. can be changed through a json file.

The rest of the paper is organized as follows. Firstly, the organization of the simulator code along with the algorithms used is discussed. Subsequently, the results of the simulation are shown.

3 The Simulator

The rasterizer developed in the previous assignment has been restructured significantly to allow for new features. The different location of the functions and files are discussed below :

3.1 mesh loader.cpp

The mesh_loader file contains different functions that are called to initialise the physics engine.
1) load_scene() : the load scene function reads the different scene parameters from the given json file and assigns them to the vertex attributes and uniform attributes.
2) init_object() : this function takes the loaded objects in the scene and resizes the meshes based on the user inputs provided by the user and translates them to a desired initial position in the scene. These objects are then simulated later from these initial positions.
3) compute_normals() : this function computes the normals for the flat shading and per vertex shading methods for the given objects by calling the compute_normal() function. The implementation of the compute_normal function have not been altered from the previous rasterizer submission in assignemnt 5.
4) compute_transformation_matrices() : this computes the matrices required to transform the vertices of the mesh from the world frame to the bi unit volume. It assumes that the input vertices of the objects are in the world frame. It also computes the matrices for perspective projection.

3.2 object.cpp

This file contains functions that are related to transforming the object in the scene.
1) resize_object() : this function uniformly scales the object based on the user input provided in the json. This is done by updating the vertex attributes of the object using a SE3 scaling matrix.
2) translate_object() : translates the object by updating the vertex attributes.
3) locate_center() : locates the center of the object by computing a bounding box (AABB) for the vertices of the mesh of the object and then computing the center of the bounding box. This method only works for a sphere and an AABB. The hope is to extend this function for any object in the future. Also, computing a bounding box based on the vertices of the object is inefficient. The idea is to later modify this function so that it computes the center of the object after initialising it and then update the location of the center by using transformation matrices directly based on how the object moves in the simulator. Thus reducing the computation required to determine the location of the center of the object.

3.3 integrator.cpp

This file performs the update step of the objects in the scene based on the current state of the system.
1) check_collision() : This function checks if two objects are intersecting each other. Right now, it is only capable of checking if a sphere collides with cube and vice versa. If there is a collision it returns true and also computes the contact forces using a spring damper model.
2) step : this function checks if an object is either fixed (which means the object is locked in space like the floor in the scene) or not fixed. If it is not fixed, then it updates the location of the object using an euler integration scheme. Further it calls the check_collision() function to see if the objects collides with any other object (for now it only checks if there is a sphere and box collision). In which case, it uses spring damper model to compute contact forces and then integrates the system.

3.4 main.cpp

This file is the center file for the simulator. It loads all the objects in the scene, initialises objects and then renders the scene using the rasterizer implemented in the previous assignment. Further, it updates the camera parameters (camera location, gaze direction) based on the user given input on polar co-ordinates. The idea here is to use fix the camera on a sphere and as the sphere is rotated the camera location changes and the direction in which the scene is rendered also changes. The user can change the speed of rotation and distance the camera is located by changing the θ, ϕ and r. Using these parameters, the camera location and gaze direction are automatically computed such that the camera faces the origin of the scene and keeps all the objects within its field of view.

4 Results

Several videos generated by the simulator are attached along with the submission so as to show the capabilities of the simulator. In most of the videos the camera is rotated differently by changing how θ, ϕ evolve. This results in the simultion shown from different camera angles. In the videos, the same mesh for a sphere is loaded and is resized to different radius lengths and locations to show the physics engine is capable of transforming the object based on user inputs. Further based on the shading parameters provided by the user each ball has a different color. The mass of each ball is also different to show that the euler integrator is working correctly as the balls bounce with different frequencies.

In one video, the number boxes loaded is increased to 2. Here again, the same mesh is used to load a box, but these objects are scaled and translated to create two different sized and located boxes. This video demonstrates that the engine can load different number of objects desired by the user. It also shows, that depending on the location of the box the collision detection algorithm is triggered automatically. In addition, in one video, the box size is reduced so that one ball does not collide with the box. Consequently, the ball falls down and goes outside the scene. This is also done to show that the spring damper model is called only when a collision occurs.

Finally, in one video per vertex shading is used to render the image instead of flat shading to show that the two shading methods could be used to render the image.

5 Conclusion

In this project, the developed rasterizer is extended to a simple physics simulator. The engine is able to simulate several balls bouncing on box by using an euler integration scheme and spring damper contact model. The engine is able to render scenes using both per vertex and flat shading. Further, the software is capable of automatically computing camera angle and location using polar co-ordinates provided by the user. Using which, the scene can be rendered from different viewing angles. Finally, the engine is capable of loading any number of objects, uniformly scale and translate them based on user inputs.

6 Acknowledgment

I would like to thank Prof. Panozzo for giving me an opportunity to work on a physics simulator as my final project. I was able to gain a lot of experience by implementing this engine. I hope to extend the engine to simulate more complex systems and also optimize the code further for higher performance.