Zackary T. Morelli

zackmorelli@gmail.com

7/27/2021

CollisionCheck is a program that predicts collisions that will happen on the linac for RT beams. It is designed using the physical dimensions of Varian C-Arm linacs. It takes beam geometry information from Eclipse, generates 3D Meshes using classes from a third-party library (GradientSpace), and then uses methods from that library to find collisions between the patient or the couch or the breast board and the gantry, or a gantry accessory. CollisionCheck works for isocentric and non-isocentric beams, for all the electron cones and SRS cones, and for all gantry angles and couch angles. The normal analysis will alert the user to anything that is within 5 cm. There is a “Fast mode” that is slightly less accurate and is designed to be used on VMAT plans where it can take a while for the program to chug through multiple arcs.

In the GUI, the user selects the plan they want to run the program on, the area of the body represented in the patient’s CT scan (roughly speaking), if they want to use FAST mode, if they want to include a collimator accessory, and the patient’s height (which can be left blank).

All of the patient objects (patient body, couch, and breast board) are made by pulling 3D Mesh information out of the structures for those objects in Eclipse. ESAPI actually has a method in the Structure class that will return a 3DMesh object of a structure using a Microsoft class. You’ll see in the code that I take the vertices and triangle indices out of the Microsoft meshes and use them to create meshes of the GradientSpace class, which has methods for intersection queries. This is pretty straightforward. However, I also make my own “bounding cylinder” of the patient, which is the main object/mesh used to predict collisions. This object is made based off the position of the body structure and then scaled in the z-axis to try and account for the rest of the patient’s body that is not in the CT scan, based off their height. After scaling the box in the z-axis, there is then a lot of manual work that goes into making an elliptical cylinder to try and approximate the human body better. This makes a big difference, because the box’s corners can cause false positive collision alerts. All of the patient objects are made in the BOXMAKER method.

The model of the gantry is made completely separately, and in a completely manual fashion, as there is no structure in Eclipse of the gantry. It is literally constructed using knowledge of the gantry head’s dimensions, including the distance between Source and Iso, and brute force geometry. The hard part about this is making sure the gantry head model is in the same coordinate system as the patient structures. The coordinate system is based off the patient, or specifically the patient’s CT scan. So if you apply a couch kick (couch rotation) in Eclipse, you are actually rotating the entire coordinate system. This needs to be accounted for with the gantry head. The especially difficult part is creating the gantry head model for non-isocentric beams, which I added later.

After the 3D model is complete, I use methods of the GradientSpace class to do intersection and distance queries to figure out if meshes are colliding with each other. There is some complicated logic involved with deciding what is a collision and what isn’t.

All of the collisions are then shown on the GUI. Each beam, regardless if it had a collision or not, is also graphically rendered in its own window. CollisionCheck does this by calling a separate executable program utilizing the HelixToolkit package.

In addition, the program is multithreaded. It runs its collision analysis concurrently for each beam in the plan using a parallel foreach loop, which is part of the TPL. However, in order to do this, we can’t use ESAPI objects, because ESAPI simply does not support multithreading. So, we solve this in the very beginning of the program by dumping all the info we need from ESAPI to run the program into our own classes, that way we can get rid of ESAPI as soon as possible. This is done in ScriptExecute.cs. At the end, we call the GUI on its own thread and pass it all the info that we took from ESAPI using our own classes, The ESAPI script then ends, so that Eclipse is no longer frozen, while the CollisionCheck program continues running on a separate Thread on the Windows OS. We have to get rid of ESAPI to do the multithreading we want to do later in the program anyway, but doing it upfront provides us with this additional benefit.

As you might expect, CollisonCheck is quite complicated. It took me a long time to get it where it is. I’ve done a good job commenting it, so you should be able to get a good idea of how it works by reading the source code.