Phase 2: Acquisition Radar Due

Your software engineering team is responsible for the development of a full-fledged missile defense system. This system utilizes several major components from different defense vendors. You will have to implement the overall system by integrating preexisting components together. The acquisition tower, dish, and sensor have just arrived. The dish and sensor work together, atop the tower, to create a geometric wedge capable of sensing inbound objects. Because the dish and sensor were from different vendors, your team must implement the “collision detection” between the “sensing wedge” and any trackable objects. Furthermore, for this phase, your team must retrieve the initial data from the radar to verify it is properly tracking targets.

Filtering and processing raw radar data is also a large task; however, for this phase we will simply insert a rudimentary filter that provides perfect truth information (since our virtual world has perfect truth data). Implementing Guassian filtering will occur in a later phase. See Interface / Design Requirements below for more details.

Furthermore, since the Missile Turret has not yet arrived and we do not have its interface, we will also insert a simply “print” callback to display the final solution produced by the radar sensor.

**It is vital that your team analyzes the output of the supplied truth system (NyklSolution/Phase2.exe).** This includes observing all output when the commands (key presses) below are entered, when tracked objects (such as a camera and WOInterceptorMissile) enter the wedge, etc. Your **team must exactly match these outputs** (**\*including spaces and decimal precision\*** to at least 3 decimal places.

# Functional Deliverables to Demo

1. Key “1”: Launch inbound RED missile.
   1. Follows Newtonian ballistic trajectory:
      1. headingDeg, rangeMeters,timeToImpact, initialRedLaunchPos are the input parameters fully describing the inbound launch. These parameters are populated from aftr.conf on startup as well as any time “Left Shift – R” is pressed
2. Key “2”: Immediately resets RED missile state to AIMING and places it back at its original position
3. Key “Left Shift – R”: Repopulate all simulation parameters from the aftr.conf file. The changes should immediately affect the system
4. Key “P”: Prints current RED Trajectory information
5. Key “D”: Prints current Sensor information
6. Key “S”: Rotates the Radar Dish (and correspondingly the Radar Sensor) by 5 degrees CCW
7. Key “LSHIFT - S”: Rotates the Radar Dish (and correspondingly the Radar Sensor) by 5 degrees CW
8. Key “V”: Toggle visibility of radar wedge.

# Interface / Design Requirements

Any WO\* can be added to the Radar’s scan list. Once added, if that WO appears within the wedge, the radar will report it upon each pulse return. Read the comments in the header file of WORadarSensor.h; you will likely have to read them several times throughout your iterative implementation in order for the design to make sense.

The radar sensor is designed to be modular – the algorithm to filter the raw sensed data is encapsulated as a Strategy Pattern via an std::function<> (RadarCallback\_onFilterRawScanData). In this same way, the final sensed position is made available through another std::function<> (RadarCallback\_onScanDataAvailable). If a user would like to receive tracking information from the radar, one can subscribe to the radar (void WORadarSensor::subscribeToRadarTrackingInfo( const RadarCallback\_onScanDataAvailable& processedDataFunc )).

Since the radar manufacturer has supplied us their implementation, there are areas of the code that we cannot modify; for example, let’s consider:

void WORadarSensor::onUpdateWO()

{

//Do NOT modify this code - manufacturer supplied

WO::onUpdateWO();

std::vector<WO\*> targets = this->scanForTargetsWithinCurrentWedge();

this->processScanData( targets, std::chrono::system\_clock::now() );

}

Although we cannot change this code, we see that every frame, the radar’s scanForTargetsWithinCurrentWedge() method is called. This method iterates over its list of possible targets and returns a vector containing a subset of those objects currently within the wedge. Afterwards, these active targets are passed into the radar’s processScanData() method.

void WORadarSensor::processScanData( const std::vector< WO\* >& targets, const std::chrono::system\_clock::time\_point& t ) const

This method consumes a list of targets and the observation time. For each target, this method invokes the filtering lambda (RadarCallback\_onFilterRawScanData). The filtering lambda, for Phase 2, is responsible for converting the target’s current X,Y,Z Cartesian position to a polar position: RangeInM, AzimuthDeg, and ElevationDeg. The range is relative to the radar dish’s current position. The filtering lambda is the mechanism by which a user can modify the radar’s conversion from the Virtual World’s truth data to a radar’s approximated output. The filter lambda returns a tuple containing: 1) a vector containing a Vector of the approximated polar coordinate output, and 2) the tracked object’s signature ID (the WO’s ID).

The reason the filter function returns an std::vector< Vector > > is for future extensibility. Right now, the tracking is executed exactly once per frame, but imagine the radar was able to run inside of its own thread independent of the frame rate. In this threaded scenario, the radar could sense multiple data points per frame. If the filter function is still only called once per frame, it must then be capable of processing multiple returns at once. Since this design decision maximizes extensibility and future usability, the “increased complexity” is justified. For Phase 2, the std::vector< Vector > will likey have a size of 1. Lastly, the vendor, arguably, could have supplied the filter making all of this a non-issue, but that would limit its extensibility.

Finally, after the filter lambda is executed, processScanData(…) will then invoke the RadarCallback\_onScanDataAvailable lambda callback and pass in the output returned from the filter lambda. The RadarCallback\_onScanDataAvailable is the mechanism by which the final radar output is returned to the rest of the system. Another system, such as the the missile turret system or tracking processor, would likely find this information very pertinent. In future phases, this lambda is the interface between the radar’s output on the tracking processors ability to receive new information.

The radar manufacturer has delivered its acquisition radar. Your team must interface with this equipment. Your team must implement the header file delivered by the radar’s manufacturer.

# Config file parameters

The config parameters shown below can be changed during runtime and reloaded by pressing SHIFT-R. The new change should immediately affect the simulation. Sample variables and values are shown below. During your demo of this phase, these values will be set to specific number to verify proper output.

#When pressing Shift - *R*, these values are populated. This file can be modified at run #time, saved, and then re - loaded into the module *to* test new inputs without rebuilding.

#------------- RED Missile Paramters

#Launch heading of RED missile

headingDeg = 135

#Distance, in meters, RED missile will travel when launched

rangeMeters = 141.42136

#Time, in *seconds*, RED missile will be airborne before reaching its *target*

timeToImpact = 3.65

#Initial position, in x, y, z meters, defining RED missile's launch point

initialRedLaunchPos = ( 100, 100, 0 )

#-------------

#------------- Radar *Parameters*

#Specified the horizontal *field* of view, in degrees, of the Radar's wedge

radarFieldOfViewDeg = 20

#Specifies the range, *from* the wedge's apex, the radar can sense (within the wedge's fov)

radarScanRangeMeters = 400

#-------------

# Presentation / Demo

Throughout your team development, keep a running list of each person’s time spent coding, problems & issues, successes, and morale. Your presentation will describe how you interfaced with the radar. Subsequently, give a qualitative discussion of lessons learned and reflection over the Phase 2 process. Your grade will be 80% based on meeting the functional requirements using robust software engineering design principles and 20% on the knowledge & wisdom conveyed during your talk. A reason for this heavier weight on the presentation is to teach other teams your process as they will be familiar with this domain.