Autonomous Neural Network-based Missile Defense System Using Heat Signatures

Hunter Werenskjold and Carson Wilber

April 2019

1 Introduction

Air missile interception systems for national defense are complex military applications utilizing multi-domain identification, tracking, and interception technologies. A typical system is composed of forward-base radio, ground-based radar, and thermal and kinetic cameras and sensors, in addition to onboard flight control on interceptors for dynamic, real-time trajectory correction.
For the scope of this project, while we implemented a short range air defense(SHORAD) interception system[2], we focused on implementing an efficient and effective threat detection algorithm to meld these simulated data and identify legitimate threats. This algorithm will be designed to inherently ignore non-threatening aerial objects, i.e. commercial aircraft and drones.
The approach to demonstrating our artificial intelligence(AI) algorithm is in the form of a game in which the user is presented a randomly generated island country and is tasked with placing radar and SAM sites across the island in a way that will, hopefully, provide the most safety to the country under threat. For added complexity, we assume that the country is located in a popular commercial airspace and therefore must constantly differentiate threats based on learned patterns.
For this project, we will be using the SFML C++ library to provide a simplified interface for the development of our simulation. We chose SFML as it still allows many low-level features of OpenGL to be used while still permitting us to utilize many high-level functions to keep us from detracting too much into the intricacies of OpenGL. In example, something that seems simple like drawing a line and circle in OpenGL can require hundreds of lines to properly implement. This is where SFML comes in with predefined drawable objects that allow us to simply describe the scene and let the SFML rendering pipeline take care of the rest with intervention where we desire.

2 Methodology

Our primary goal is to devise an AI algorithm that will properly classify objects flying through short-range radar monitored areas. If an object is classified - or later determined - as a threat the system will notify the nearest surface-to-air missile (SAM) site to the projectile which will handle intercepting and safely deactivating the incoming projectile or threat.

2.1 Initializing Our World

The first thing we had to address when starting this project was the random generation of landmasses. We discussed many approaches, but settled on perlin noise[3] as the best approach for landmass generation due to its ease to modify and simplicity to implement. We then determined our own weighted system for determining placement of cities in the world involving the placement of cities somewhat randomly, ranking them, and then culling the top desired cities.

2.1.1 Landmass Generation

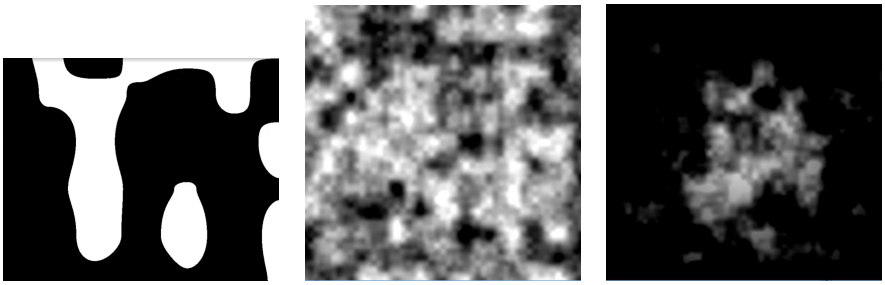
Firstly, we define the dimensions of the NxM simulation space, N for width and M for height. We then define a function that will iterate through this space and calculate a heightmap from a modified perlin noise equation that maps from a multi-dimensional domain to a one-dimensional range to generate a scalar field, $P(x, y) : R^2 \rightarrow R$. Where P is defined by the pseudo-code,

```
DEF P(x, y):
    fX = floor(x)
    fY = floor(y)

    s = N(fX, fY)
    t = N(fX + 1, fY)
    u = N(fX, fY + 1)
    v = N(fX + 1, fY + 1)

    lerp1 = cosineInterpolation(s, t, nX - fX)
    lerp2 = cosineInterpolation(u, v, nX - fX)

    return cosineInterpolation(lerp1, lerp2, nY - fY)
```

and the function $N(x, y) : R^2 \rightarrow R$ is defined by,

```
DEF N(x, y):
    n = floor(nX) + floor(nY) * 57
    n = (n * 2^13) ^ n
    nn = abs((n*(n*n * 60493 + 19990303) + 1376312589))
    return 1 - (nn / 1073741824)
```

and finally, the function cosineInterpolation$(a, b, x) : R^3 \rightarrow R$ is defined by,



(a) Boolean Rendering of Perlin Noise (b) Perlin Noise with Octaves (c) Final Perlin Noise Generation

Figure 1: Perlin Noise Generation

```
DEF cosineInterpolation(a, b, x):                    1
    ft = x * PI                                      2
    f = (1 - cos(ft)) * 0.5                          3
    return (a * (1 - f) + b * f)                     4
```

We assume that cos, floor, and PI are predefined.

By using these functions we are able to generate the image shown in Figure 1(a) - where black describes land and white indicates water. We were now generating a landmass, but this was not satisfactory, even when sampling from a more "zoomed out" perspective. After some tinkering we were able to introduce octaves, or a summation of the same perlin noise equation with exponentially decreasing amplitudes sampled at increasing frequencies - as seen in Figure 1(b). Octaves are fantastic but with each octave introduces additional linear complexity to this already intensive process, as such we only use three octaves. This is a step in the right direction, but we need to apply a simple shaping function to have the land follow a curvature such that an island will form at the center of the simulation space. To do this, we will add a scalar multiplier that approaches 0 as the sampled location approaches the edges of the simulation space - we can do this by using a distance weighting function as shown below.

$$f(v) = -\alpha\sqrt{(0.5-x)^2 + (0.5-y)^2} \tag{1}$$

Where v is the value returned by the perlin noise function at location x and y in the simulation space and α is a user defined scalar - we chose 1.7. This function also assumes that the simulation space is normalized. We then elevate the value by an arbitrary scalar e in the domain of real numbers such that $0 < e < 1$, this provides us with Figure 1(c). Finally, we specify a water level $[0, 1]$ in the domain of real numbers to determine what will be water and what will be land.

Figure 2: Rendering of Perlin Noise with GLSL Fragment Shader Code

2.1.2 Landmass Rendering

We then proceeded to use GLSL code to create a fragment shader for real-time dynamic coloring of the landmass and it's biomes. Using GLSL allows us to alleviate some of the $O(n^2)$ large-data computations being performed in the initial landmass generation. We do this with a simple piece-wise function to determine sea, beach, grass, mountains, and peaks. The corresponding psuedo-code can be seen below:

```
DEF colorLand(altitude):
    IF(altitude < 0.4 + (waterLevel*0.1)):
        return sand
    ELSEIF(altitude < 0.8):
        return grass
    ELSEIF(altitude < 0.9):
        return rock
    ELSE:
        return snow
```

where altitude is a scalar value sampled from the given heightmap - stored as a texture in vRAM - and the return value is a three-dimensional vector describing the desired red, green, and blue (RGB) color. For our project, we used the following values for coloring the landmass biomes:

```
#define sand  vec3(0.8764, 0.9, 0.5)
#define grass vec3(0.3117, 0.6764, 0.1196)
#define rock  vec3(0.8294, 0.6764, 0.1196)
#define snow  vec3(0.9764, 0.9764, 0.9764)
#define water vec3(0.0, 0.1, 0.9);
```

2.1.3 Objects and the Object Manager

Our approach to the organized existence of objects in the simulation revolved around creating an object manager that would iterate, update, and manage abstract world objects. This object manager requires the objects it calls on to have access to the world, geography, and all other objects, so to resolve this we provide a pointer to the world data to each object's update method. This could be deemed as over-exposure and a risk to internal security, but for the scope of this system and the additional complexity we would need to add for a problem that never arose we did not find this to be an issue.

As for our architecture we used to simulate objects in the world, we decided to create a virtual superclass - WorldObject - that all objects in the object manager must derive from. This enables simple cross-interaction between managing and co-dependent objects. We extend from this superclass directly for all objects except those that must be analyzed by the radar system. Objects in flight are all derived from the virtual class FlyingObject, which derives from WorldObject.

2.1.4 City Placement

For the placement of cities, we decided to take a random distribution of valid locations across the simulation space, sum the score of the location of each city, and repeating the process y number of times. We then select the top performing distribution and place the cities accordingly.

To retrieve the top performing cities from the trials we call the function performTrials(x, y) which can be seen below. This function will score the distribution and swap the distribution with the currently top scoring if found to be better performing.

```
DEF performTrials(x, y):                                          1
  FOR i = 0 to y:                                                 2
    distribution(i).score = scoreCities(x, distribution(i))       3
                                                                  4
    IF (y > 0 AND distribution(i).score > distribution(0).        5
        score):
      swap(distribution(i), distribution(0))                      6
                                                                  7
  return distribution(0).cities                                   8
```

The values x and y are the number of cities to be tested and the number of trials to perform, respectively, with a return value being the cities found to be the top overall scoring. Each city is scored over four criteria:

- The city is at least a distance d from the other cities
- The city is not located at too high of an altitude
- The city is connected by land to other cities

- The city is near water

In an attempt to normalize the scoring of each individual criteria, we had the score for each sum up to 1. This allows us to distribute weighting values to each parameter to making sure the decision of the placement algorithm statistically leans towards developing adequate placements. This algorithm does still allow aberrations to occur, which helps mimic the real-world distributions of cities in island countries.

2.1.5 Hostile Missile Manager

In order to simulate a missile-based attack on a remote island country we decided to use a managing object that times and launches missiles from the bounds of the simulation space. We do this by randomly selecting a point on the boundary of the space and assigning a missile object to the closest city in relation to that point.

The simulation of the missile objects created by this manager are kept quite simplistic. On creation, they self-generate data such as speed, direction, and size. The missiles travel linearly towards their target city and exist until either they are intercepted by a SAM site or reach their target and destroy the city. To prevent dangling pointers, the missile will mark itself as having arrived at the destination and await deconstruction by the missile defense system. This is a problem we could have fixed by using C++'s smart pointers or a more complex object system, but this would increase the scope of our project far beyond the intended goal.

2.1.6 SHORAD Defense System

To help encapsulate all the components of the missile defense system we create an object with references to all radar objects, SAM site actions, and observed projectile data. By encapsulating all the data to one object we are able to easily store the incoming data of each projectile and analyze the data to determine the threat of each independent thread. If a missile becomes of a high or immanent threat level we are able to instruct one of the closest non-busy SAM sites to intercept the projectile before contact with the possible targeted city.

Radars work by first identifying the projectile to see if it has already been observed or not. Then the radar gathers all relevant information about the projectile, such as the position, possible direction, and speed. Radars exist purely to gather data from objects within their viewing radius.

SAM sites are relatively simplistic in that they idle until a projectile has been assigned to the radar. They then are assigned a pointer to the projectile and simulate the intercepting of the missile by drawing a line to the projectile and using a time-delta accumulator function to trigger the destruction of a missile.

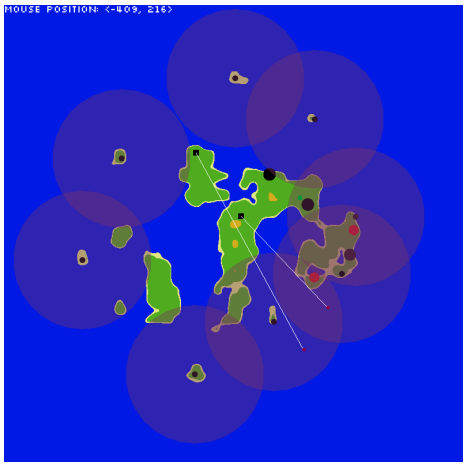


Figure 3: Running Simulation With Eight Radars and Two SAM Sites Targeting Two Incoming Missiles

By using such a function, we are able to give the illusion of an interceptor traveling across the simulation space without needing to compute another object in the world - thus adding computational complexity and cluttering the user-interface.

2.1.7 Keras and Tensorflow

In order to incorporate an intelligent agent into our missile defense system, we used the Keras library for Python3 with a Tensorflow backend and GPU acceleration during training.
Data used to train our model originated in two papers, one containing heat signatures for rocket-propelled missiles (4) and the other containing heat signatures for passenger jets (5). From the base signature measurements in these papers, we construct a dataset of perturbed heat signatures with a noise consistent with experimental simulated noise under different weather conditions in (5). This produces 10,000 samples of randomly noisy data with which to train our model.
The data ultimately used for training is normalized (subtraction of the mean and division by the standard deviation), and no validation set was used, though performance during practical application in our program is consistent with high accuracy on validation.
The model is constructed to be a sequential set of dense layers, beginning with an input of 31 data points for each heat signature and concluding with a probabilistic binary classification between "missile" and "plane" classes. The inner layers contain 200, 100, 50, and 25 neurons, sequentially.

Training was performed for $train\_epochs = 10$ and $batch\_size = 50$. Model

tuning was performed using binary cross-entropy loss of the classification and stochastic gradient descent with a learning rate $LR = 0.01$. Loss significantly reduced to near-zero and accuracy reached 100%.

In our program, heat signatures are generated using the same base signatures and slight more exaggerated random noise. At each time step, a new noisy signature is generated by each missile and plane in the world. If a flying object is within detection distance of a radar site, its signature at that time step is passed through our model to produce a classification. The difference of the missile and plane classifications are normalized to $(0, 1)$ and a moving average of the class prediction is kept as $class\_pred = 0.5 * old\_prediction + 0.5 * new\_prediction$. Threats considered $IMMEDIATE$ (> 0.9) or $HIGH$ (> 0.66) are intercepted by the nearest SAM site.

Credit is due to Piotr Plonski, developer of the *keras2cpp* utility we used to convert and use our Python3 model in C++. The library can be found on GitHub under the MIT license at (6). The MIT license is included in our source code distribution.

3 Conclusion

Our use of AI neural networks and weighting algorithms with managing classes to simulate air traffic and missile-based assaults on an island country felt quite novel if still relatively simplistic overall. We do believe that by chasing our hypothesis we could still provide a useful methodology for distinguishing possible projectiles from boost phase to midcourse phase with quite precise accuracy given heat signatures trained off of neural networks.
With the knowledge that what we were chasing was more in tune to entry-level interests, we decided to target the simulation more towards an interactive prototype to showcase an idea that could be expanded with the right knowledge. As such, allowing the user to place the SHORAD components does allow for worst case placements in which the destruction of the country is guaranteed, we know this, but the interaction is enjoyable. The most obvious simplifications can be seen in providing an accurate simulation missiles demonstrating a missile command style linear approach to their target, SAM sites using an approximation to simulate interception of a projectile, and the assault coming from from all directions. These should not detract from the overall method we are trying to show.
Short range air defense systems are quite complex and require far more understanding of real-world systems than we have time to research in one semester. Given these restrictions, we are very content with what we were able to develop in such a confined amount of time using so many algorithms, libraries, and fields of computer science that, prior, we were still quite new to. Still our project does demonstrate a working system in most aspects given the difficulty experienced in finding proper data sets to work with in generating our weights for our neural

network.

[1] https://www.sfml-dev.org/documentation/2.5.1/
[2] https://fas.org/sgp/crs/weapons/IN10931.pdf
[3] http://libnoise.sourceforge.net/noisegen/
[4] https://www.sciencedirect.com/science/article/pii/S1000936117300912
[5] https://pdfs.semanticscholar.org/4a90/a7ef4a1fc2ab45f39e00668b6885dabd49d6.pdf
[6] https://github.com/pplonski/keras2cpp