Aaron Fox
CECS 622
Dr. Elmaghraby

Homework 1

1. Search for alternative definitions of "modeling" and "simulation" and report with your comments.

Modeling: A more scientific definition of modeling as related to computing is one that "consists of writing a computer program version of a mathematical model for a physical or biological system" [1]. This is a more narrow definition which focuses on the more literal definition of modeling as applied to computing, although models can be applied, for example, to physical objects such as plastic toy soldiers to represent a war strategy. This narrower interpretation of models is a bit restrictive in that it focuses only on models that can be written via programming languages and modeling software on computers.

Dictionary.com defines modeling as "the representation, often mathematical, of a process, concept, or operation of a system, often implemented by a computer program" [2]. This is a more abstract definition of the use of models that aligns more closely to the abstract definition used in the lecture slides, yet it still emphasizes how models are still often mathematical and computer-related, which is mostly true.

A third definition of modeling, by vocabulary.com, interprets it as "anything that represents something else, usually on a smaller scale. Military commanders use physical and computer modeling to plan offensives. Modeling is helpful because it allows you to take a good look at something that is too big or impractical to see otherwise" [3]. This is a definition that was common amongst all others, for it, like the class definition, emphasized how models are a representation of anything else; the definition remained abstract enough so that it could include effectively any form of modeling, whether that's artistically, linguistically, mathematically, or any other form.

Simulation: Following the first alternative definition of modeling given above, Nature Springer then defines simulations as related to computing as simulations "that are run according to such programs can produce knowledge out of reach of mathematical analysis or natural experimentation" [1]. This is a narrower definition than that given in class, which emphasized how simulations are a translation of models into something that can be experimented on. This definition in particular assumes that those simulations can yield information that could otherwise not be obtained through nature, which, of course, isn't always true. For example, many simulations in economics do in fact yield the same results over a period of a decade, but those simulations are run in minutes because they are much faster. The use case of simulations is thus much larger than this definition claims, for they can be used to yield the same results in a more timely manner, in a safer manner, or for other reasons.

Merriam-Webster defines simulations as the “examination of a problem often not subject to direct experimentation by means of a simulating device” [4]. This essentially is a distilled definition of that given in class, for the “simulating device” defined here serves as a model that is translated into a form usable for experimentation. This definition differs in that it ties in the use of simulations to help examine a problem via indirect experimentation; this definition thus also includes the central purpose of simulations as discussed in class.

From a business point of view, businessdictionary.com defines simulations as something that accomplishes the task of “acting out or mimicking an actual or probable real life condition, event, or situation to find a cause of a past occurrence (such as an accident), or to forecast future effects (outcomes) of assumed circumstances or factors” [5]. While this rather clinical definition is very business-like, it holds true to the computer science interpretation of the word as well. In particular, this definition gives two specific use cases of using translated models for experimenting -to either find a past cause or to forecast a future situation by mimicking an event (i.e., by creating a model). Thus, this business definition holds true to the denotation and connotation given in class.

2. Find a published “simulation” research article or project report and discuss whether it follows the “simulation steps” discussed.

The research paper article evaluated for this question is titled “An Economic Simulation Study of Large-Scale Dairy Units in the Midwest” by K. Bailey et al. at the Commercial Agriculture Program of the University of Missouri [6]. The article begins by adequately describing the problem in the very first line of the abstract: “The objective of this study was to assess the economies of scale and to analyze the economic feasibility of large-scale dairy units in the Midwest.” The authors then get more specific, noting how they are focusing on evaluating the impact of economies of scale on the profitability of variable sizes based on the number of cows (“cow units”) on farms. The authors’ objective and plans are intertwined with the problem formulation: the simulation and models are to be used simply for answering the economies of scale question. The explicit plan of the study was to study this problem for “for 150-, 300-, 500-, and 1000-cow units.”

The needed concepts and influences of the model are clearly conceptualized in the text, as they acknowledge that the model needs to be a tool that incorporates “a detailed dairy enterprise with an ability to project financial statements over a 5-yr planning horizon” and that “the model consists of five interactive modules: production, feed, labor, loan, and expense.” The paper, following the simulation study steps laid out in class, thus effectively follows the model conceptualization step. The collected data used for the economic model consists of many assumptions concerning the price of milk and production equipment. In the article’s “Production and Pricing Assumption” section, the authors lay out how they found the most *common* numbers and prices used for dairy production across the country. This is probably the best approach over using the average for each price, but it is still valid to criticize the particular collected data used in the study, for, as the authors even admit, “the results of this analysis are highly dependent on the assumptions used.” The article does not go very deep into how they collected their data

modes and averages that were used by the model, and that looks to be a resounding flaw in the article.

The data and conceptualizations are then translated into the Commercial Agriculture Dairy Simulation Model, or CADSIM. This simulation is explained well in the article and the authors do an excellent job of guiding through what conceptualizations and data is used and input into the CADSIM. The article does verify that the programs in the simulator are in accordance with the recommendations and suggestions of the “Farm Financial Standards Task Force and with input from the Farm Credit System.” This proves that the model has been validated by the leading authorities of the industry standards. The authors of the article do appear to hide behind this statement and fail to provide actual proof of validation and verification of the model, however. Blind trust in the statements of these industry authorities is thus required.

The experiments of the simulation are clearly laid out for each size of the cow-units in Table 1 of the article, and the results and conclusions drawn from the simulation are elaborated upon for several pages in the Results and Conclusion section of the paper, concluding that cow-unit sizes of 500 and 1000 are the best investments utilizing the impact economies of scale. The remaining two steps discussed in class, implementation and documentation and reporting are necessarily included in this research article. There is insufficient documentation of the actual CADSIM program in the article. However, there are separate articles documenting the simulator itself.

Overall, this research article aligned very closely with the steps laid out in class, and it would not be much of a surprise if at least some of the creators of this simulation study had themselves read the simulation steps found inside the class textbook Discrete-Event System Simulation. In fact, it was surprising to not see the textbook’s methodologies at least referenced in the annotations of the article, for the paper did such an excellent job of following each step very closely.

3. Write a Simulation program and test it for a 3-Dimensional space pursuit of two spaceships each equipped with a laser beam weapon that can destroy its target if it is within a cylindrical angle of α degrees and a distance of ß. Do not assume a pure pursuit, but make any necessary assumptions for smart spaceship commanders.
Some sort of graphic (character graphic is allowed) output is required in addition to a summary report discussing your results.

Introduction: A 3D Space Pursuit simulation program was designed and implemented in the Unity game engine using the C# language. Three-dimensional space ship assets were downloaded and used in the program from the Unity Asset Store. In the simulation, both ships were equipped with “lasers” such that one ship could shoot the other ship if the other ship was within an angle of α and a distance of β. The ships had “aggression levels” that determined how aggressive each ship was in pursuing an attack on the other. Each ship was equipped with two main movement methods: attacking and escaping. Depending on the aggression level of the

particular ship, the ships would either attack or escape more often based on random chance influenced by their respective aggression levels. The simulation aided in determining the optimal aggressiveness of the ships, which could, in theory, be used for actual future space commanders in determining their space fighting strategies in the future.

Approach: Each spaceship had its own C# script which influenced their movements and choice of actions, named `ShipOne.cs` and `ShipTwo.cs`. These scripts contained all of the variables, behaviors, and movements needed for each ship's simulated actions. All of the publicly declared variables in the ship could be easily changed in the Unity Inspector editor so that many different values of the speed, aggression levels, alpha angles, and beta distances could easily be changed and the effects/results could be measured. The initial `Start` function of the `ShipOne` scripts began by first assigning a value of the RigidBody to the variable `rb` so that various physics could be performed on the ships, such as acceleration, velocity changes, rotations, translations, and so on. The script then maintained a constant forward velocity assigned by the `speed` variable set in the instructor. The pseudo-random `Random` class that comes with the Unity Engine was then seeded with the number 43 for debugging and testing purposes. Finally, the Start function ends by starting the coroutine `Movements`, which determines the actions and movements of each ship.

Inside the `Movements` coroutine, an infinite while loop runs until the script itself is disabled when either ship is destroyed. Inside this while loop, the length of the coroutine's movements is determined by the line

```
yield return new WaitForSeconds(Random.Range(1, 3));
```

which allowed forced the movements to either continue their previous movement or to change movements every 1 to 3 seconds. This allowed for rapid changing of movements as needed and accounted for a more practical use of the space commander's abilities and judgements. The while statement, in pseudo code, looks like:

```
while no ship has been destroyed
      wait 1 to 3 seconds
      if a random number between 0 and 10 is higher than the aggression level
            run the EscapeAwayFrom method
      else
            run the GoToward method
```

A higher aggression level thus meant that the script was more likely to run the `GoToward` method, which was implemented to go toward a distance that is predicted to be β away from the other ship in the future. An aggression level of 10.0 thus meant that the ship would always pursue to go the predicted β distance away from the other ship, while an aggression level of 0.0 meant that the ship would always avoid the opposing ship by rotating in the exact opposite direction and going forward with the `EscapeAwayFrom` method.

Both the `GoToward` and `EscapeAwayFrom` methods were similar in that that they first calculated the other ship's expected position, accounting for the input beta distance required for destroying the opposing ship via laser, and then calculated the target direction based on if the goal was to escape away from or to pursue the other ship's calculated expected position. Using basic vector math, pursuing the target's expected position is as simple as subtracting the current ship's position from the 3-tuple vector `expectedOtherShipPosition`, yielding the nearest beta distance away from the target direction of the ship's expected position. For the escaping away method, the vector math required involved the opposite subtraction, wherein the current ship's position was subtracted from the `expectedOtherShipPosition` vector. The ship then kept it's constant forward-facing facing velocity and then rotated towards the calculated expected position of the opposing ship.

Inside the constantly running FixedUpdate Unity Physics routine, a twice-nested if loop checks several times during every frame if the ship's relative angle and distance is equivalent to the input alpha angle and beta distance. If so, then the opposite ship is "shot by a laser," i.e. its Unity GameObject is destroyed.

Results: After running through several randomly-seeded iterations of the scripts and adjusting the parameters, it became obvious that, for the majority of the opposing ship's aggression levels, a ship's aggression level of around 7.3 out of 10 yielded the most likely chance that the ship would win the shootout. This means that a ship should pursue the enemy roughly 2.7 more times than it tries to evade its predator, regardless of if the ship is slower or faster than its opposing ship.

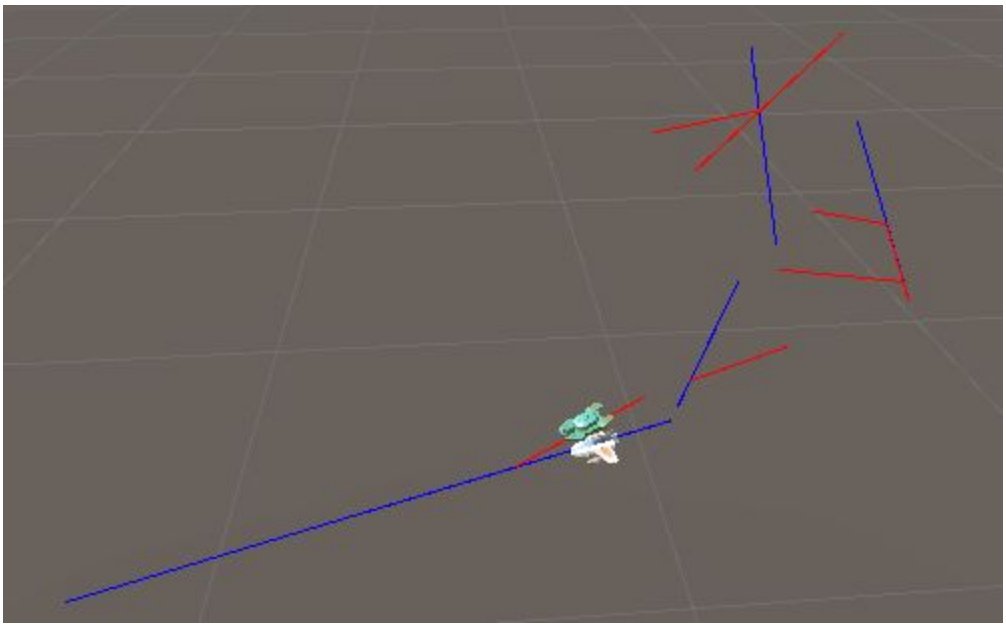
Discussion: While this simulation was relatively simple and doesn't take into account every possible variable involved in futuristic space pursuits, this program serves as a decent model on aggressive versus passive behavior in entities outside of this situation, such as possible wartime land-based strategies or in animal behavior. Some pictures below detail some of the 3D graphics that were used to visualize the simulation of the programmed model.

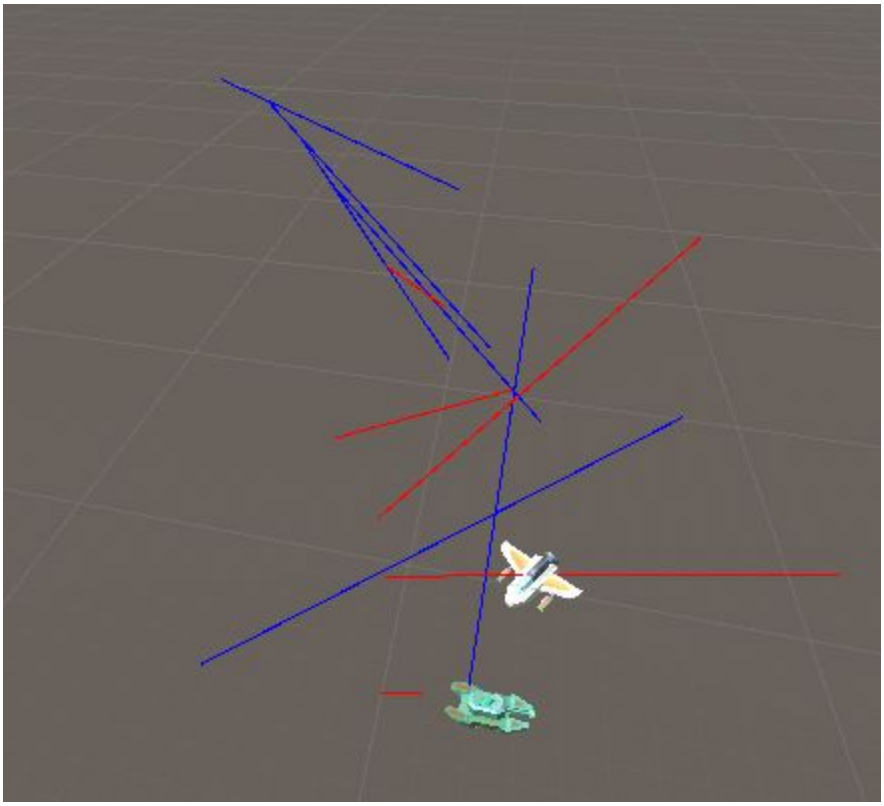
If more time were to be invested into this simulation program, more variables would have been considered. Most notably, some obstacles and a limited arena may be useful and influential in the results of this situation, for those are practical obstacles found in the real world rather than the infinite universe assumed in this model. Variable speed may also prove to provide more swift and skilled maneuvers for the ships and may prove to illustrate that avoiding the opposite ship should be a more utilized option than the assumed constant speed that was used in this model.

In addition, more in-depth strategies and strategic models could have been tested rather than the random-based strategies used in the model (although the models were random, the input parameters still heavily influenced the ship behaviors, which allowed for the mocking of specific strategies, however limited.) To be sure, the simulated behavior here has its limits due to its dearth of possible variables, but it still remains a useful model for more basic primitive strategies

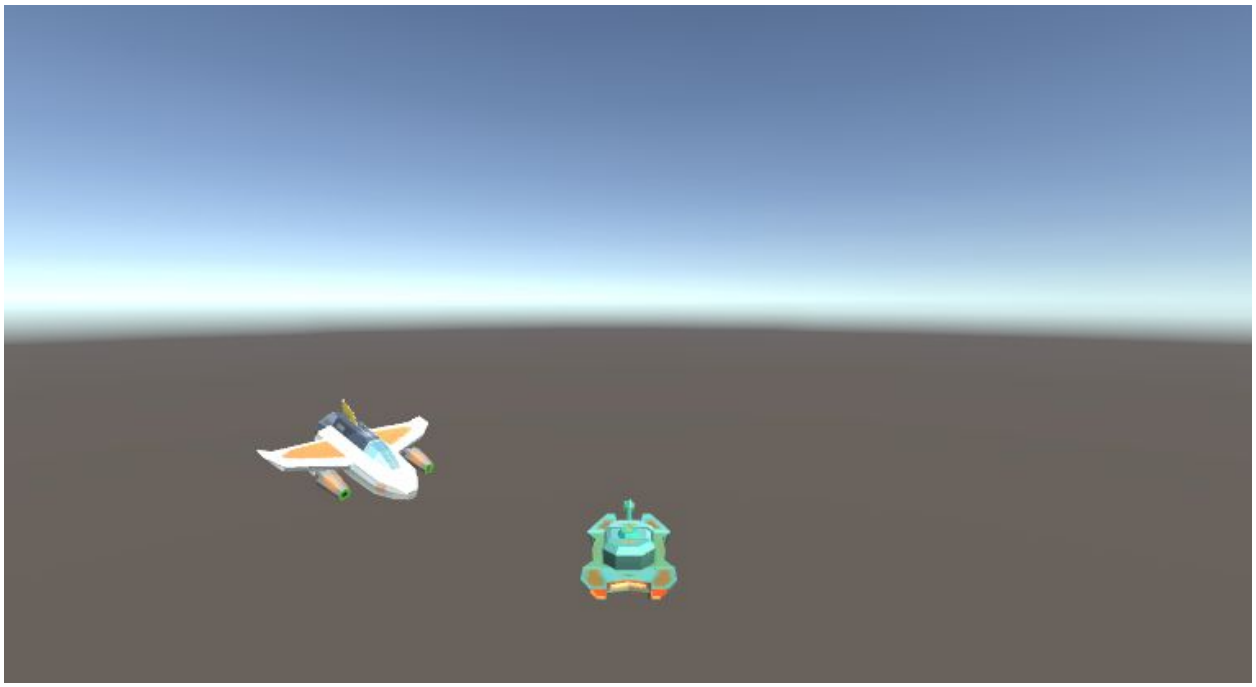
(which, ironically, probably wouldn't be used by futuristic species of animals that are intelligent enough to operate these advanced spacecraft.)

All in all, a practical and more useful understanding of simulations, models, and their uses was gained from this project. A more rudimentary knowledge of how these simulations can work and how they can be useful was gained, including an improved knowledge of graphics, the Unity Engine, vector math in simulations, and of planning out a model and simulation's variables, functions, and classes.





Temporary debug rays were drawn to show the particular routes taken by the spaceships in the simulation



Multiple camera angles were able to be used in the simulation. A third person perspective from the point of view of the tank ship is used here.

Appendix

Code used for Simulation program in Part 3:

ShipOne.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ShipOne : MonoBehaviour
{
    public Transform otherShip; // The other ship's position, angle, and speed
    public Rigidbody rb;
    [Range(1.0f, 100.0f)]
    public float speed = 50.0f;
    public float alphaAngle = 30.0f;
    public float betaDistance = 10.0f;
    public Rigidbody otherShipRB;
    [Range(0.0f, 10.0f)]
    public float aggressionLevel = 7.0f;
    public GameObject otherShipGameObject;

    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody>();

        // Maintain constant velocity in forward direction
        rb.velocity = new Vector3(0, 0, speed);
        Random.InitState(43);
        StartCoroutine("Movements");
    }

    // Update is called once per frame
    void Update()
    {

    }
```

```
    // Use this function for physics within Unity
    private void FixedUpdate()
    {
        Vector3 targetDir = otherShip.position - transform.position;

        float angle = Vector3.Angle(targetDir, transform.forward);

        float distance = Mathf.Abs(Vector3.Distance(otherShip.position, transform.position));

        float errorTolerance = 5.0f;

        Debug.Log("Ship One Angle: " + angle + "\nShip One Distance: " + distance);

        //float testingErrorTolerance = 10.0f;
        //if (angle < alphaAngle + testingErrorTolerance && angle > alphaAngle -
testingErrorTolerance)
        //{
        //    Debug.Log("Ship One: Within Angle distance of " + testingErrorTolerance+ "!");
        //    if (distance < betaDistance + testingErrorTolerance && distance > betaDistance -
testingErrorTolerance)
        //    {
        //        Debug.Log("Ship One: Within distance distance of " + testingErrorTolerance + "!");


        //        //Destroy(otherShipGameObject);
        //    }
        //}
        if (angle < alphaAngle + errorTolerance && angle > alphaAngle - errorTolerance)
        {
            if (distance < betaDistance + errorTolerance && distance > betaDistance -
errorTolerance)
            {
                Debug.Log("Destroyed Ship Two!");
                Destroy(otherShipGameObject);
                // Disable script at this point. The simulation is finished
                this.enabled = false;
                // Disable other script as well to prevent errors
                otherShipGameObject.GetComponent<ShipTwo>().enabled = false;

            }
        }
        //Debug.Log("Angle == " + angle);
```

```
        //if (angle < 5.0f)
        //    print("close");


        //StartCoroutine("EscapeAwayFrom", 4.5f);
        //StartCoroutine("GoToward", 4.5f);

        // Adjust velocity to always aim from front of ship
        rb.velocity = transform.forward * speed;
    }

    IEnumerator Movements()
    {
        while (true)
        {
            yield return new WaitForSeconds(Random.Range(1, 3));
            if (Random.Range(0, 10) > aggressionLevel)
            {

                //Debug.Log("Ship One: Escaping away from");
                EscapeAwayFrom(0.0f);
            }
            else
            {
                //Debug.Log("Ship One: Going toward");
                GoToward(0.0f);
            }
        }

    }
    public void EscapeAwayFrom(float timeToWait)
    {
        float timeForPredictedPath = 50.0f;

        // Get predicted path of other ship 5 seconds from now
        Vector3 expectedOtherShipPosition = otherShipRB.position + otherShipRB.velocity *
timeForPredictedPath * Time.deltaTime;

        // ROTATE AWAY FROM CODE
        // Determine which direction to rotate towarsds
        Vector3 targetDirection = transform.position - expectedOtherShipPosition;

        // The step size is equal to speed times frame time.
```

```
        float singleStep = speed * Time.deltaTime;

        // Rotate the forward vector towards the target direction by one step
        Vector3 newDirection = Vector3.RotateTowards(transform.forward, targetDirection,
singleStep, 0.0f);

        // Draw a ray pointing at our target in
        Debug.DrawRay(transform.position, expectedOtherShipPosition - transform.position,
Color.red);//, 1.0f);

        // Calculate a rotation a step closer to the target and applies rotation to this object
        transform.rotation = Quaternion.LookRotation(newDirection);

        // END ROTATE AWAY FROM CODE
    }

    public void GoToward(float timeToWait)
    {

        // Use timeForPredictedPath to allow ship to take the allotted required time to get within
        // a beta distance of the other ship
        // NOTE: otherShipRB.velocity.magnitude is the magnitude of the speed of the other ship
        // Time needed is distance over time or m / m / s = s
        float timeForPredictedPath = betaDistance / otherShipRB.velocity.magnitude * 60;
        Debug.Log("timeForPredictedPath == " + timeForPredictedPath);

        // Get predicted path of other ship timeForPredictedPath seconds from now
        Vector3 expectedOtherShipPosition = otherShipRB.position + otherShipRB.velocity *
timeForPredictedPath * Time.deltaTime;
        //Vector3 expectedOtherShipPosition = new Vector3(otherShipRB.position.x +
otherShipRB.velocity.magnitude * timeForPredictedPath * Time.deltaTime,
otherShipRB.position.y + otherShipRB.velocity.magnitude * timeForPredictedPath *
Time.deltaTime, otherShipRB.position.z + otherShipRB.velocity.magnitude *
timeForPredictedPath * Time.deltaTime);// + /*otherShipRB.velocity* */ + timeForPredictedPath *
Time.deltaTime;

        // ROTATE TOWARD CODE
        // Determine which direction to rotate towards
        Vector3 targetDirection = expectedOtherShipPosition - transform.position;

        // The step size is equal to speed times frame time.
        float singleStep = speed * Time.deltaTime;
```

```
        // Rotate the forward vector towards the target direction by one step
        Vector3 newDirection = Vector3.RotateTowards(transform.forward, targetDirection,
singleStep, 0.0f); // TODO: see if alphaAngle should go here or not

        // Draw a ray pointing at our target's expected future direction
        Debug.DrawRay(transform.position, expectedOtherShipPosition - transform.position,
Color.red, 10.0f);

        // Calculate a rotation a step closer to the target and applies rotation to this object
        transform.rotation = Quaternion.LookRotation(newDirection);

        // Attempt to rotate to alpha angle specified to destroy ship
        //transform.Rotate(90.0f, 0.0f, 0.0f, Space.Self);

        // END ROTATE TOWARD CODE
    }
}
```

ShipTwo.cs:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ShipTwo : MonoBehaviour
{
    public Transform otherShip; // The other ship's position, angle, and speed
    public Rigidbody rb;
    [Range(1.0f, 100.0f)]
    public float speed = 50.0f;
    public float alphaAngle = 30.0f;
    public float betaDistance = 10.0f;
    public Rigidbody otherShipRB;
    [Range(0.0f, 10.0f)]
    public float aggressionLevel = 7.0f;
    public GameObject otherShipGameObject;

    // Start is called before the first frame update
    void Start()
    {
        rb = GetComponent<Rigidbody>();

        // Maintain constant velocity in forward direction
        rb.velocity = new Vector3(0, 0, speed);
```

```
        Random.InitState(43);
        StartCoroutine("Movements");
    }

    // Update is called once per frame
    void Update()
    {

    }

    // Use this function for physics within Unity
    private void FixedUpdate()
    {
        Vector3 targetDir = otherShip.position - transform.position;

        float angle = Vector3.Angle(targetDir, transform.forward);

        float distance = Mathf.Abs(Vector3.Distance(otherShip.position, transform.position));

        float errorTolerance = 5.0f;

        Debug.Log("Ship Two Angle: " + angle + "\nShip Two Distance: " + distance);

        //float testingErrorTolerance = 10.0f;
        //if (angle < alphaAngle + testingErrorTolerance && angle > alphaAngle -
testingErrorTolerance)
        //{
        //    Debug.Log("Ship Two: Within Angle distance of " + testingErrorTolerance+ "!");
        //    if (distance < betaDistance + testingErrorTolerance && distance > betaDistance -
testingErrorTolerance)
        //    {
        //        Debug.Log("Ship Two: Within distance distance of " + testingErrorTolerance + "!");


        //        //Destroy(otherShipGameObject);
        //    }
        //}
        if (angle < alphaAngle + errorTolerance && angle > alphaAngle - errorTolerance)
        {
            if (distance < betaDistance + errorTolerance && distance > betaDistance -
errorTolerance)
            {
                Debug.Log("Destroyed Ship One!");
                Destroy(otherShipGameObject);
                // Disable script at this point. The simulation is finished
```

```
            this.enabled = false;
            // Disable other script as well to prevent errors
            otherShipGameObject.GetComponent<ShipOne>().enabled = false;
        }
    }
    //Debug.Log("Angle == " + angle);
    //if (angle < 5.0f)
    //    print("close");


    //StartCoroutine("EscapeAwayFrom", 4.5f);
    //StartCoroutine("GoToward", 4.5f);

    // Adjust velocity to always aim from front of ship
    rb.velocity = transform.forward * speed;
}

IEnumerator Movements()
{
    while (true)
    {
        yield return new WaitForSeconds(Random.Range(1, 3));
        if (Random.Range(0, 10) > aggressionLevel)
        {

            //Debug.Log("Ship Two: Escaping away from");
            EscapeAwayFrom(0.0f);
        }
        else
        {
            //Debug.Log("Ship Two: Going toward");
            GoToward(0.0f);
        }
    }

}
public void EscapeAwayFrom(float timeToWait)
{
    float timeForPredictedPath = 50.0f;

    // Get predicted path of other ship 5 seconds from now
    Vector3 expectedOtherShipPosition = otherShipRB.position + otherShipRB.velocity *
timeForPredictedPath * Time.deltaTime;

    // ROTATE AWAY FROM CODE
```

```
        // Determine which direction to rotate towarsds
        Vector3 targetDirection = transform.position - expectedOtherShipPosition;

        // The step size is equal to speed times frame time.
        float singleStep = speed * Time.deltaTime;

        // Rotate the forward vector towards the target direction by one step
        Vector3 newDirection = Vector3.RotateTowards(transform.forward, targetDirection,
singleStep, 0.0f);

        // Draw a ray pointing at our target in
        Debug.DrawRay(transform.position, expectedOtherShipPosition - transform.position,
Color.blue);//, 1.0f);

        // Calculate a rotation a step closer to the target and applies rotation to this object
        transform.rotation = Quaternion.LookRotation(newDirection);

        // END ROTATE AWAY FROM CODE
    }

    public void GoToward(float timeToWait)
    {

        // Use timeForPredictedPath to allow ship to take the allotted required time to get within
        // a beta distance of the other ship
        // NOTE: otherShipRB.velocity.magnitude is the magnitude of the speed of the other ship
        // Time needed is distance over time or m / m / s = s
        float timeForPredictedPath = betaDistance / otherShipRB.velocity.magnitude * 60;
        Debug.Log("timeForPredictedPath == " + timeForPredictedPath);

        // Get predicted path of other ship timeForPredictedPath seconds from now
        Vector3 expectedOtherShipPosition = otherShipRB.position + otherShipRB.velocity *
timeForPredictedPath * Time.deltaTime;
        //Vector3 expectedOtherShipPosition = new Vector3(otherShipRB.position.x +
otherShipRB.velocity.magnitude * timeForPredictedPath * Time.deltaTime,
otherShipRB.position.y + otherShipRB.velocity.magnitude * timeForPredictedPath *
Time.deltaTime, otherShipRB.position.z + otherShipRB.velocity.magnitude *
timeForPredictedPath * Time.deltaTime);// + /*otherShipRB.velocity* */ + timeForPredictedPath *
Time.deltaTime;

        // ROTATE TOWARD CODE
        // Determine which direction to rotate towards
        Vector3 targetDirection = expectedOtherShipPosition - transform.position;

        // The step size is equal to speed times frame time.
```

```
        float singleStep = speed * Time.deltaTime;

        // Rotate the forward vector towards the target direction by one step
        Vector3 newDirection = Vector3.RotateTowards(transform.forward, targetDirection,
singleStep, 0.0f); // TODO: see if alphaAngle should go here or not

        // Draw a ray pointing at our target's expected future direction
        Debug.DrawRay(transform.position, expectedOtherShipPosition - transform.position,
Color.blue, 10.0f);

        // Calculate a rotation a step closer to the target and applies rotation to this object
        transform.rotation = Quaternion.LookRotation(newDirection);

        // Attempt to rotate to alpha angle specified to destroy ship
        //transform.Rotate(90.0f, 0.0f, 0.0f, Space.Self);

        // END ROTATE TOWARD CODE
    }
}
```

References

[1] Springer Nature, “Computer modelling,” *Nature News*. [Online]. Available: https://www.nature.com/subjects/computer-modelling. [Accessed: 12-Jan-2020].

[2] “Modeling,” *Dictionary.com*. [Online]. Available: https://www.dictionary.com/browse/modeling. [Accessed: 12-Jan-2020].

[3] “modeling - Dictionary Definition,” *Vocabulary.com*. [Online]. Available: https://www.vocabulary.com/dictionary/modeling. [Accessed: 12-Jan-2020].

[4] “Simulation,” *Merriam-Webster*. [Online]. Available: https://www.merriam-webster.com/dictionary/simulation. [Accessed: 12-Jan-2020].

[5] BusinessDictionary.com. (2020). *What is a simulation? definition and meaning*. [Online] Available at: http://www.businessdictionary.com/definition/simulation.html [Accessed 12 Jan. 2020].

[6] K. Bailey, D. Hardin, J. Spain, J. Garrett, J. Hoehne, R. Randle, R. Ricketts, B. Steevens, and J. Zulovich, “An Economic Simulation Study of Large-Scale Dairy Units in the Midwest,” *Journal of Dairy Science*, vol. 80, no. 1, pp. 205–214, 1997.