Spacecraft Debris Avoidance in a 3D Environment

New Mexico

Supercomputing Challenge

Final Report

April 10, 2024

Team Unreal

St.Thomas Aquinas School

Team Members

Catherine Sedillo

Amelia Sedillo

Teacher

Eric Vigil

Project Mentor

James Sedillo

Contents

Executive Summary..........2
Introduction..........3
Description and Methodology..........3
  Unreal Engine 5.1.1..........3
  Microsoft Visual Studio 2022..........4
  Unreal Game Level: The Space Environment..........4
    Asteroid Classes: AAsteroid and BP_Asteroid..........6
  Shuttle..........6
    Shuttle Classes: AShuttle Class and BP_ShuttleCustom..........7
    Shuttle Motion Control..........8
  Wayfinding Class..........8
    Imminent Collision Sensing and Roll Movement..........9
  AShuttleWaypoint Class and the Wayfinding Algorithm..........10
Results..........11
Conclusions..........13
Recommendations..........13
References..........13
Acknowledgements..........14

Executive Summary

The need for vehicles and drones to maneuver through complex, 3-D environments is here in the form of small flying drone applications with future applications for sea and space. In these applications, a human may not be a suitable pilot for the vehicle due to a human's piloting limitations (e.g. slow reaction times) or communications limitations (e.g. latency) with remote-piloting thus necessitating developments in autonomous piloting. Furthermore, the vehicle may need to maneuver through a complex, debris-filled environment full of obstacles that may cause damage to the vehicle if a collision occurs. Fortunately, the software tools of today such as the free-to-use Unreal Engine game development software, allow developers to test algorithms in a virtual 3-D environment. Employing Unreal 5, Blueprints, C++, vector operations, and collision detection routines; the developer has a very useful set of tools to undertake this challenge. Using these capabilities, we developed and tested our own methods for a potential space application where a virtual shuttle must maneuver autonomously through a debris field of asteroids.

Introduction

Advances in autonomous navigation and obstacle avoidance in 3-Dimensional (3-D) environments will be necessary for a variety of current and future applications where human piloting is either inadequate or impossible. Human shortfalls can include slow human response time and/or radio communications lag between the human controller and a remotely-piloted drone. Applications for autonomous 3-D navigation and obstacle avoidance can or may eventually include flying drones beneath the canopy of a dense forest environment, a narrow canyon, or a cave-system; submarine drones swimming beneath solid ice-sheets; micro/nano-robots maneuvering through the blood vessels of a cancer patient; and a spacecraft flying through the dense gas and dust cloud of a primordial solar system.

This project seeks to develop methods for the spacecraft scenario through the use of the Unreal Engine game development environment along with the companion, C++ compiler provided by Microsoft called Visual Studio 2022 Community. The scenario consists of a single shuttle-like spacecraft maneuvering through an asteroid-rich environment to a fixed destination in an unknown solar-system. The goal is for the shuttle to avoid asteroids that interfere with its ideal trajectory to the destination as it travels toward that destination. If the shuttle reaches the destination without colliding with an asteroid, the run is considered successful.

Description and Methodology

The development tools for this project consist of Unreal Engine 5.1.1 and Microsoft Visual Studio Community 2022. One important benefit of these software tools was their zero-cost and Unreal Engine's marketplace for low-cost spaceship models and environments. On a personal note, we noticed that a majority of the games we enjoy playing were developed using Unreal Engine and wanted to explore how Unreal is used for video game development. Furthermore, The developers of Unreal provide a high-level, graphical programming language called "Blueprints" for simplified and more-rapid code development with the option to also use C++ for high-performance code. The proportion to which Blueprints or C++ is employed is entirely at the game developer's discretion.

Unreal Engine 5.1.1

Unreal Engine 5.1.1 was chosen because it was the latest version of Unreal Engine that supported the "Colony Shuttle Spaceship"[1] package that we purchased from the Unreal Marketplace. This package provided the 3-D model of the spacecraft, a space level/environment, asteroids, enemies, and a variety of sound-effects and visual-effects.

Unreal Engine itself provided an abundance of key features necessary for the development of this project. These capabilities included:

- Stunning lighting and graphical visualizations with little-to-no effort needed on the part of us, the software developers.

- Useful C++ and Blueprints base-classes for the creation of objects within the 3-D game environment.
- Useful vector-mathematics functions for locating and moving objects within the 3-D game environment.
- Object collision detection functions.
- Text, line, and spherical debugging visuals.
- An extensive library of tutorials on YouTube provided by a community of Unreal Engine developers.

Microsoft Visual Studio 2022

Since this project was developed on a Windows 10 machine, Microsoft Visual Studio Community 2022 was chosen as the C++ compiler and IDE.

Unreal Game Level: The Space Environment

The game level comprises the environment in which all of the space objects exist including the spacecraft and the asteroids. We used the level developed by [1] with some simplifying alterations (removal of enemies, unneeded asteroids, and unneeded object spawn-points). A screenshot of the unaltered package is shown in Figure 1.

Figure 1: Colony Shuttle Spaceship package as provided by the original developer in [1].

After some changes to the original, we obtained a level with the general appearance of Figure 2.

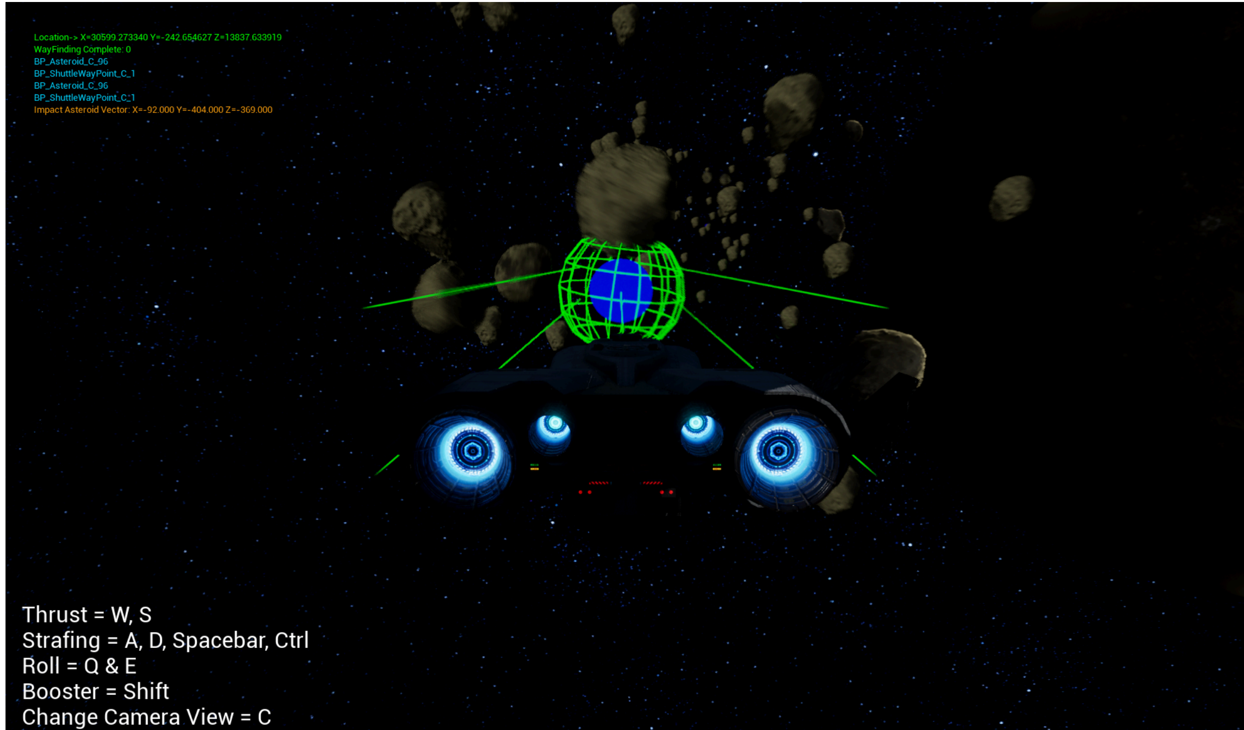

Figure 2: Our simplified level.

As can be seen, we changed the level so it consists of the shuttle, asteroids, a waypoint visual (blue sphere), and debugging visuals (green lines and a green sphere).

We added code to our level's Blueprint to program the spawning action of the asteroid objects. Our Blueprint code allows for a variable number of asteroids to spawn in random locations within a configurable box-shaped volume. This volume was configured to concentrate the asteroids within a space between the shuttle's spawn-point and the shuttle's final destination, thus creating random obstacles in the shuttle's path. The relevant blueprint code is shown in Figure 3.

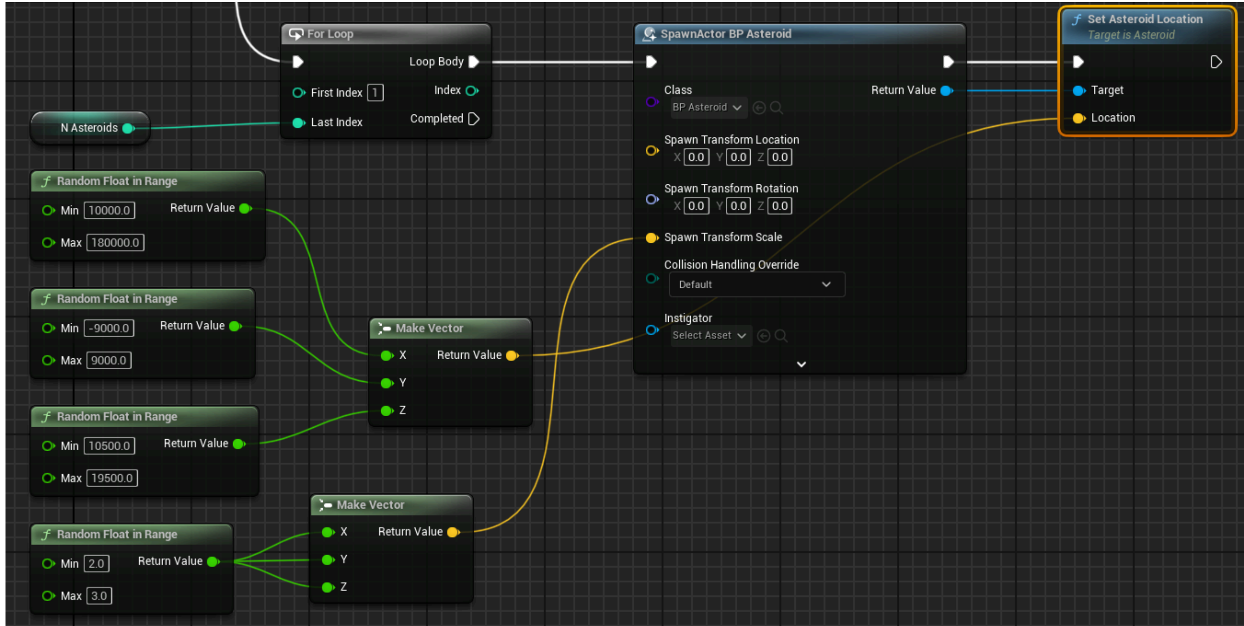

Figure 3: Level Blueprint with asteroid spawning code.

The majority of objects within an Unreal game level are considered “Actors” and derive from the AActor class provided by the Unreal library of classes. Locations of objects are specified by Unreal’s “FVector” data-type. Using three random number generators, we created the X, Y, and Z constraints for the asteroid spawning volume. Additional variation was added for the scaling of the asteroid’s size.

Asteroid Classes: AAsteroid and BP_Asteroid

As mentioned before, the asteroid objects (class “AAsteroid”) are C++ child classes of the Unreal “AActor” class. For the asteroids, minimal additional code was necessary for the child class. The additions included data member variables for storing the asteroid’s location and velocity (FVector variables), a float describing the size of the asteroid and corresponding setter and getter member functions to access these variables. From the AAsteroid C++ class a Blueprint class was derived called BP_Asteroid. The BP_Asteroid class was then available for use within the level Blueprint code.

Shuttle

The second object of importance was the shuttle. For this object we copied the “static mesh” or the 3-D model of the shuttle from [1] as seen in Figure 4.

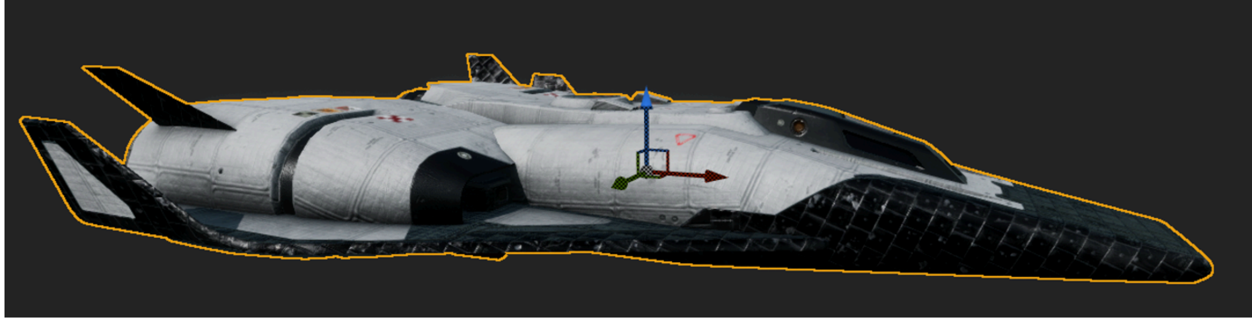

Figure 4: Shuttle static mesh model.

Shuttle Classes: AShuttle Class and BP_ShuttleCustom

Our shuttle class derives from the Unreal parent class called "APawn." Unlike the Asteroid's AActor parent class, the APawn parent class allows for player's (or AI) to acquire control of APawn-derived objects. We named our shuttle class "AShuttle" and included the following data member variables:

- ShuttleMesh (type *UStaticMeshComponent): A pointer to the 3-D visual representation of the shuttle.
- PitchPIDController (type: FPIDController): a PID controller [6] object dedicated to shuttle pitch control.
- YawPIDController (type: FPIDController): a PID controller [6] object dedicated to shuttle yaw control.
- ShuttleVelocity (type: FVector): The shuttle's current velocity vector.
- ShuttleAngularVelocity (type: FRotator): The shuttle's current rotator vector (pitch, yaw, and roll values).
- _AutoPilot (type: bool). When false, allows the player to move the shuttle through the level. Pressing the "H" key activates the autonomous feature of the shuttle
- _YawControl (type: float): the current closed-loop controller output value for shuttle yaw.
- _PitchControl (type: float): the current closed-loop controller output value for shuttle pitch.
- _ShuttleWayfinder (type: Wayfinding, to be described later): an instance of a class used for guiding the shuttle through the asteroid field.

When requesting that Unreal generate a new or derived class for you, Unreal will create the initial code for your class in the Visual Studio editor. The default member functions provided by Unreal for AActor- and APawn-child classes includes the following:

- A class constructor function with the name of your class, in our case AShuttle().
- BeginPlay(): a class member function for object initialization that is more reliable than the class constructor function.
- Tick(): A member function that is called every frame, representing a discrete instance of time. This is useful for updating your object's state and position at the game's visual frame-rate.

We added the following member functions:

- SetupPlayerInputComponent(): for binding keyboard keys to specific callback functions.
- GetAutopilot(): returns the value of _AutoPilot.

- GetYawControl(): returns the value of _YawControl.
- GetPitchControl(): returns the value of _PitchControl.
- OnMeshBeginOverlap(): an overridden function that gets called when the shuttle collides (or more specifically when the static-meshes of one or more 3-D objects overlap) with another object on the level (i.e. asteroids and waypoints).

Shuttle Motion Control

We wanted to apply Unreal's physics engine to our shuttle initially, but when pitch and yaw control became too unstable too often, we decided to simplify the model by disabling the physics engine and use a velocity-limiting model on every axis of the shuttle's motion. Below were the velocity limits we set for each shuttle axis:

- Shuttle forward thrust: fixed speed at 10,000.0 * Tick_DeltaTime. Instant acceleration to this fixed speed.
- Shuttle roll speed: fixed speed at 100.0 * Tick_DeltaTime. Instant acceleration to fixed speed.
- Shuttle pitch speed: variable speed with a maximum of 100.0 * Tick_DeltaTime, PID-controlled.
- Shuttle yaw speed: variable speed with a maximum of 100.0 * Tick_DeltaTime, PID-controlled.

These values resulted in a subjectively-satisfying motion response.

The PID (Proportional Integral Differential) controllers were applied to the Pitch and Yaw motion of the shuttle so that the shuttle would make a fluid and accurate adjustment of the shuttle's forward orientation toward the direction of the next waypoint object. This code was developed by [6] for use in Unreal Engine and was freely available via github. After installing it as a module, we were able to include "FPIDController" in our AShuttle code. After some experimentation, we settled on the following PID coefficients for both pitch and yaw:

- P: 0.1
- I: 0.0
- D: 0.001
- Max output value: 1.0

The I-value was set to zero because the shuttle didn't show any signs of having a steady-state error.

Wayfinding Class

With the shuttle now having the software components needed for automatically piloting the shuttle toward a fixed point in the level, a new object class called "Wayfinding" was developed to spawn waypoint destination objects for the shuttle to fly to in order to avoid colliding with asteroids. As mentioned earlier, Wayfinding exists as a member variable of the AShuttle class called _ShuttleWayfinder. Wayfinding's first task when instantiated, is to accept an FVector value specifying the shuttle's final waypoint location at which it will immediately spawn a waypoint object. We set this location as the far extent of the asteroid debris field. With a

waypoint now within the level, activation of the shuttle's autopilot will cause the shuttle to thrust forward and to orient itself toward the waypoint. As the shuttle is making its way through the asteroid debris field, it makes calls to Wayfinding's Update() member function during each execution of the shuttle's Tick() to keep Wayfinding updated on the shuttle's progression through the field. Update() accepts a pointer to the AShuttle object as well as a pointer to Unreal's *UWorld variable, where access to these two variables allows Wayfinding to access the full-state of the shuttle and the full state of all objects in the level (i.e. asteroids), respectively.

Imminent Collision Sensing and Roll Movement

The shuttle will proceed to the final destination waypoint unless it detects an asteroid within a fixed distance of the shuttle's top, bottom, left, and right sides. These extrema are viewable as four green lines at the extents of the shuttle's sides as shown in Figure 5 (the center green line highlights the direct path to the next waypoint).

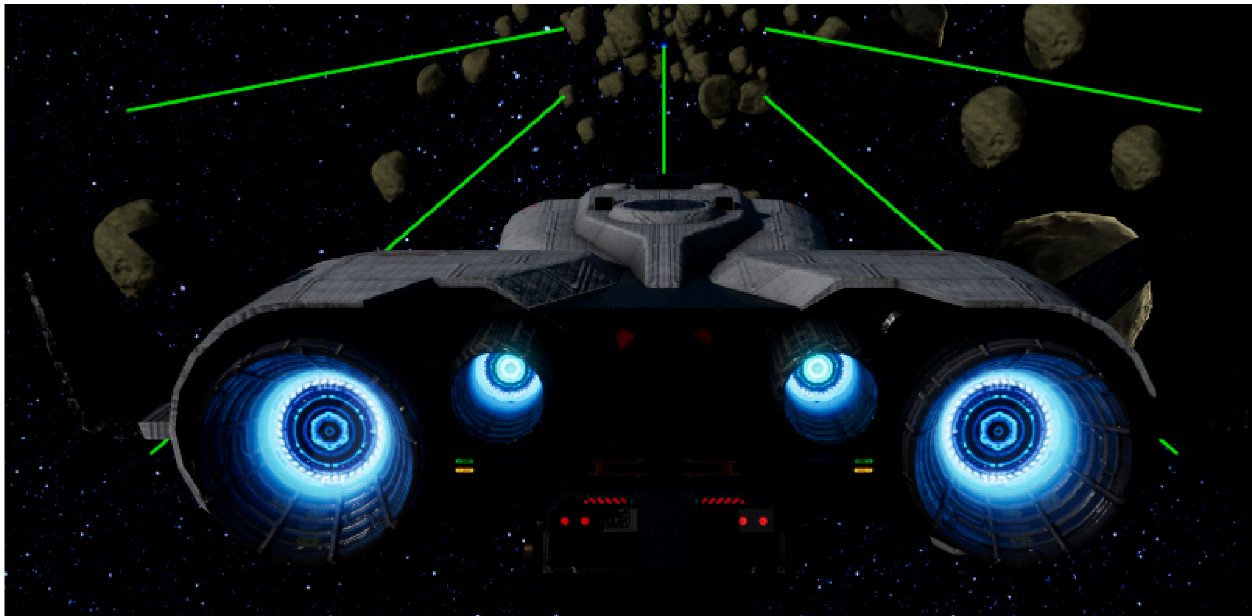

Figure 5: Four green lines at the shuttle's extents highlight the shuttle's collision detection sensing.

Using the UWorld object's LineTraceSingleByObjectType() function at each of the four extremes, the shuttle can be notified if an asteroid is intersecting these lines. If no asteroid intersects the lines, the shuttle proceeds to the next waypoint object. If an asteroid is detected, the function will return a pointer to the asteroid object. When an asteroid is detected, two actions occur: 1) the ship will begin a roll motion until the asteroid no longer intersects the lines and 2) Wayfinding's Update() function will spawn a new waypoint to guide the shuttle away from the asteroid. The new waypoint will take precedence over the as-yet-to-be-achieved, final destination waypoint.

AShuttleWaypoint Class and the Wayfinding Algorithm

Wayfinding's primary purpose is to Spawn waypoints to guide the shuttle through the asteroid debris field.

This strategy is performed with the aid of the final class AShuttleWayPoint which is another class derived from AActor. The class represents the waypoint objects that the shuttle follows and adds little additional functionality to the existing base class. Member variables of AShuttleWaypoint include:

- MeshComponent (type *UStaticMeshComponent): A pointer to the 3-D blue sphere mesh.
- AsteroidToAvoid (type *AActor): A pointer to the Asteroid object this waypoint is intended to help the shuttle to avoid.

Two simple member functions were added:

- AssignAsteroid(): sets AsteroidToAvoid.
- GetAsteroidtoAvoid(): returns AsteroidToAvoid.

With all of the classes defined, we can now explore the Wayfinding algorithm for waypoint spawning. Figure 6 represents the vectors considered for generating waypoint spawn locations.

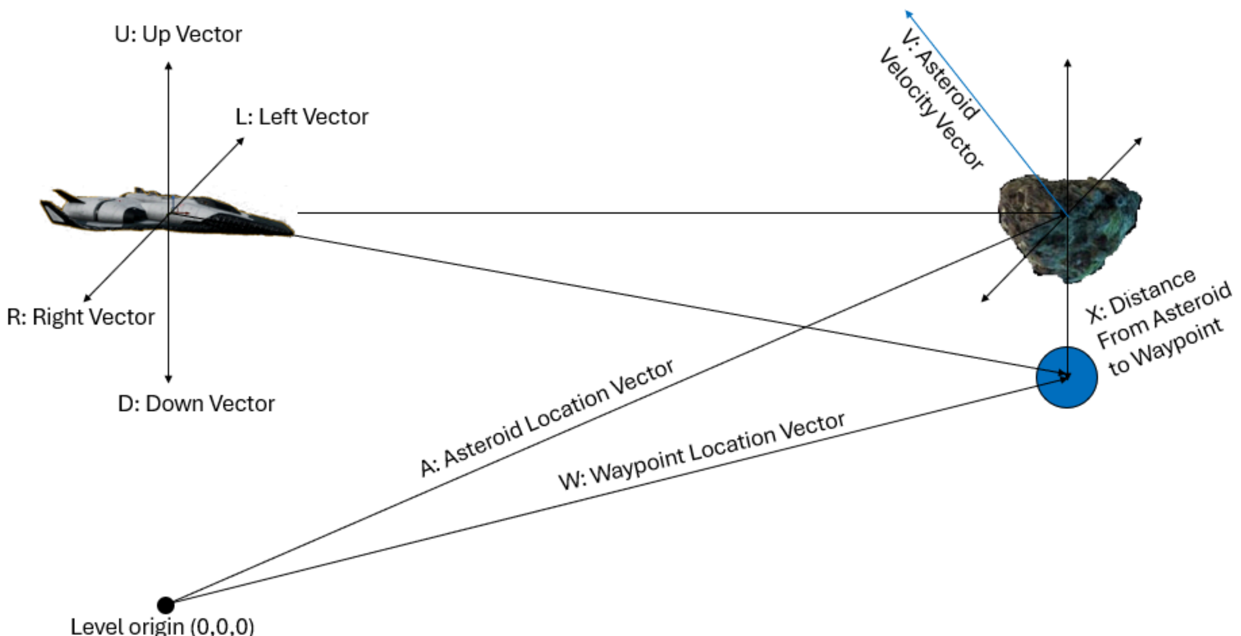

Figure 6: Vectors used for Wayfinding algorithm.

When the LineTraceSingleByObjectType() function detects an intersection with an asteroid, some vector math is used to calculate a single waypoint spawning location for the asteroid as follows:

1. Compute the dot product of V (Asteroid velocity vector) against each of the shuttles U, R, D, and L unit-length vectors. Select the U, R, D, or L vector that yields the least-valued dot product (more negative the better).
2. Compute a distance (X) from the asteroid where X = Shuttle_Radius + Safety_Margin. Shuttle_Radius represents a generalized, radial size of the shuttle and Safety_Margin is an extra distance to apply to the length of X and multiply the vector computed in 1 to give it a length other than unit length.
3. Add the vector in 2 to the asteroid location vector A, this will yield the spawn location of the waypoint at vector W for the shuttle to fly toward.

Vector W is basically a waypoint location with the following characteristics:

- Located near the asteroid so that the shuttle doesn't have to go far out of its way to avoid the asteroid.
- It's placed in one of four locations around the asteroid, so it has some "resolution" to its potential set of locations (4-possible).
- Is placed in a location opposite of the asteroid's velocity vector to assist the shuttle and the asteroid with generally moving in directions opposite to each other near the encounter point
- Is placed approximately at a right-angle to the shuttle's forward vector, thus making fullest-use of the safety margins computed in X.

The shuttle proceeds to the waypoint until it "collides" with it upon which the waypoint is destroyed and removed from the level. The shuttle then re-engages the final destination waypoint until the next asteroid is encountered and the process repeats until the final destination waypoint is reached and destroyed.

Results

To test the performance of the shuttle's obstacle avoidance, we ran ten trials where the shuttle had to traverse through a dense field of 100 asteroids with each asteroid having a random velocity vector where X, Y, and Z speeds were in the -500 to 500 range (these are unitless speeds which imparted a noticeable velocity to the asteroids). We then tabulated the number of waypoints generated and the number of times the shuttle hit the asteroid.

Run #	Waypoints Created	Asteroid Collisions	Video File	Notes
1	8	0	Unreal Engine 5 2024.04.09 - 20.23.40.07.mp4	
2	3	1	Unreal Engine 5 2024.04.09 - 20.26.21.08.mp4	
3	2	0	Unreal Engine 5 2024.04.09 - 20.27.52.09.mp4	
4	3	1	Unreal Engine 5 2024.04.09 - 20.29.21.10.mp4	
5	9	0	Unreal Engine 5 2024.04.09 - 20.30.45.11.mp4	
6	2	0	Unreal Engine 5 2024.04.09 - 20.32.07.12.mp4	
7	47	5	Unreal Engine 5 2024.04.09 - 20.33.49.13.mp4	Erratic run with spinning asteroids in the way.
8	5	1	Unreal Engine 5 2024.04.09 - 20.36.08.14.mp4	One spinner
9	3	0	Unreal Engine 5 2024.04.09 - 20.37.55.15.mp4	
10	3	3	Unreal Engine 5 2024.04.09 - 20.39.18.16.mp4	

Table 1: Run data.

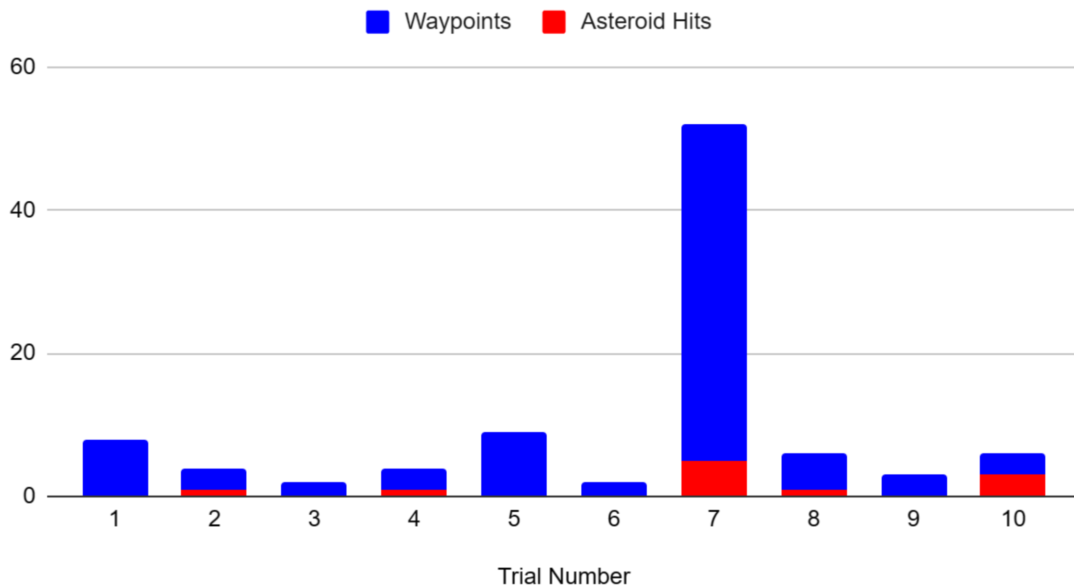

Figure 7: Graphical representation of Table 1

Here we see that 5 of the 10 runs resulted in no collisions with asteroids and one abnormal run (#7) where the shuttle encountered two clustered, spinning asteroids and regenerated waypoints for these asteroids multiple times. Taking all of the data into account the shuttle hit 11 asteroids in 85 encounters, yielding a general asteroid avoidance success rate of 87% over nine runs.

Conclusions

Our spacecraft debris avoidance method resulted in a decent baseline solution that produced results with an overall 87% asteroid avoidance success rate. The spacecraft was able to navigate around most asteroids except for when encountering more complex debris fields with tightly-clustered and spinning asteroids. The algorithm is effective for the scenarios we created, but the testing could benefit from broadening the test parameters.

Recommendations

Further developments are definitely necessary to make this a more realistic solution. Firstly, a more realistic model for the shuttle would improve the realism of the testing. Secondly, the development of an iterative path prediction method would help in cases where asteroids are tightly clustered and/or moving with higher velocities.

References

1. CGPitbull. Colony Shuttle Spaceship. Unreal Marketplace. https://www.unrealengine.com/marketplace/en-US/product/scifi-colony-shuttle-with-flight-system

2. Dan’s Game Dev Channel. (2023, December 29). Unreal Engine Tutorial Vectors 02: Position Vectors. YouTube. https://www.youtube.com/watch?v=SLUCENNoBV8

3. Forsythe, Alex. (2023, December 31). Blueprints vs. C++: How They Fit Together and Why You Should Use Both. YouTube. https://www.youtube.com/watch?v=VMZftEVDuCE

4. Forsythe, Alex. (2023, December 31). The Unreal Engine Game Framework: From int main() to BeginPlay. YouTube. https://www.youtube.com/watch?v=IaU2Hue-ApI

5. Introduction to C++. (1997). Microsoft Corporation Document No. X03-09163.

6. UPIDController. (2023, December 27). Github. https://github.com/robcog-iai/UPIDController

7. Zhenyu, George Li. (2023). Unreal Engine 5 Game Development with C++ Scripting. Birmingham: Packt Publishing Ltd.

Acknowledgements

Special thanks to Mr. Vigil for inspiring us to participate in the Supercomputing challenge, to James Sedillo for mentoring us through the supercomputing challenge, and to Elias for reviewing our project.