Team []

Team Bracket (Team 1)

CSC 440: Final Project Report

Eric Diep
Michael Dunn
Darius Haynes
Talha Khan
Zachary Neal

Table of Contents

1. Project Goals

1.1 Goal Introduction

In the evolving world of software development, software security is a difficult but important task for all developers to consider in order to protect applications from threats and vulnerabilities. Programmers often go through the process of writing code and making sure that it works as intended while dealing with close deadlines, so software security is often overlooked. This can result in reliability and integrity issues that potential attackers could exploit. Through this project, we will demonstrate the importance of developing a game with secure software principles in mind to ensure data integrity within our application. Our project team is split into two distinct groups, developers and testers, that are working together to complete our project using architectural design, risk analysis, code review, and penetration testing.

1.2 Developers and Testers

The developers in our project group are tasked with creating a video game using the popular Unity game engine. The process of creating the game will include practices of **DevSecOps** (*Developer-Security-Operators*), a methodology used for effective communication between the development team and the operations team (testers), specifically concerning security issues and mitigation strategies. The goal of DevSecOps is to achieve better security outcomes by ensuring that security experts can provide guidance and feedback at all stages of the software delivery process (JFrog). This methodology/practice is used to ensure that a project is able to progress rapidly while adhering to secure software principles.

For example, our team went through various phases of development in the creation of the video game. There was a concept phase initially where the team brainstormed ideas and thought about how to structure the game. Then, after many further stages, the team was able to craft a game experience for the final product. As the game was developed, it was tested to ensure that it was adhering to our standards for secure software development.

The core ambition for the testers is to understand the game's rules, environment, and limits of playability. The main focus of our testing will be on the high scores feature in the game. That is the most important feature that our developers will try to protect. Meanwhile, the testing team will use a commonly used cheat software called **Cheat Engine**, a memory modification tool that people use to manipulate values within a game. The software enables users to modify single-player games to gain an advantage (Soffar).

The testing team will do two evaluations to check for vulnerabilities in the project. An initial test will be conducted before anti-cheat measures are added and a second test will be conducted after the measures have been implemented. We will use **Visual Studio 2022** to provide code analysis to check for any potential vulnerabilities in the codebase. After understanding the results from the code analysis software, the testing team will discuss with the developers any potential issues

that need to be addressed. This process will be repeated throughout the project's development in order to ensure that all potential vulnerabilities are addressed and mitigated in a timely manner.

2. Description/Background

2.1 Game Overview

Unity is a popular game engine used to develop video games that are enjoyed by millions of people. However, most game developers do not consider secure software engineering principles when developing their products. With our project, we will develop a simple game in the Unity game engine and verify that the game is developed with secure software principles in mind.

2.2 Unity

Unity is a cross platform game development engine released in 2005 by Unity Technologies (Axon). According to Arm Limited, a game engine is 'a software development environment, also referred to as a "game architecture" or "game framework," with settings and configurations that optimize and simplify the development of video games across a variety of programming languages.' It can include a variety of engines to help streamline the game development process such as physics, artificial intelligence, and 2D or 3D graphics rendering engines (*Arm Limited*). Using a game engine helps developers save time since the framework created by the engine covers a variety of required components necessary for game development. Instead of having to implement rendering technologies from scratch, developers can utilize the tools provided by a game engine and save years of time.

There are a variety of popular and successful games that were developed using the Unity game engine such as:

- *Angry Birds 2* (2D, Mobile)
- *Beat Saber* (VR)
- *Subnautica* (3D, PC/Console)
- *Cuphead* (2D, PC/Console)

Each of these games are from a variety of different genres, but they were all made using Unity (Drake). This variety shows that Unity is a versatile engine that has proven technology used to power some of the most popular indie games.

Cost & Availability

Unity is provided for free for personal use and has a variety of plans for professional use with the cheapest professional version costing $2,040/year. Most developers use the Unity Personal plan since it is available for individuals and small organizations that make less than $100K in revenue/funds raised in the last 12 months (Dealessandri). Unity can be downloaded for free from the Unity Technologies website here: https://unity.com/download. It is available on Windows, Mac OS, and Linux (specifically Ubuntu and CentOS). No other tools were needed for

this project as all assets sourced were either designed by us or provided free of charge on the Unity Asset Store.

Benefits

Ease of Use
The game engine is popular for its ease of use with beginners or indie developers. It is free of charge and allows developers to write games in C# rather than C++, which makes game development easier with a more defined structure like C# compared to the older C++ (Dealessandri). Since Unity is free of charge and uses a simpler language like C#, developers can create games more efficiently. With a low barrier to entry, our developers can complete the project quickly and efficiently.

Availability of Assets
Additionally, Unity also features an asset marketplace called the Unity Asset Store that contains large amounts of free and low cost options for game assets such as 3D models, 2D sprites, music, and more. The store contains over 84,000 assets from a variety of publishers that developers can utilize in their projects (Dealessandri). This marketplace is very beneficial to developers that want to prototype or reduce development times since quality assets can be used instead of designing assets from scratch. Some of these assets can be utilized in our project in order to reduce development time and cost.

Drawbacks

Unity Promotes Bad Code Practices
Unity was not originally designed as a game engine. Instead, it was designed for web development with JavaScript, so it was not meant for large game development projects in C# (Dealessandri). Unity attaches scripts to game objects, so the methodology of separating data from code is difficult for developers and it becomes harder to understand larger projects.

Unity Releases Updates Prematurely
A drawback to Unity is that they release updates regularly that have bugs in tools provided with the engine (Dealessandri). Sometimes versions are unstable, so developers usually stick to older versions of Unity so that their projects are not affected. This can be a drawback since developers are not able to access new features since instability with basic features can discourage developers from using newer Unity versions.

Unity's Impact on Software Development

Unity is one of the most used game engines for indie game development. Due to this, a large number of beginner developers will learn how to use game engines by learning Unity first. Therefore, Unity Technologies has a large responsibility in making sure that games developed using the engine are secure and trusted by users considering how many users could be affected by security flaws found within games made using Unity.

In fact, Unity Technologies released their internal Secure Software Development Life Cycle (SSDLC) as an open source project so that other developers can benefit from the same software development practices that they use to ensure security in their software tools (Caldwell). Using this life cycle model, developers can help find software vulnerabilities early in the project's development and mitigate their impact on the final product. Unity Technologies also hosts a bug bounty program and maintains a responsible disclosure policy in order to ensure that code vulnerabilities are patched in a timely manner with minimal impact. Unity lists all patched vulnerabilities on their website at https://unity.com/security ("Unity Security").

The most recent vulnerability disclosed by Unity Technologies was a remote code execution vulnerability discovered in October 2022 by Michael DePlante of Trend Micro Zero Day Initiative. Importing FBX or Sketchup associated 3D modeling file types were susceptible to memory corruption vulnerabilities that could lead to an attacker performing remote code execution. The issue was fixed in a patch on January 30th, 2023 and the remediation strategy for developers is to update their Unity version ("Unity January 2023 Security Update Advisory").

This vulnerability did not affect our project since the project runs on 2023.1.11f and the patch version for this vulnerability was a previous version, 2023.1.0a26. Although this vulnerability was fixed and Unity does not have a long list of vulnerabilities, the patch still took three months to release. This could be a risk to Unity projects in the future if not publicly known vulnerabilities are not promptly fixed.

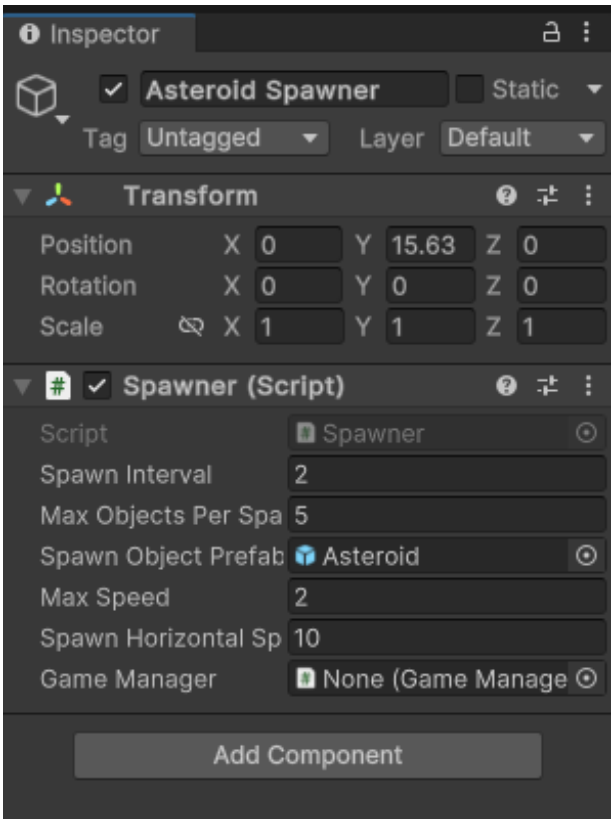
Subjectively, we believe that Unity is more viable for our project due to our experience using Unity software, the abundance of free tutorials and assets made available to Unity developers, and its ease of use. C# is easier for developers to understand and use compared to C++, so writing and giving feedback on the code is trivial for our development team. Although there are some risks with Unity releasing updates prematurely and taking an extended period of time to mitigate vulnerabilities, the benefits of using a developer friendly environment outweigh the potential risks.

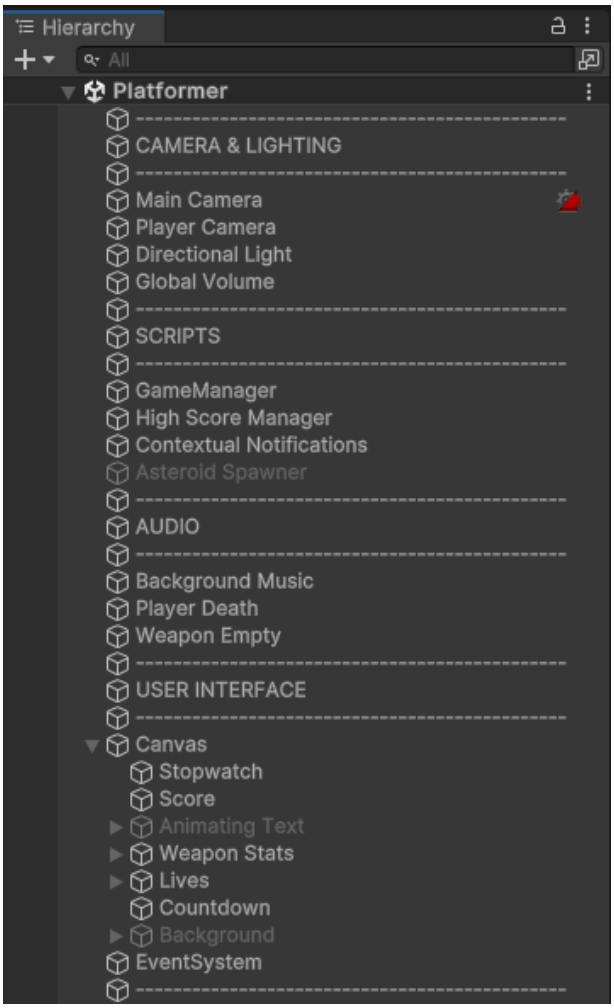
Unity Fundamentals: Scene and Component-Based Architecture

Games in Unity are scene based. Each portion of the game can be separated into a different scene. This makes it easier for developers to optimize their games since only the most relevant objects are present in each scene. For example, there could be a scene for the main menu and a scene for the game world. Scenes could also be individual levels of a game if necessary.

Additionally, objects in Unity are component based. Each object has components that make up its shape and function. Components are scripts that define physical objects within the scene. Scripts are C# source code files that can be attached to objects within Unity, hence why they are referred to as components. Unity uses C# as well as a visual scripting language to write code for projects. However, our project will only use C# code since we are experienced in developing applications in C#. Unity uses C# since it is a versatile and popular open source language. There are a variety of programming tutorials available online for C# compared to C++ since C# is more beginner-friendly, so it was the perfect choice for Unity.

Components/scripts can have their properties reassigned in the Unity "Inspector" window to customize their features and behaviors when the game is running. This allows developers to define variables in their scripts without immediately assigning a value to them. This prevents Unity from having to recompile the C# code in the scripts every time that a developer wants to change a variable's value, which reduces development time.





Unity features a parent-child relationship hierarchy system for objects within scenes, so it is easier for developers to find as well as create more complex behaviors between objects. Our project features a well defined hierarchy with objects separated by category. The headings present in the hierarchy such as “Camera & Lighting” along with the line separators are not necessary for the game to run, but they help developers understand the project design faster.

2.3 Code

2.3.1 Version Control

Our team is using GitHub for version control within our project. GitHub is a free code hosting service that helped our team collaborate and kept our code safe and secure. Included in our report is a link to the formatted GitHub repository that can be found in Appendix 1. Our project specifically used GitHub Desktop for tracking changes within the project. Using the desktop app instead of running Git commands through the command line helped us avoid making common mistakes such as overriding other developers’s work or deleting files accidentally.

2.3.2 Code Style

Our project is written using the C# naming conventions standards such as PascalCase method names and camelCase variable declarations. We also use the return early pattern where the objective is to reduce the amount of nested if statements by evaluating invalidating conditions early and only continuing to the rest of the method if those conditions are validated as true.

For example, instead of writing code like this:

```C/C++
void ValidationMethod( ) {

	if ( firstCondition ) {

		// Do initial work.

		if ( secondCondition ) {

			// Do secondary work.

			if ( thirdCondition ) {
				// Complete work here.
			}

		}
	}

}
```

We write code like this:

```C/C++
void ValidationMethod( ) {

	if ( !firstCondition )
		return;

	// Do initial work.

	if ( !secondCondition )
		return;

	// Do secondary work.

	if ( !thirdCondition )
		return;

	// Complete work here.

}
```

This helps our code remain readable for others. Other code styles used in the project are mostly personal preference such as line spacing and spacing between parentheses to improve readability.

2.3.3 Code Tools

In our project, we are using Visual Studio 2022 as our code editor and IDE (integrated development environment). Visual Studio 2022 features Unity C# integration, so it helps us with finding the best ways to optimize our code. It also helps us with following naming conventions and ensuring that unused code does not remain in the final build.

3. Game Design

3.1 Asteroid Crashers

The game, *Asteroid Crashers*, features an astronaut that has to avoid being hit by asteroids by either moving out of the way or shooting asteroids to gain points. With only three lives, the astronaut must try to survive as long as possible. Once the astronaut has been hit by an asteroid or falls off the map three times, the game will end and the player will be able to see their score and a list of high scores.

3.1 Story

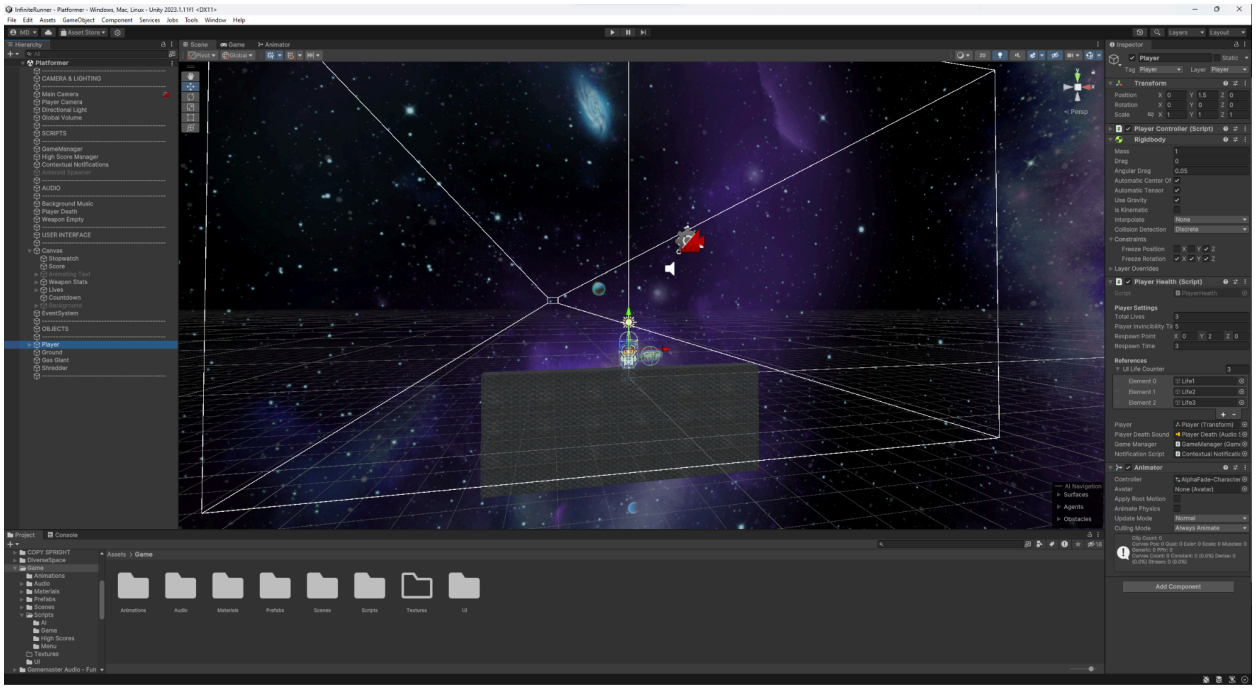
In the far future in another solar system, an astronaut is tasked with protecting the space station from an unknown enemy that is guiding asteroids towards the space station. They must use their rocket launcher gun to destroy as many asteroids as they can while also not getting hit or falling off the platform.

3.2 Level Design

Asteroid Crashers features a single, infinite duration level that allows the player to play for as long as they are able to survive the onslaught of asteroids falling towards them. Once their lives are up, then the game will end and they will be directed towards a high scores scene with a list of their top scores.

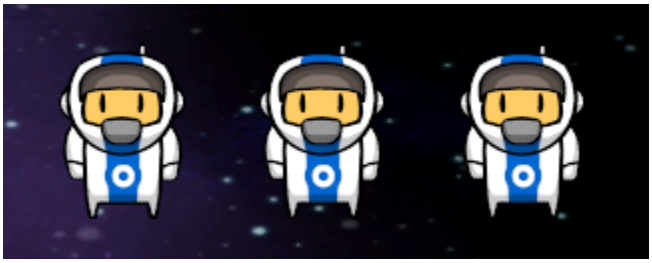
The game features a platform that is slightly smaller than the size of the player's screen. Originally, the platform was intended to cover the entire playspace. This required the use of invisible barriers surrounding the platform to prevent the player from walking off screen. However, if the player decides to run into a wall, they would be able to stick to walls if they keep pushing themselves on the wall due to the way Unity physics work. If the force is continuously applied and the player does not release the key, the player's rigidbody will continue to exert force on the barrier and will be stuck in place until that movement key is released.

After trying numerous tweaks to fix the issue with the character controller, we decided to go with a smaller platform that allows the player to fall off the edges. This adds an additional element of tension since low gravity jumps in the game can cause the player to overshoot the platform and end up falling to their death. Therefore, all of the barriers could be removed and the player should theoretically only get stuck to the platform if they fall off and time it correctly. However, without touching the ground, they will be unable to jump back up onto the platform and falling asteroids will be able to hit them if they are stuck to the platform for too long.



Our project uses a 3D scene with a 2D orthographic camera in order to blend 2D and 3D objects together. It also allows for us to use the Unity NavMesh pathfinding system to implement AI navigation into our project.

3.3 Lives



The game features a traditional life counter system that was inspired by classic games such as Pac-Man and Galaga. Lives are displayed in the top right corner as copies of the astronaut. Each time the player dies, one of these copied astronauts disappear. This is an effective way to show the player that those are their lives left rather than explaining it in a tutorial. Once all three astronauts disappear and the final life is used, the game ends and players are directed to the high scores scene. This works by having the player health script communicate with the game manager script when the player runs out of lives. This C# code snippet handles the player death process and delegates work to other methods when necessary. The player is respawned after a predefined amount of time with the **Invoke()** method call at the bottom of the **PlayerDied()** method.

PlayerHealth.cs:

```
C/C++
public void PlayerDied( bool playerFell ) {

	if ( !playerFell && invincibilityTimeLeft > 0 ) // Player is currently
immortal, no need to subtract lives.
	return;

	player.gameObject.SetActive( false );
	playerDeathSound.Play( );

	// Make sure that the current scene is not the main menu scene. That
scene features an AI agent, but it has infinite lives.
	if ( SceneManager.GetActiveScene( ).name != "Menu" ) {

	UpdateLifeCountUI( );

	livesLeft--;

	if ( livesLeft < 0 ) { // No lives remaining, send player to the game over
screen.

	gameManager.GameOver( );
	return;

	}

	// Show "Respawning..." Text on UI
	if ( notificationScript != null )
	notificationScript.ToggleTextVisiblity( "Respawn", true );

	}

	Invoke( nameof( RespawnPlayer ), respawnTime );

}
```

3.4 Asteroids

Asteroids play a pivotal role in the project's gameplay. The system runs on a time interval that spawns a variable number of asteroids every few seconds. The interval between asteroid spawning is predetermined, but the randomness of the number of asteroids spawned increases the game's tension. The asteroid spawning logic is shown below.

Spawner.cs:

```
C/C++
// Spawn objects forever until the current game ends.
private IEnumerator SpawnObject( ) {

  while ( this.enabled ) {

        yield return new WaitForSeconds( spawnInterval );

        int numObjectsToSpawn = Random.Range( 1, maxObjectsPerSpawn );

        // Create each object based on numObjectsToSpawn's value.
        for ( int i = 0; i < numObjectsToSpawn; i++ ) {

              // Set random position based on horizontal spread and a random
velocity based on the maxSpeed.
              Vector3 newSpawnPosition = transform.position + new Vector3(
Random.Range( -spawnHorizontalSpread, spawnHorizontalSpread ), 0, 0 );

              GameObject newSpawnedObject = Instantiate( spawnObjectPrefab,
newSpawnPosition, Quaternion.identity );
              Rigidbody rb = newSpawnedObject.GetComponent<Rigidbody>( );

              rb.velocity = new( Random.Range( -maxSpeed, maxSpeed ),
rb.velocity.y, Random.Range( -maxSpeed, maxSpeed ) );

              rb.angularVelocity = new( 0f, 0f, Random.Range( -1f, 1f ) );

        }

        if ( gameManager != null )
              gameManager.AsteroidsSpawned( numObjectsToSpawn );

  } // End of while loop

} // End of SpawnObject( )
```

3.5 Weapon System

Weapons are used in the game as a primary way for the player to influence the game around them. The player can fire rockets towards asteroids and if the rocket collides with the asteroid, the player scores a point and the asteroid is destroyed. Yet, if the rocket does not hit an asteroid, the outcome could become interesting. For example, if the rocket hits the player when it falls back down, the player loses a life. This will cause the player to consider future repercussions for their actions and plan accordingly.

The weapon system was developed with modular design and features numerous properties that can be fine tuned to design the right weapon for the project. With this system, additional weapons could be developed to keep the gameplay exciting.

Asteroids and missiles both include a projectile impact script that determines object behaviors between colliding physics bodies. Object collisions are handled by this method below:

ProjectileImpact.cs:

```
C/C++
private void OnCollisionEnter( Collision collision ) {

	// Current Object is Asteroid. If the other object is an asteroid or a
missile, early return here.
	if ( isAsteroid )
	if ( collision.collider.CompareTag( "Asteroid" ) ||
collision.collider.CompareTag( "Missile" ) )
	return;

	// If this collision has already been handled, early return.
	if ( hasHitTarget )
	return;

	hasHitTarget = true;

	// Hit Asteroid
	if ( collision.collider.CompareTag( "Asteroid" ) ) {

	gameManager.AddScore( );
	Destroy( collision.gameObject );

	}
	// Hit Player
	else if ( collision.collider.CompareTag( "Player" ) ) {

	if ( collision.gameObject.TryGetComponent<PlayerHealth>( out PlayerHealth
playerHealth ) ) {

	playerHealth.PlayerDied( false );

	}

	}

	// Explode Asteroid/Missile
```

```
        Explode( );

}
```

By using a single script, interactions between physics bodies can be easily updated without having to change code in multiple scripts for each interactable object.

3.6 High Scores

After the game is over, players are sent to a high scores screen where their top ten high scores are listed. This list has been limited to ten scores because of space limitations in both the user interface and size on the disk. We did not want the high scores list to store every score recorded since that could lead to large file sizes eventually on the disk (especially if we started tracking more data within the score files). Therefore, we limited the number of scores to ten to align with more traditional, classic games such as the ones mentioned in the “Lives” section. The overall score for each player attempt is calculated using a simple equation listed below:

```
C/C++
float overallScore = ( timeSurvived * survivalPointsPerSecond ) + (
asteroidsDestroyed * pointsPerAsteroidDestroyed )
```

Time survived and asteroids destroyed are used to calculate the overall score used to order the top ten list of scores based on the player’s performance. We did not want to encourage players to stay alive for as long as possible without shooting any asteroids, so each asteroid destroyed grants the player 10 points while each second grants the player 1 point towards their score. These numbers can be modified to encourage/discourage player strategies in the future with more testing.

3.7 Cut Content

Originally, we were intending to feature multiple weapons and power ups that the player would be able to use to make the game more engaging. However, due to time constraints, we were forced to stick to one weapon and no power ups. Although, the weapon system created for this project has the ability to be modified to add more weapons in the future since it was designed to be modular and adaptable. This content can be incorporated into future versions of the game as well as more polish to the game mechanics to make the game more interesting.

3.8 Future Plans

Future plans for the game could include: multiple weapon types, powerups, variable difficulty, multiple locations, and other game modes such as: alien invasion, tower defense, etc. With the modular design of the current product, these features could be implemented without much difficulty. Still, each new addition would require implementing more secure software engineering principles such as further validation checks in order to prevent cheating.

3.9 Securing the Game

The game will use local storage to save player scores. Due to this, score files will be encrypted using AES encryption to avoid tampering. Saved high scores are stored as a class while the game is running and are serialized into a JSON file before being saved to disk at the location: C:/Users/**CurrentUser**/AppData/LocalLow/Team Bracket/Asteroid Crashers/scores.json.

Validation checks will also be performed on scores in real time and after the player finishes the level in order to ensure that the high scores have not been tampered. If any of the validation checks fail, the score will be discarded and the game will end. This will not prevent attackers from targeting other portions of the game's code, but it will at least ensure that the most important feature of the game, high scores, cannot be affected.

There are two validation techniques used for the anti-cheat implementation in *Asteroid Crashers*. The first method is a simple check within the Update() method (runs every frame) that will check to see if the number of asteroids destroyed has changed dramatically since the last assignment. If so, the game will go to the anti-cheat screen.

GameManager.cs:

```
C/C++
private void Update( ) {

	if ( asteroidsDestroyed > totalAsteroidsSpawned || asteroidsDestroyed > 
prevAsteroidsDestroyed + 3 ) {

		SceneManager.LoadScene( "Error" );

	}

	if ( !gameOver )
		gameRunningTime++;

	if ( stopwatch.GetStopwatchTime( ) > gameRunningTime + 5f ) {
		SceneManager.LoadScene( "Error" );
	}
```

```
}
```

These checks would be most likely the first to fail if the attacker was dedicated enough to understand that they would need to change the totalAsteroidsSpawned variable before changing asteroidsDestroyed. After that, they could increment the score by 3 each time while also incrementing prevAsteroidsDestroyed. Same with the timeSurvived check. However, this is challenging to do with tools and would require more effort from the attacker to achieve their goals.

If these validation checks fail, there is a secondary validation method that is delayed until after the attacker completes the game. This validation method is run once the game is completed and will determine whether the high score is saved.

GameManager.cs:

```
C/C++
private bool ScoreValidation( float timeSurvived ) {

	bool isScoreValid = true;

	// If the number of destroyed asteroids is greater than the amount
spawned, invalidate the score.
	if ( ( asteroidsDestroyed > 0 ) && ( asteroidsDestroyed >
totalAsteroidsSpawned ) )
	isScoreValid = false;

	// If the time between asteroids destroyed is less than the weapon's
actual fire delay, invalidate the score.
	if ( ( Mathf.Abs( timeSurvived / Mathf.Max( 1, asteroidsDestroyed ) ) -
weaponScript.FireDelay ) < ToleranceEpsilon )
	isScoreValid = false;

	// If the total number of asteroids spawned exceeds the amount possible
per interval, invalide the score.
	if ( ( ( totalAsteroidsSpawned / timeSurvived ) *
spawnerScript.SpawnInterval ) > spawnerScript.MaxObjectsPerSpawn )
	isScoreValid = false;

	// If the total time survived is greater than the time the game has been
running plus 5 seconds, invalidate the score.
	if ( timeSurvived > gameRunningTime + 5f )
```

```
        isScoreValid = false;

        return isScoreValid;
    }
```

These three checks will determine whether the score should be invalidated and return the result of the validation check. By postponing this check until the end of the game, the attacker has to study the software more to understand how it works. Even if the first validation check fails, this check will hopefully prevent high score tampering.

Additionally, the game will be compiled using IL2CPP (**I**ntermediate **L**anguage to **C++**) instead of Mono (C# compiler) so that the source code will be harder for attackers to understand.

3.10 Mono

Mono is the default scripting backend for Unity that compiles the C# code at runtime using a methodology called just-in-time compilation (JIT). However, some platforms do not support this type of compilation, so Mono is not effective on every platform (“Mono overview”).

3.11 IL2CPP (Intermediate Language to C++)

IL2CPP is the other option used for building Unity projects. This is an alternative to the default Mono compiler that can support an extensive variety of platforms. It uses a method called ahead-of-time (AOT) compilation that compiles the project during the build process instead of during runtime. This compiler works on the backend by converting MSIL (Microsoft Intermediate Language) code such as C# scripts into C++ code and then uses that to create a native binary (executable) for the selected platform (“IL2CPP Overview”).

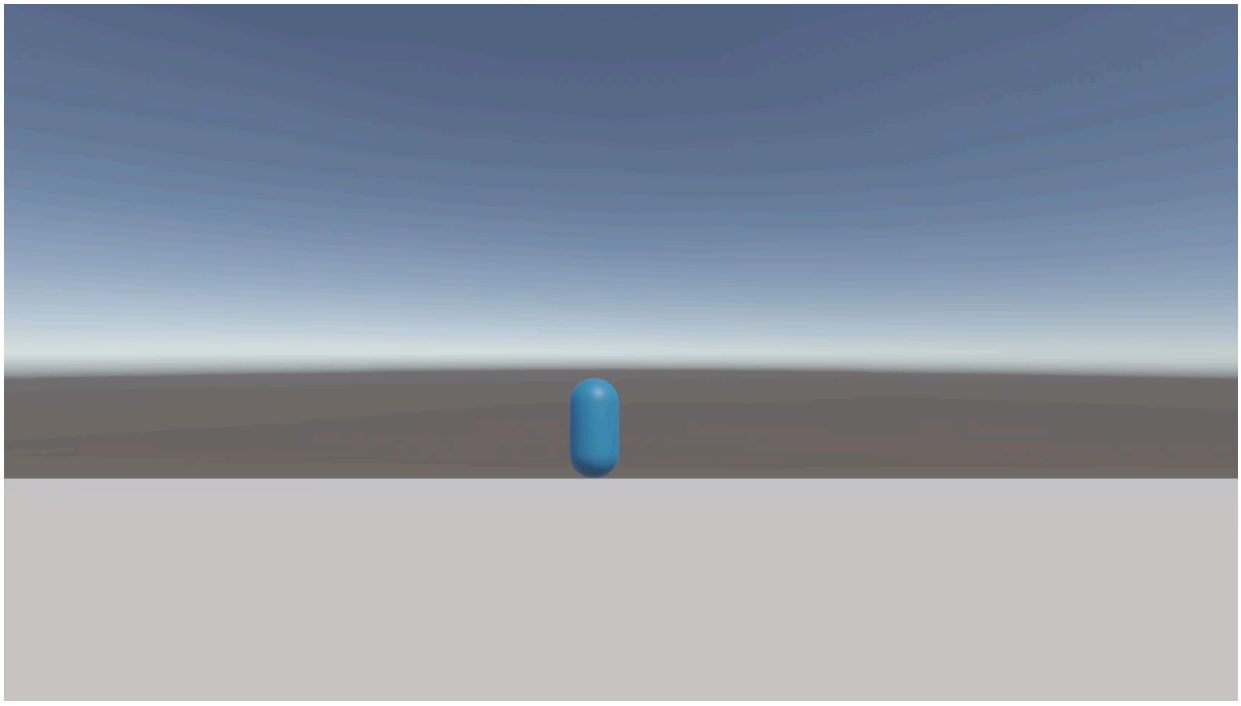
3.12 Mono vs. IL2CPP

IL2CPP compilation can lead to increased performance in a variety of platforms. Nevertheless, IL2CPP build file sizes will be slightly larger and compile time will be longer considering the build has to include machine code using this compilation process. In most cases, Unity would pick Mono if both compilers are supported on a certain platform because of this drawback (“IL2CPP Overview”).

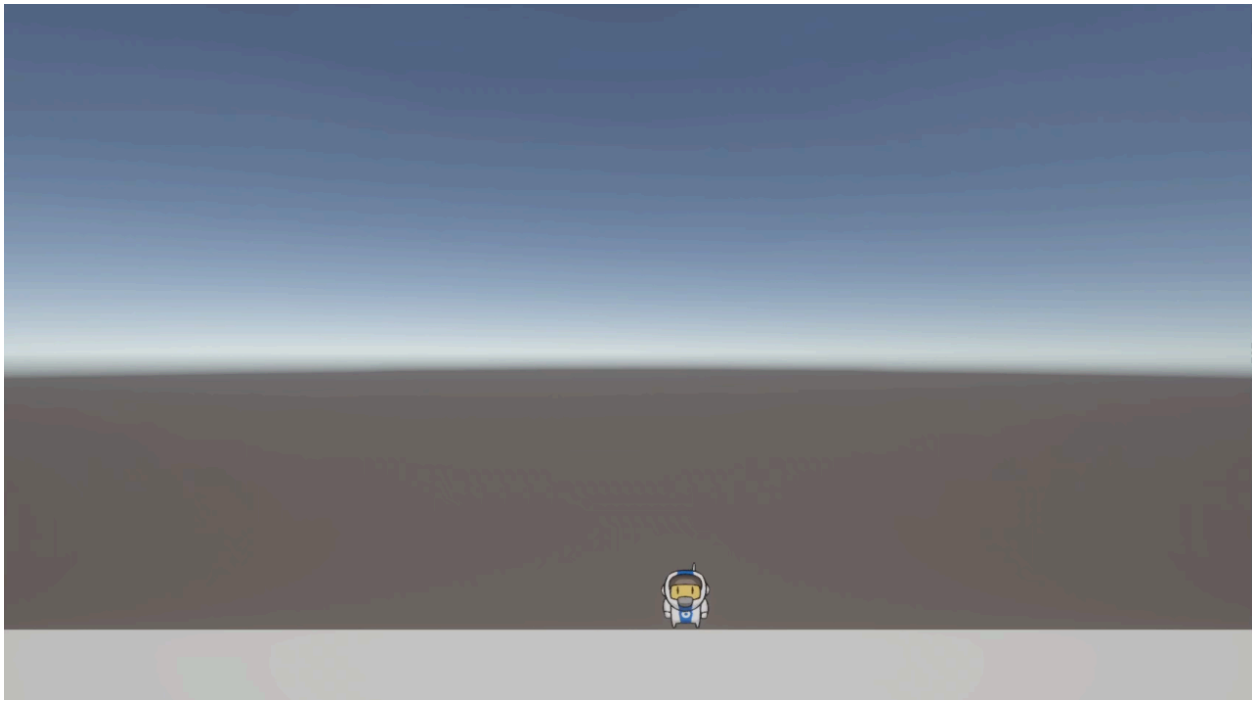
We chose IL2CPP for this project since the code created by the conversion to C++ is more obfuscated than the code compiled using Mono. This should make it slightly more challenging for attackers to interpret the decompiled source code since the translation between C# and C++ is very different (Parihar). Unity does not include the original C# source code within build files, but it can be reverse engineered using software tools trivially.

Therefore, obscuring the code in any way possible to make it harder for an attacker to reverse engineer should be beneficial towards ensuring that the software remains secure. The increased build time due to this choice will not be an issue since Unity provides a built-in editor runtime that can allow for fast incremental development without building executables each time. Consequently, the build process that IL2CPP uses strips any unused bytecode that the final application does not use, so that would help prevent attackers from exploiting unused code in the project (“IL2CPP Overview”).

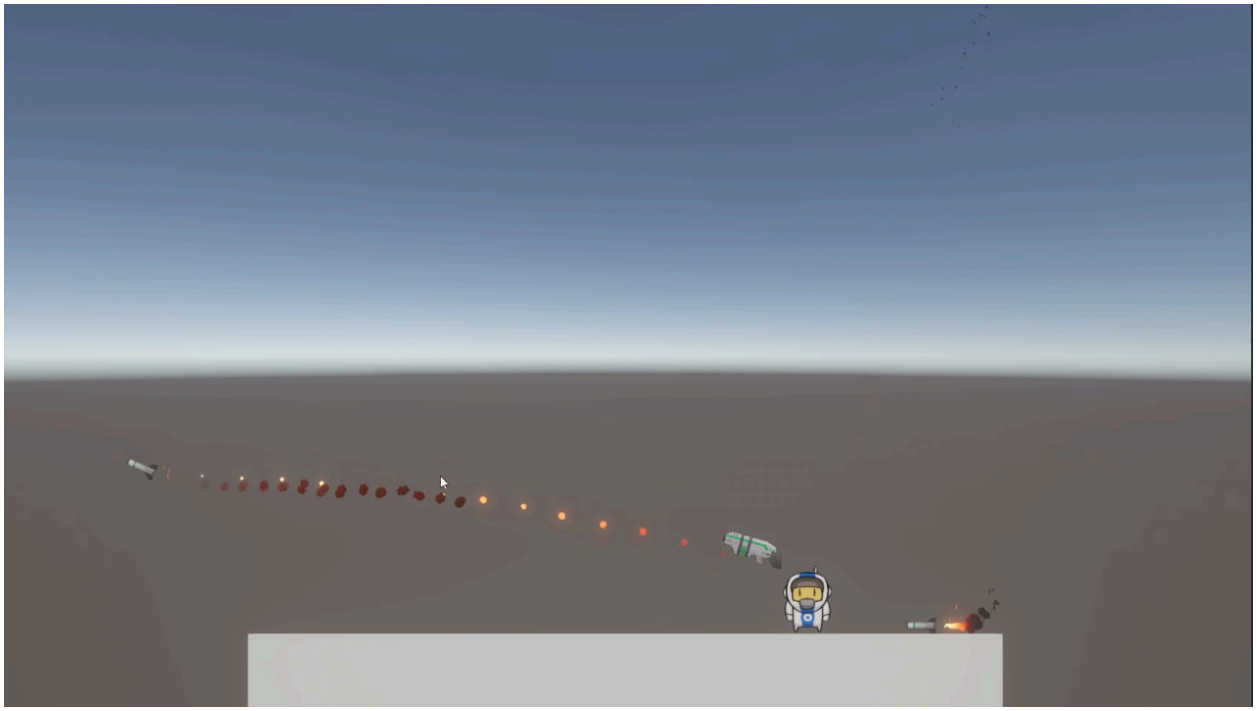
3.13 Development Progress Screenshots:



Alpha 0.1.0: Basic character controller and hitbox created.

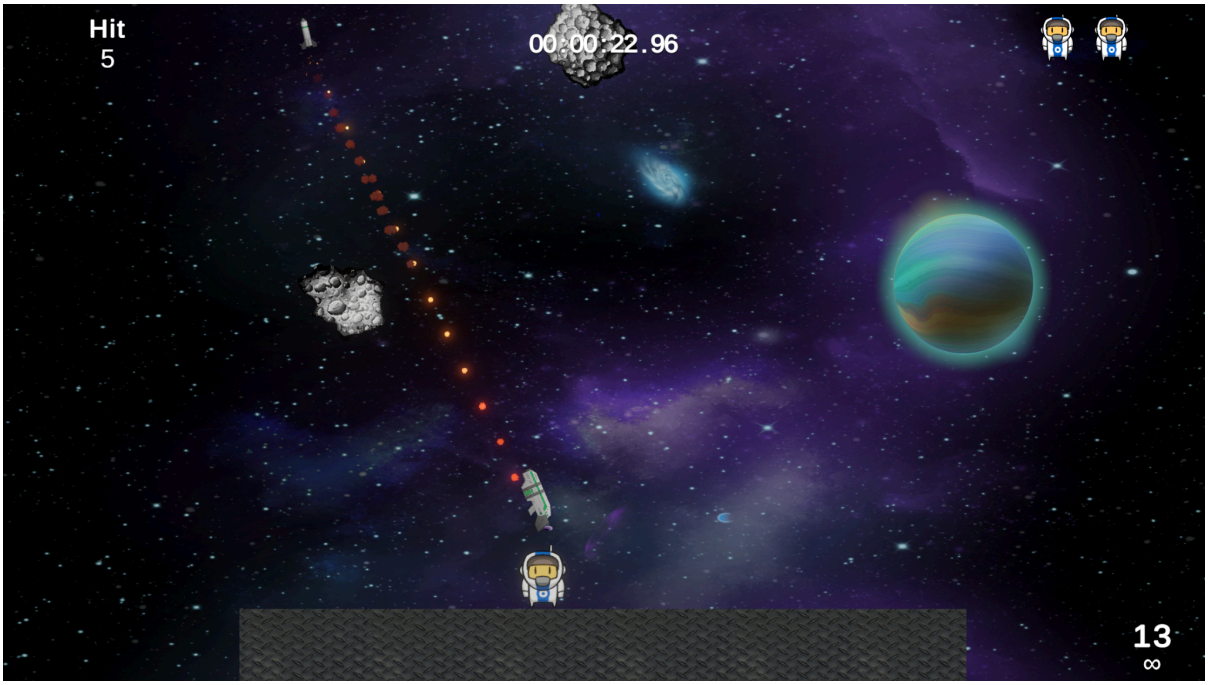


Alpha 0.2.0: 2D sprite character added and animations connected to character controller.



Alpha 0.4.0: Weapon system added and projectile firing was tested. Additionally, the platform was shrunk.

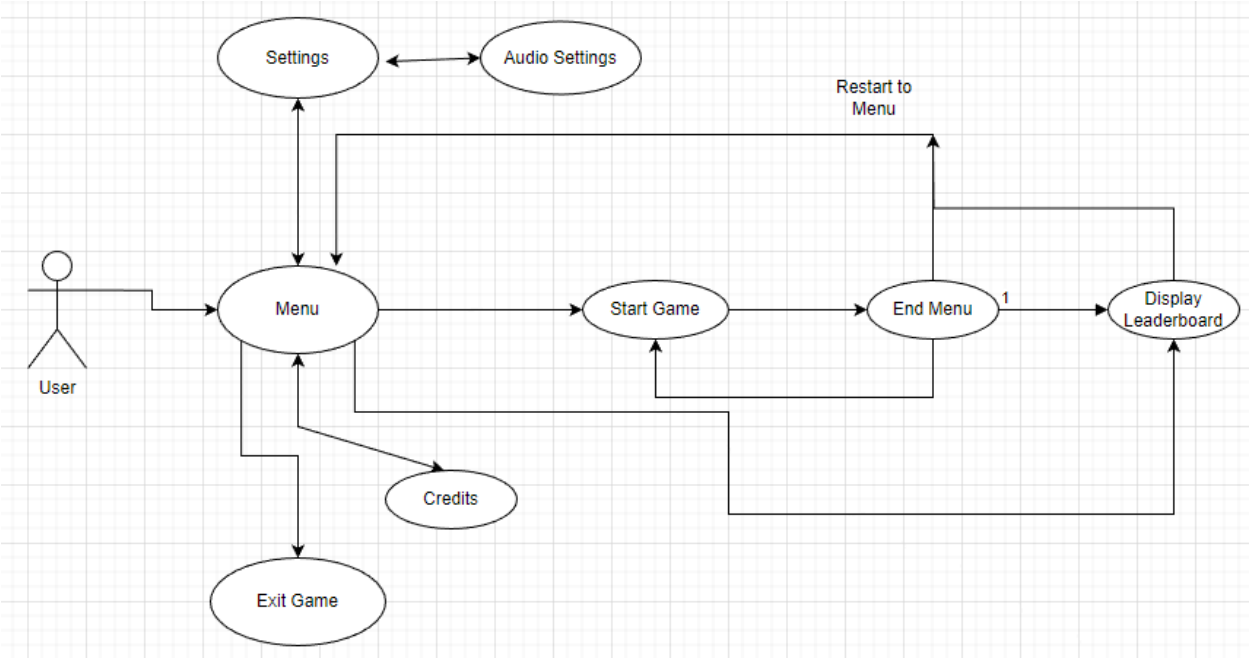
Beta Version 0.8.0: Background scenery and UI added.



Final Version 1.0.0: Weapon information user interface added and “Lives” user interface updated.

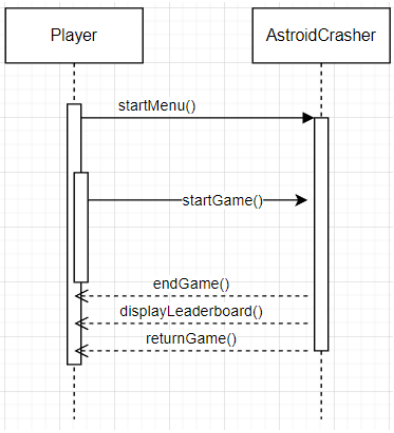
4. Architectural Design

The design of this project follows the standard principles of the Unity game development engine. It is a high level design structure that focuses on a player directly interacting with the game during runtime. The user should be able to effortlessly navigate through the game's menus by clicking on labeled buttons on the screen. This is shown below with the use case of a standard player interaction with the game.



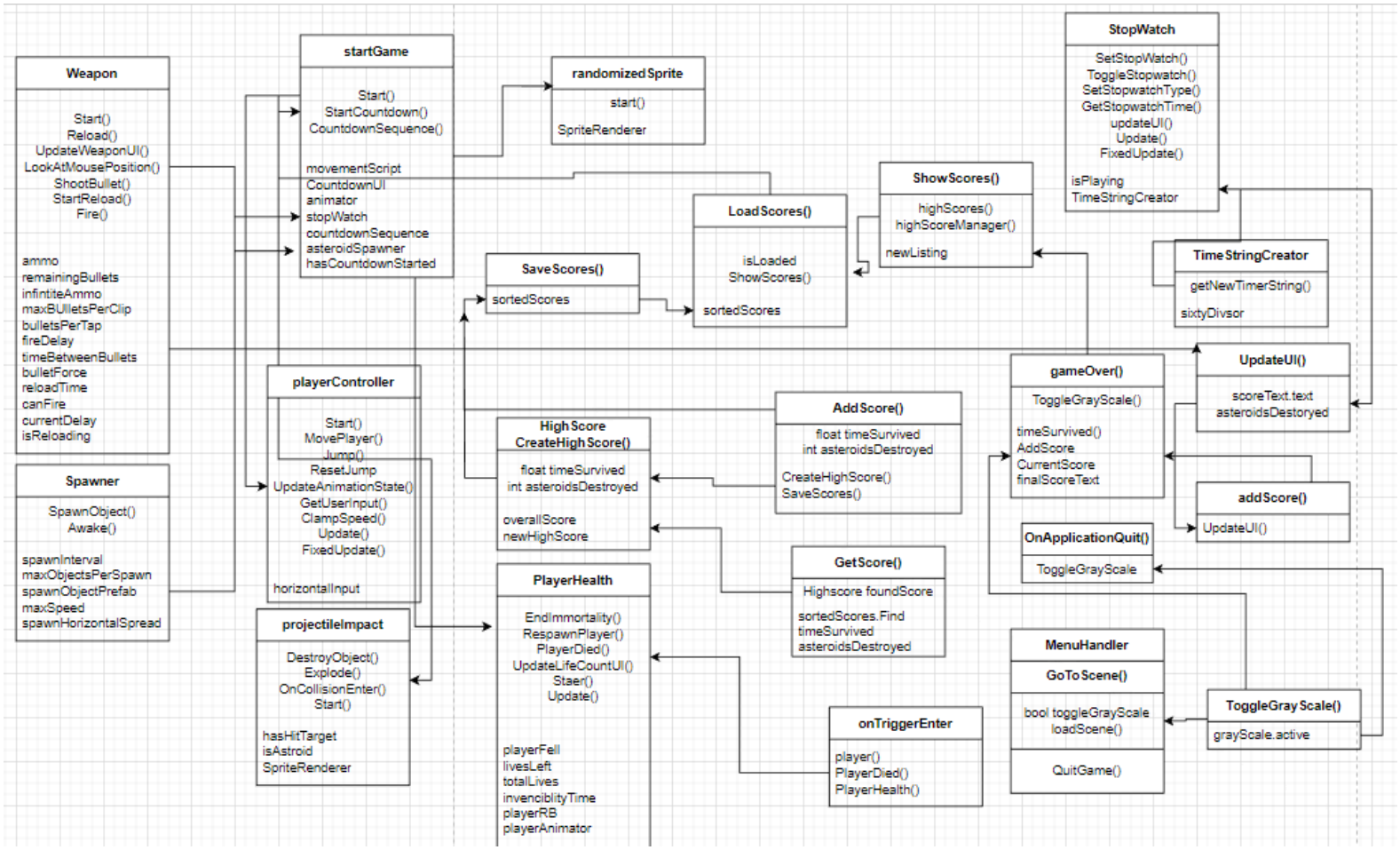
After the user launches the game, they will be prompted with a list of options on the main menu. Once the game is launched the player will have 5 options to choose from:

- Start Game - Plays the game.
- Credits - Shows a sequence of every asset used and everyone involved in the creation of the project.
- Settings - Allows for the user to change audio preferences and lists the control scheme for the game.
- High Scores - Shows a list of player high scores.
- Exit Game - Closes the application.



The sequence diagram shown above is used to demonstrate the standard interaction between the game and the player during runtime.

The class diagram shown below demonstrates all of the classes used in the project.. There are many different classes being used at the same time, so it looks convoluted onto a class diagram. However, splitting code into multiple classes helps to ensure that the project can be expanded upon in the future with interchangeable code.



5. Secure Software Methodology

Our secure software methodology contains architectural design, code review, penetration testing, and abuse cases. For the code review, we utilized Visual Studio's code analysis feature. Visual Studio's code analysis tool was used to find potential design, performance, and/or security problems. There we found one-hundred plus non critical suggestions for mostly third party code. All of the suggestions were harmless considering that they consisted of syntax, naming conventions, and statement simplifications. Among those suggestions were third party code that we did not end up using and were removed for system hardening. We were planning on using Fortify, but the Fortify software that we were using did not support C#, so we were unable to use that as well. However, C# does not allow unsafe code by default, so any potential security flaws would have been flagged by the compiler.

During the creation of our sequence diagram, we tried to find out where the game might have some vulnerabilities and made a list of potential abuse cases. Then, we were able to address some of the vulnerabilities through programming (our anti-cheat system). However, we are forced to accept some of the potential vulnerabilities such as an attacker redistributing our game if we do not post it on a reputable game distribution platform such as Steam. An example of an abuse case that was fixed by us is that when downloading a Unity game, it ships with all the assets the game requires in their unencrypted format. We were able to encrypt our high scores file, "scores.json," and use a key to encrypt/decrypt the save files.

```C/C++
private string EncryptString( string plainText, string key ) {

      //Advanced Encryption Standard
      //creating new instance of aes class and aesAlg variable
      using Aes aesAlg = Aes.Create( );
      //key is converting key string to byte array using UTF-8
      aesAlg.Key = Encoding.UTF8.GetBytes( key.PadRight( 32 ) );
      aesAlg.IV = new byte[ 16 ];

      // Create an encryptor to perform the stream transform.
      ICryptoTransform encryptor = aesAlg.CreateEncryptor( aesAlg.Key, aesAlg.IV );

      //Using MS CryptoStream to encrypt the string with Key and IV defined.
      using MemoryStream msEncrypt = new( );
      using ( CryptoStream csEncrypt = new( msEncrypt, encryptor,
CryptoStreamMode.Write ) ) {
        using StreamWriter swEncrypt = new( csEncrypt );
        // Write all data to the stream.
        swEncrypt.Write( plainText );
```

```
        }

        return Convert.ToBase64String( msEncrypt.ToArray( ) );

    }
```

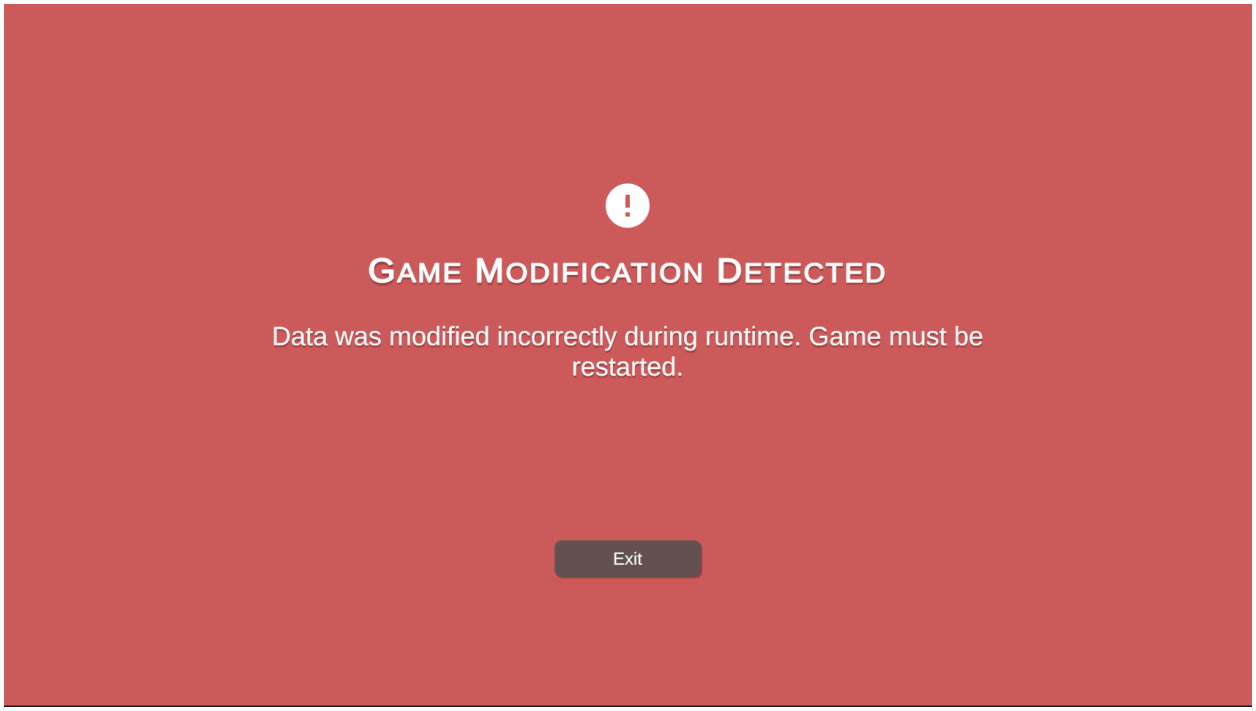
Above is our file encryption function which takes advantage of AES (Advanced Encryption Standard). Using AES allows for the option of 128, 196, or 256-bit encryption depending on how strong the developer wants the encryption to be. There is a tradeoff when choosing an encryption strength due to the time needed to encrypt/decrypt the high scores file.

```
C/C++
private string DecryptString( string cipherText, string key ) {

        using Aes aesAlg = Aes.Create( );
        aesAlg.Key = Encoding.UTF8.GetBytes( key.PadRight( 32 ) );
        aesAlg.IV = new byte[ 16 ];

        // Create a decryptor to perform the stream transform.
        ICryptoTransform decryptor = aesAlg.CreateDecryptor( aesAlg.Key, aesAlg.IV );

        //Using MS CryptoStream to decrypt the data
        using MemoryStream msDecrypt = new( Convert.FromBase64String( cipherText ) );
        using CryptoStream csDecrypt = new( msDecrypt, decryptor,
CryptoStreamMode.Read );
        using StreamReader srDecrypt = new( csDecrypt );
        //read decrypted bytes and return to string format to be returned
        return srDecrypt.ReadToEnd( );

    }
```

Above is our file decryption function which utilizes the aesAlg.Key and aesAlg.IV to decrypt the encrypted json. The aesAlg.Key is the key unknown to the user buried within the system, while the aesAlg.IV is an IV key that is available to find if the user wants to find it. Since during the encryption and decryption the functions use the same “key,” AES (Advanced Encryption Standard) is a symmetric algorithm. Unfortunately, our high score encryption algorithm features the encryption key in plaintext currently since we do not have a server to handle encryption away from the user. If we were able to use a server, we would handle high score data (as well as storing encryption keys) on the server to avoid any potential influence from the user.

Normally when a player finishes a game, the game will create the high score file, if it does not exist, and store the player’s time and score. However, our game will check the score before it is

saved to the file, so it is much harder for attackers to influence their high scores. Utilizing validation checks will allow us to detect when someone is cheating. With the use of simple mathematical equations and comparisons, we are able to determine if a score is unattainable with the provided time and spawn rate of the asteroids to verify the score without having it checked by a server.



For example if a player ends the game by surviving for 30 seconds and destroying 3000 asteroids, the provided values do not match the game's logic. The high score will be discarded and the player will be sent to the anti-cheat scene shown above.

6. Application Review and Vulnerability Assessment

We chose to develop our game securely on the Unity game engine platform. Unity is a cross platform game engine that uses C# for game development. In our initial inspection of Unity, it was developed to allow developers to build projects with security in mind. In order to implement security features such as file encryption, Unity provides a cryptography library implemented as Unity.Security.Cryptography. The aforementioned library is the one used to secure our game.

Unity.Security.Cryptography

Our game takes advantage of the encryption feature by implementing AES (Advanced Encryption Standard). AES is a symmetric encryption algorithm and allows us to implement file encryption in our game by having a secret key and public initialization vector (IV). For our game, we secured the high scores file using this encryption algorithm.

The targeted file is a JSON file stored at User/**UserName**/AppData/LocalLow/Team Bracket/scores.json. This file contains a list of scores including details of each score such as final score, time survived, and the number of asteroids destroyed. By being able to control when it's encrypted we ensured the integrity of the scores when the scoreboard is shown to the player. Although there is an issue with the encryption key stored in the code, this can be mitigated in the future by utilizing a server. However, since our high scores are stored locally and only show local scores to the player, attackers are discouraged from modifying the file since there would be nothing to gain.

7. Penetration Testing

Pen-Testing Overview

Throughout the penetration testing process, we were able to manipulate the game in various ways without directly accessing the code. This is an important discovery since it shows that an attacker can influence a Unity game by utilizing free and popular cheating tools. For our game, the biggest threat is changing the scoreboard, so we wanted to test and ensure that the integrity of the high scores is maintained.

If we decided to develop an online leaderboard that showed all player scores, we would want to make sure that this scoreboard could be trusted to show accurate data. Although minor, influencing a scoreboard can diminish the reputation of a game since it discourages innocent players from trying to get a high score. If all of the top scores are fake, then a player would not be able to reach the top of the list of high scores legitimately.

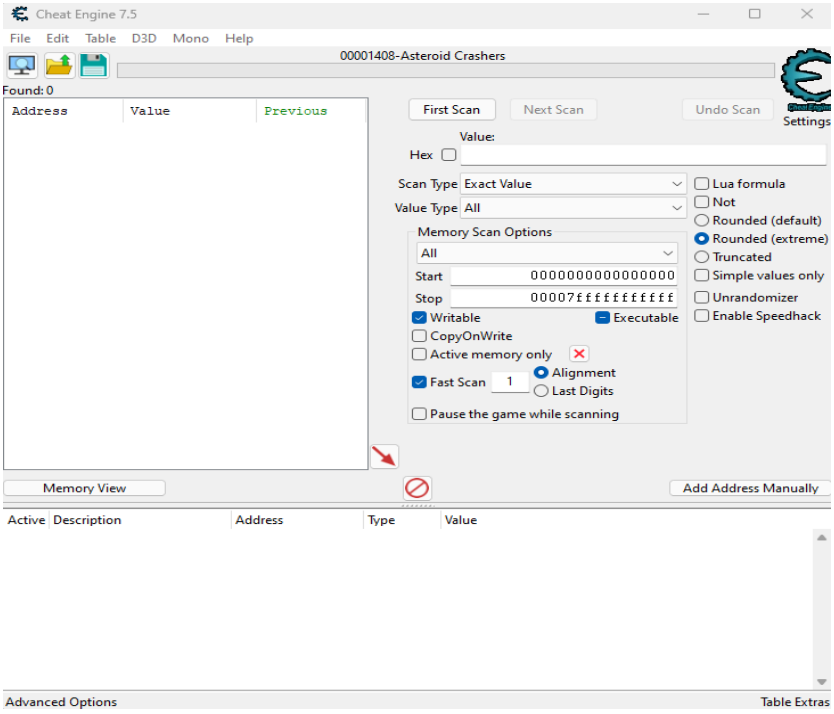
Another use case for anti-cheat in single player games is that mobile games primarily use microtransaction models to earn revenue. Microtransactions are purchases made by players in-game where real money is traded for in-game currency that can be used to buy items to enhance the gameplay experience. If an attacker was able to modify the amount of in-game currency that they have without paying for the microtransaction, businesses could lose a lot of money. This would be a much more severe threat compared to our plan to secure high scores, but we can demonstrate throughout this project that it is possible to secure Unity games against this threat.

In order to test our anti-cheat system, we used a variety of tools such as ILspy, dnSpy, IL2CPPDumper and MelonLoader. However, the most popular method that most attackers use to break Unity games is Cheat Engine. These are all free programs that are easily downloadable which allow for manipulating in-game variables in real time.

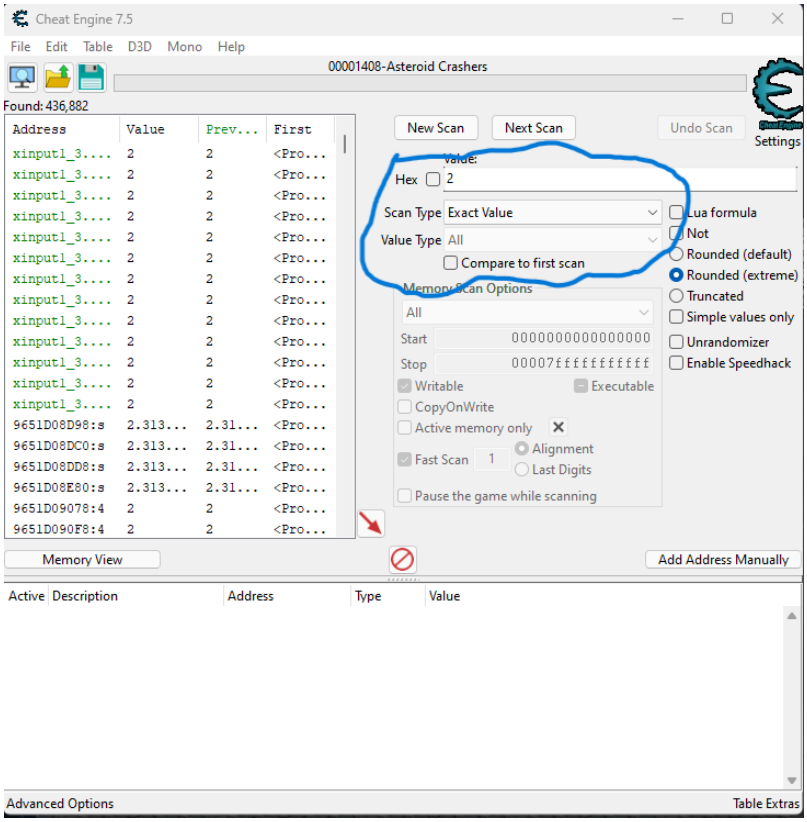
7.1 Cheat Engine

Cheat Engine is a piece of software that runs in the background that reads memory addresses and allows a user to modify the values stored in those addresses. The software works by using a memory scanner to analyze different addresses utilized by an application. The scanner returns a list of addresses and their values by searching for addresses that have updated their values. Modifying the values stored in these addresses allows the user to change how values are shown in-game. It is accomplished by running multiple scans of the program. These scans are able to detect exactly which values are being changed in the application. After finding these addresses, the user can change the value inside that address and update it in real time.

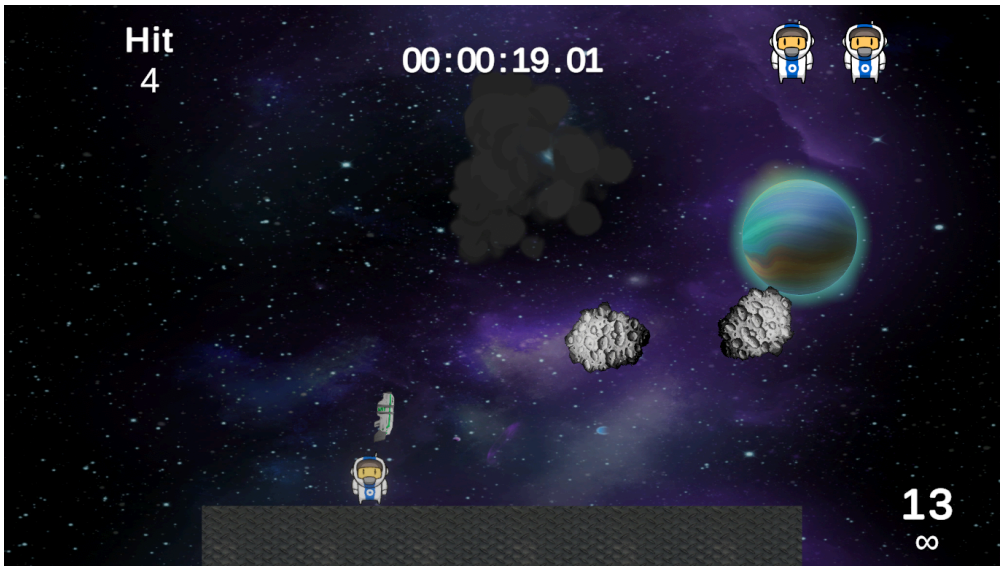
Cheat Engine Use:
This is a screenshot of Cheat Engine running. In order to modify software using this tool, an attacker must select the process. After selecting a process, it will return a list of addresses found within the program. In the value slot, the attacker can modify the current value and change it as desired. In the picture below we will be selecting the asteroids destroyed value in the top left. We selected this because this value directly affects the score in the leaderboard.



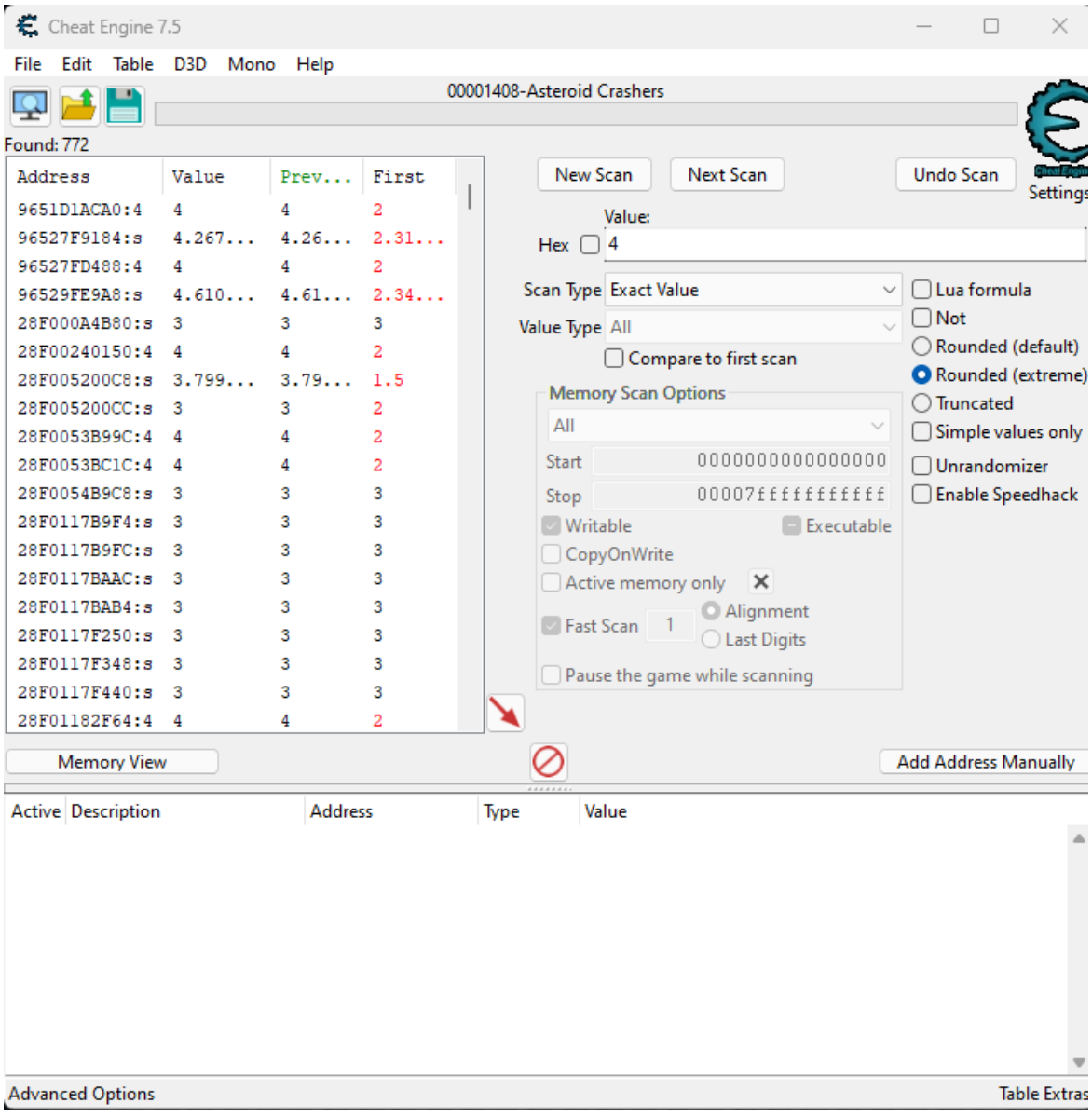
In this case, we have already destroyed two asteroids, so we enter "2" into the value slot and scan. This scan will look through the memory values and scan for every memory address where the value is two.



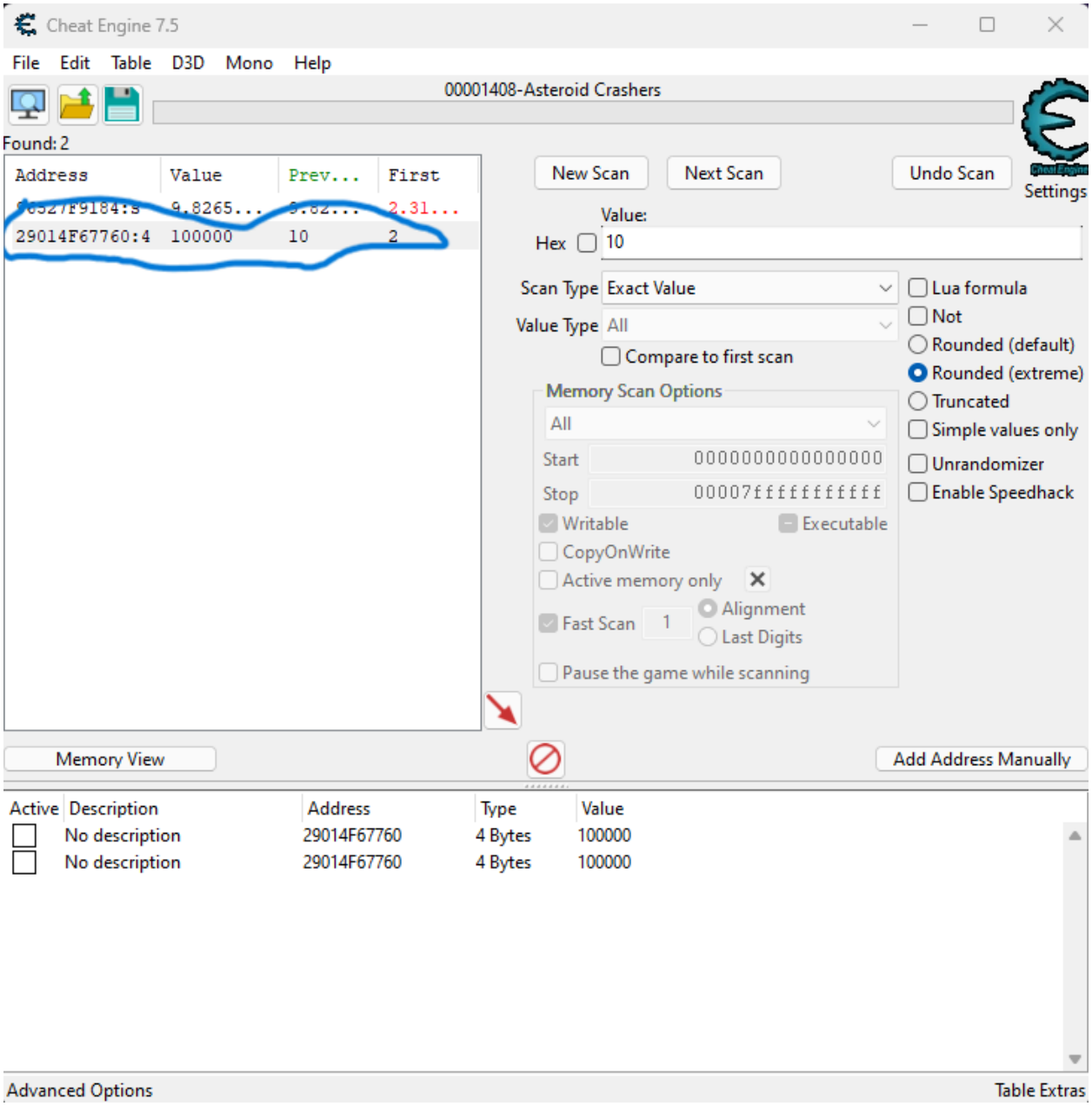
As shown above, there are 436,882 results found where the value found is 2. This means that we need to play the game and update the asteroidsDestroyed value again.



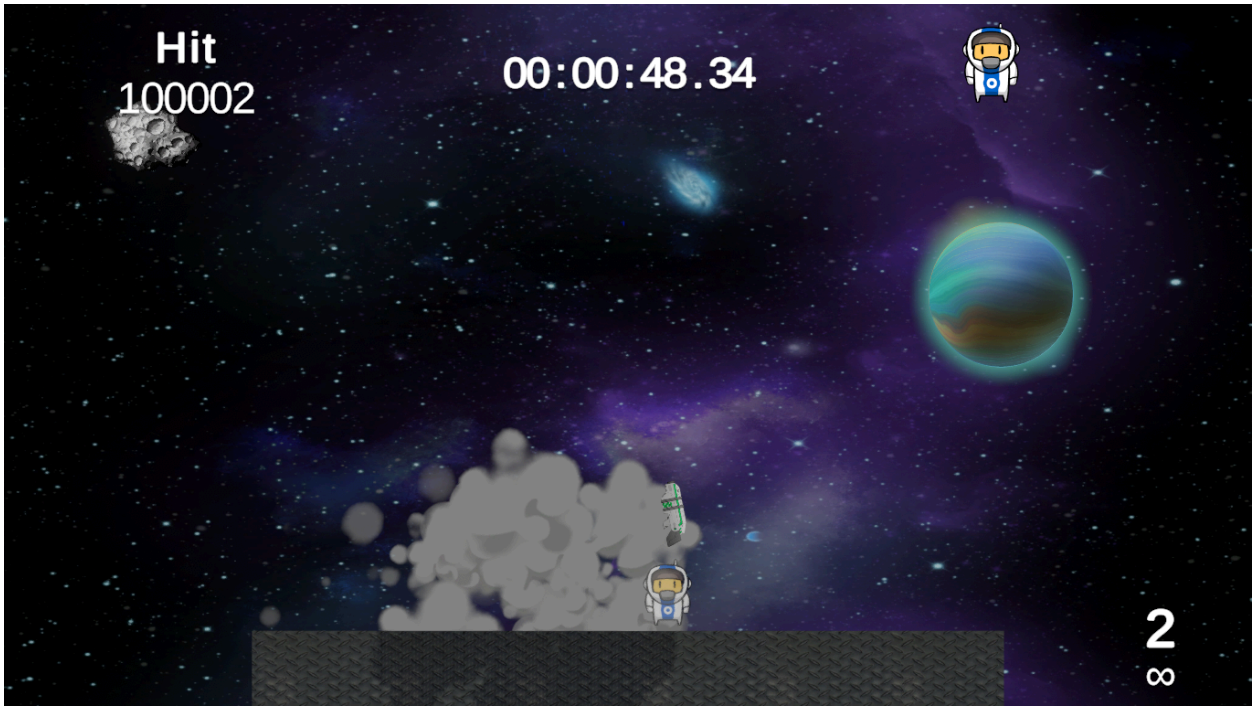
For this test, we hit four asteroids. We ran the Cheat Engine search again for the value “4” this time.



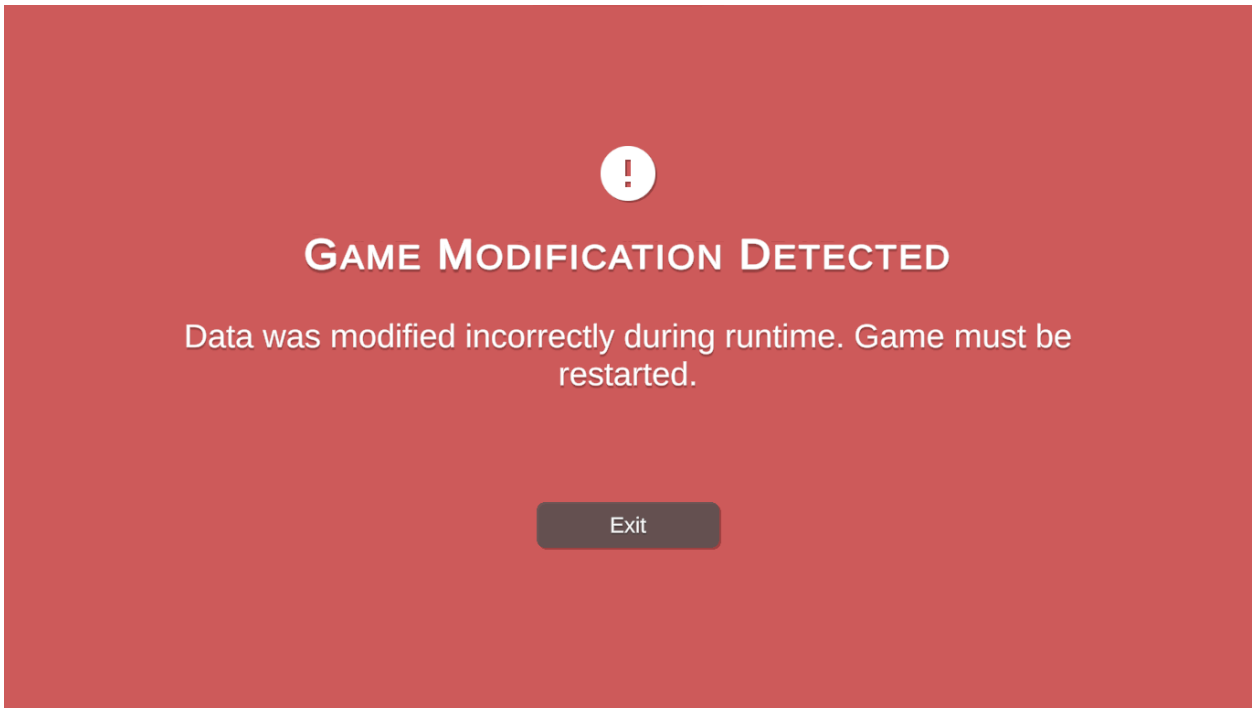
As we can see above, there are considerably less results found for the value of “4.” This is because the program searches for the value “4” that was updated from the value “2” since the last scan. In order to find the exact value, we kept updating the number in the game and inputting it into Cheat Engine to scan again.



In this scenario, we had to run the scan an additional two more times in order to limit the score to two options. Attackers can change the value by right clicking on the listing in the results and clicking “Edit.” They can then edit any value and it will be modified in the game.

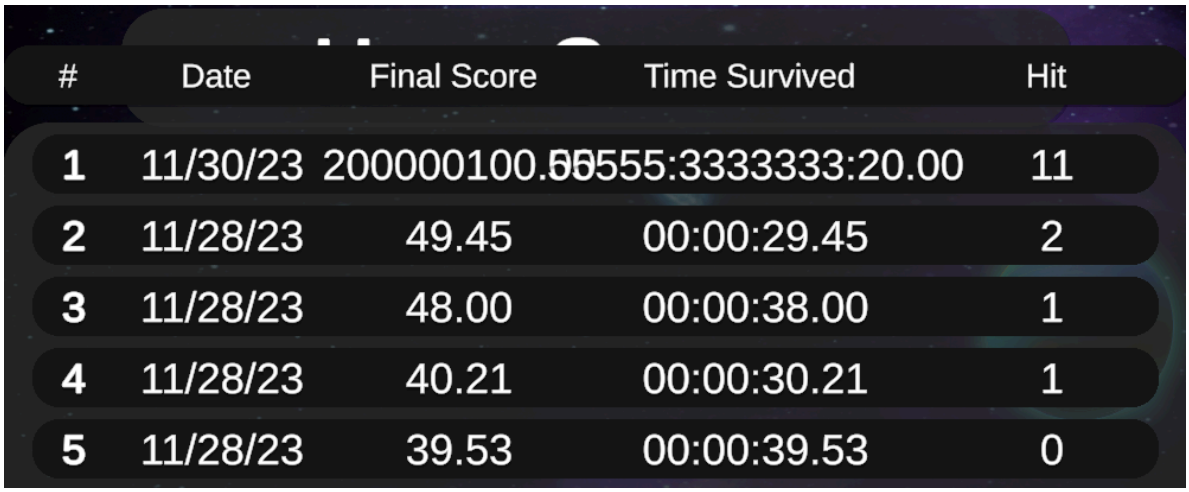


Here we updated the number to “100002.” It can be seen in the top left of this screenshot above. This will directly affect the leaderboard in the end screen. However, this was tested on an earlier version of the game and since then the development team has patched this method of changing the leaderboard.



Now the program runs and detects if the score is higher than the amount of asteroids that have been spawned. If so the game will output the screen above and no longer save any progress. However, we have found a simple workaround that allows us to instead change the time which also changes the high score amount.

As seen above, because the development team patched the asteroidsDestroyed value, we decided to attack the timeSurvived variable to increase the high score. We used the same method as earlier, and inputted the value as “100,000,000,000”. However, it seems that the program only holds a value of “55555” days. This gives us a high score which would take the player years to obtain. This also can be seen in the leaderboard below. This exploit was patched by the development team after being discovered by the penetration team.



#	Date	Final Score	Time Survived	Hit
1	11/30/23	200000100.00	55555:3333333:20.00	11
2	11/28/23	49.45	00:00:29.45	2
3	11/28/23	48.00	00:00:38.00	1
4	11/28/23	40.21	00:00:30.21	1
5	11/28/23	39.53	00:00:39.53	0

Overall, Cheat Engine is an effective way to change values that are being displayed on the screen. However, if developers add validation checks to those values, which is simple, it can lead to this method being ineffective. It is also much more difficult to change values that are not being shown directly onto the screen.

An example of this in our program is our lives system. It shows 3 lives in the top right, but depending on how the developers implemented the system, the value could be random and the developer could simply subtract 1 from that original value on the player's death, up to 3 times.

Example: The lives could start at 10, and once the lives go down to 7, the game would end.

This means if an attacker was looking for the total number of lives value in cheat software, they would not consider searching for a number such as 10, they would be more likely to try and search for the number 3. To check values such as lives, attackers would need to directly access the code. This can be done using the other methods that we have tested, however they are more difficult to use as shown below.

ILSPY

A popular way of hacking a Unity application is to use the software tool, **ILSPY**. ILSPY is an open-source .NET assembly browser and decompiler invented by Daniel Grunwald and David Srbecky. A decompiler is a program that translates an executable file into high-level source code (a programming language readable and writable by humans). The IL in ILSPY stands for Intermediate Language. Intermediate Language files include *.dll* files and *.exe* files. High-Level source code is first translated into an IL file, which is then translated into machine code (Binary Code) that the computer can read. ILSPY is therefore used to translate a file back into its original source code. It takes a *.dll* file and translates it back into a *C#* file. It's typically used on videogame files, as the source code for the games is often unavailable.

ILSPY is written in the .NET Framework. The .NET Framework is "a software development framework for building and running applications on Windows." It was invented by Microsoft back in 2002 and has seen steady improvement since. There are 2 major components in the architecture of the .NET Framework:

1. The *Common Language Runtime (CLR)* - an execution engine in charge of handling running applications.
2. The *Class Library* - provides a set of APIs and types for common functionality.

This library includes APIs for reading and writing files. The .NET framework provides language interoperability. This means that each programming language can use code from other programming languages. It provides language interoperability for multiple languages. Using the .NET Framework, ILSPY is able to decompile *.dll* files into C# files. In order for ILSPY to work a user would need the "Assembly-CSharp.dll" file. Therefore, it is not possible to penetrate our

game with ILSPY since it compiled with Intermediate Language to C++ (IL2CPP) instead of Mono C#.

dnSpy

Another potential way of hacking Unity applications is the software **dnSpy**. DnSpy is a .NET debugger and assembly tool, originally developed by an anonymous person. On December 21, 2020, the project was discontinued. DnSpyEx is a continuation of the project developed by Elektrokill. The main difference between dnSpy and dnSpy is that the new version supports .NET 8.0 (the latest version). DnSpy allows you to view and edit the game's files by using the .dll file from the game. While the ability to edit makes it superior to ILSPY, dnSpy only works with Mono .dll files. Our game uses IL2CPP .dll files so dnSpy by itself will not work.

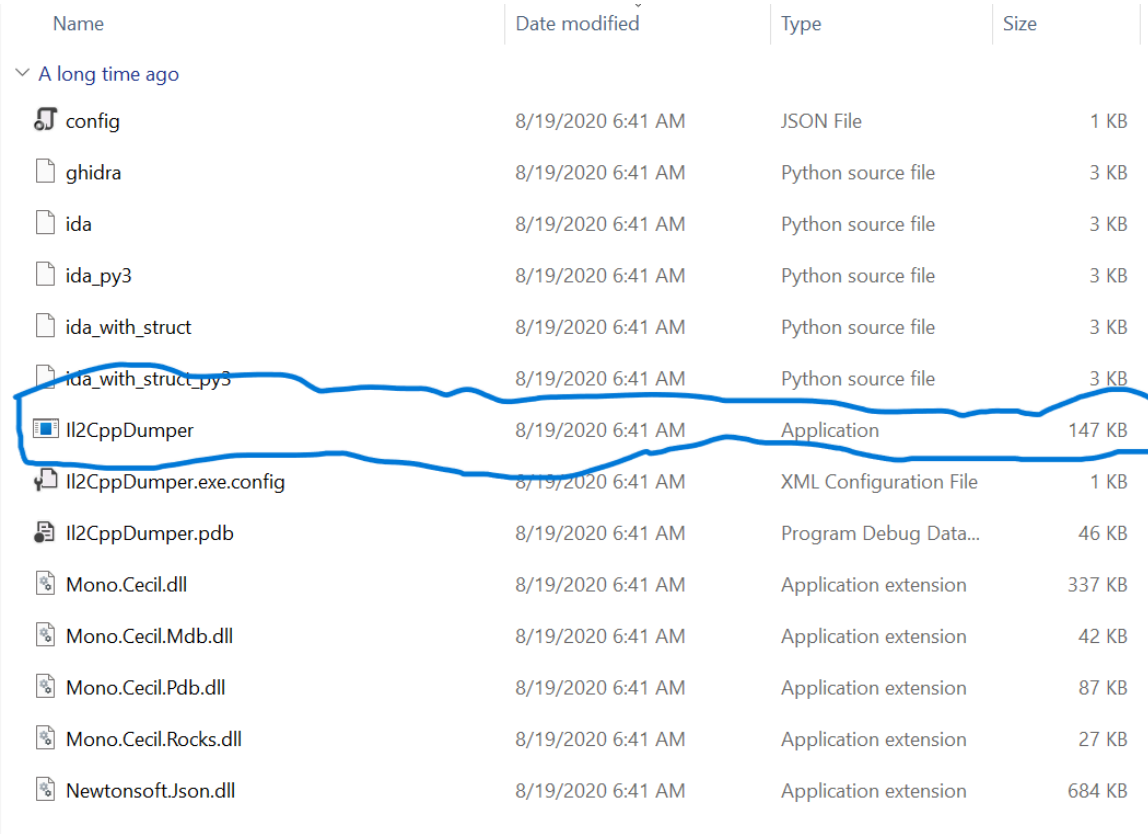
IL2CPPDumper

Introduction:

Another potential way of hacking a Unity application is to use a piece of software named **IL2CPPDumper** developed by Perfare. IL2CPPDumper is written in C# and specifically designed to penetrate IL2CPP Unity applications. It allows you to translate IL2CPP .dll files into Mono .dll files. Combining this with dnSpy allows you to reverse engineer an entire game's code.

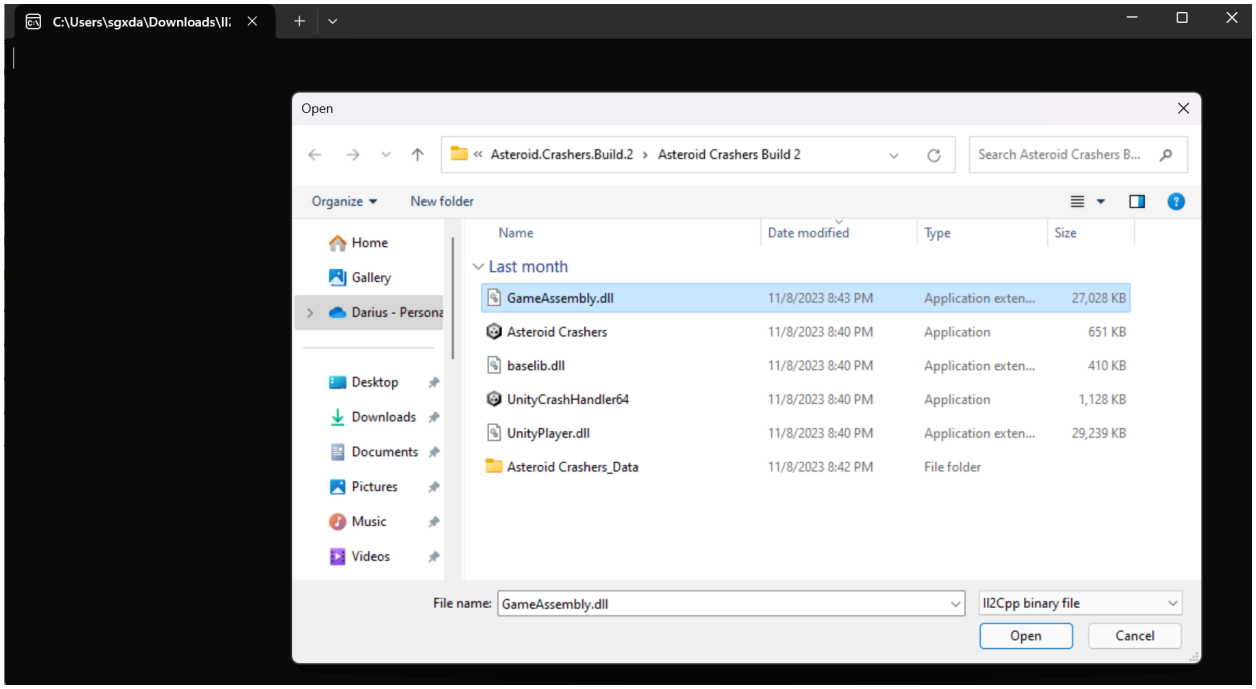
Using:

First: Download the IL2CPPDumper software from its GitHub repository and run it.

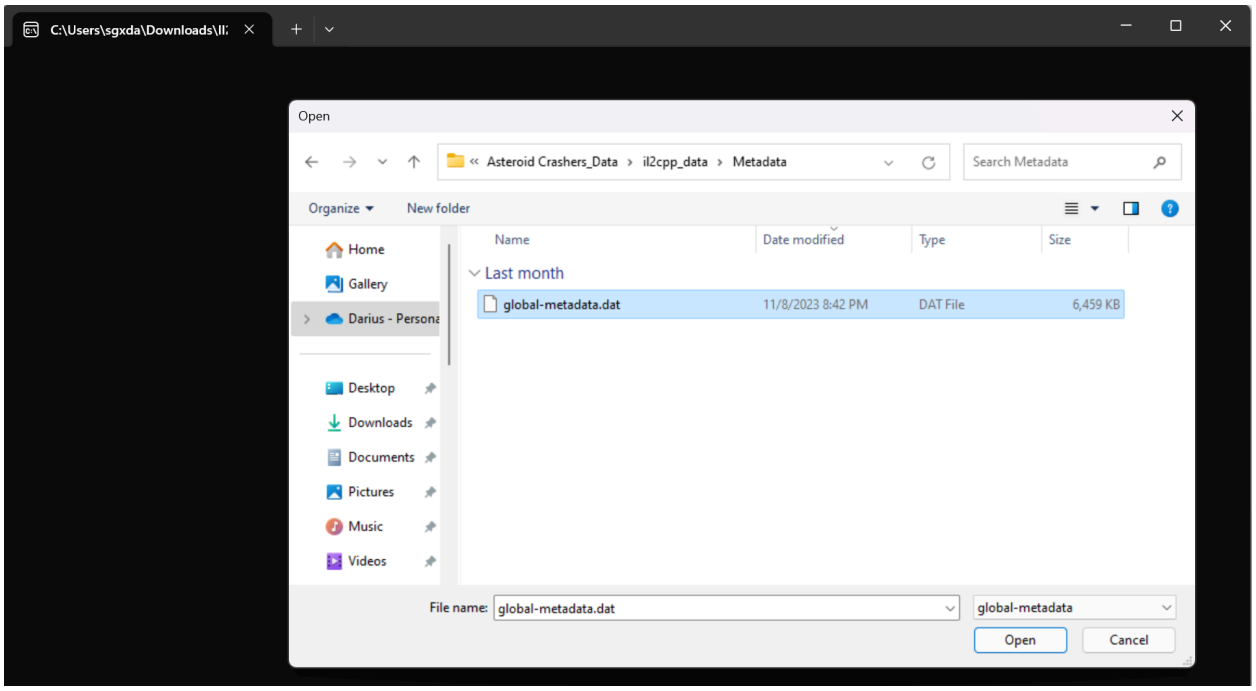


Name	Date modified	Type	Size
A long time ago			
config	8/19/2020 6:41 AM	JSON File	1 KB
ghidra	8/19/2020 6:41 AM	Python source file	3 KB
ida	8/19/2020 6:41 AM	Python source file	3 KB
ida_py3	8/19/2020 6:41 AM	Python source file	3 KB
ida_with_struct	8/19/2020 6:41 AM	Python source file	3 KB
ida_with_struct_py3	8/19/2020 6:41 AM	Python source file	3 KB
Il2CppDumper	8/19/2020 6:41 AM	Application	147 KB
Il2CppDumper.exe.config	8/19/2020 6:41 AM	XML Configuration File	1 KB
Il2CppDumper.pdb	8/19/2020 6:41 AM	Program Debug Data...	46 KB
Mono.Cecil.dll	8/19/2020 6:41 AM	Application extension	337 KB
Mono.Cecil.Mdb.dll	8/19/2020 6:41 AM	Application extension	42 KB
Mono.Cecil.Pdb.dll	8/19/2020 6:41 AM	Application extension	87 KB
Mono.Cecil.Rocks.dll	8/19/2020 6:41 AM	Application extension	27 KB
Newtonsoft.Json.dll	8/19/2020 6:41 AM	Application extension	684 KB

Second: Navigate to the game's folder and find the file called "GameAssembly.dll." Double click on it and then open it with the software.



Third: The File Explorer window will briefly flash. Now the software is looking for the game's metadata. Navigate to the game's metadata. Click on it, and then click open.



Fourth: Normally, the process would continue. However, due to changes in the way Unity compiles code in newer versions of IL2CPP, IL2CPPDumper is unable to work with the provided files. Therefore, IL2CPPDumper is unable to continue the process and must stop here.

Melonloader & Unity Explorer

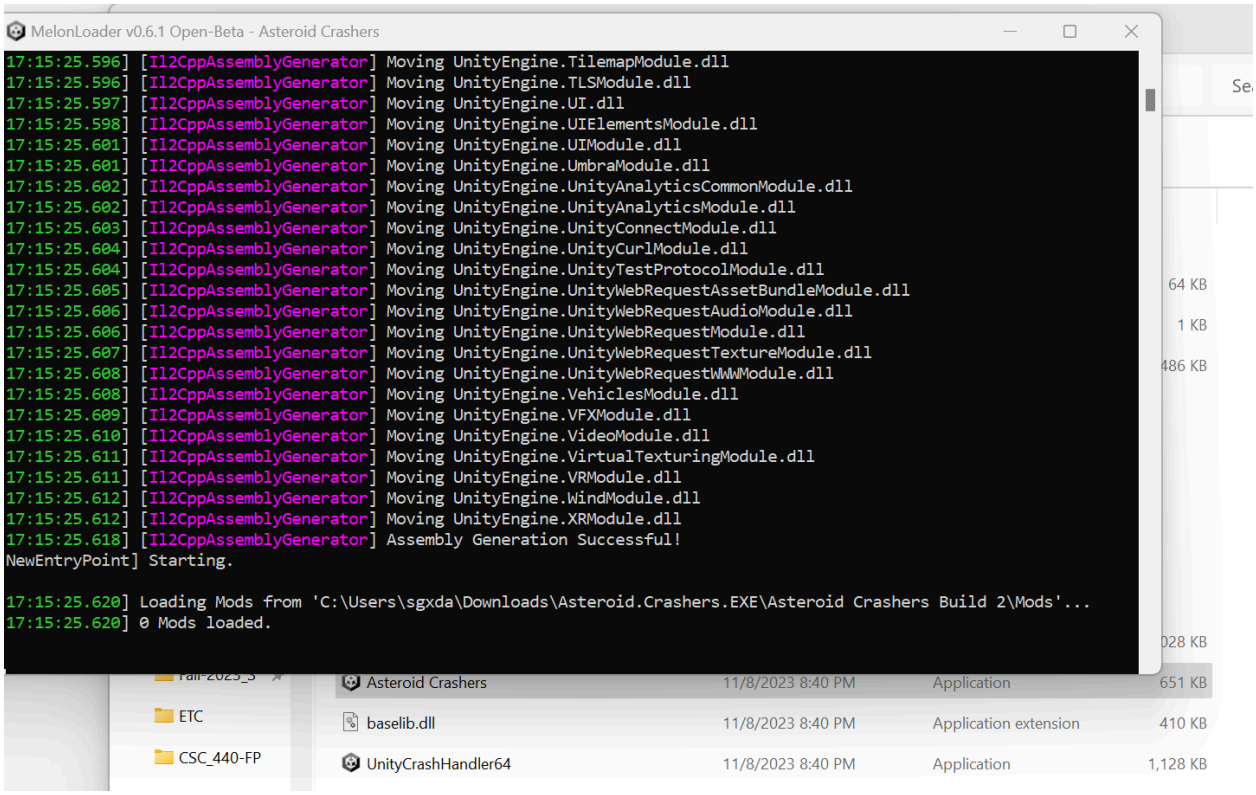
Melonloader is a universal mod loader for Unity games invented by LavaGroup and written in C#. A “mod” is any additional content added to a piece of software that was not part of the developer’s original intentions. This could be a new weapon, new story, new map, new game mode, etc. Often, players will use a mod loader to add mods to a game. Without the use of a mod loader, adding mods to a game can be difficult. It requires knowledge of the game’s folders and extreme patience. Mods can often conflict with each other. If a certain mod loads before another mod, the second mod file will be corrupted and unable to load (potentially breaking the game). This happens because often mods depend on other mods to work. For example, one mod adds a new framework for a game. Then another person (who is not the creator of the framework mod), makes a mod using that framework. So if the mod that uses the framework loads before the framework mod itself, an error will occur. A mod loader will take care of all these problems for us. It will add mods into the correct folder and load mods into the correct order.

We will be using Melonloader to load a mod called UnityExplorer. UnityExplorer is an in-game user interface (UI) for exploring, debugging, and modifying Unity games invented by Sinai. It is built using Unity itself and therefore written in C#. Unity Explorer will allow us to both see and edit the game files. Using these 2 pieces of software, we can edit the code for the game without requiring any access to the game files.

How to Use:

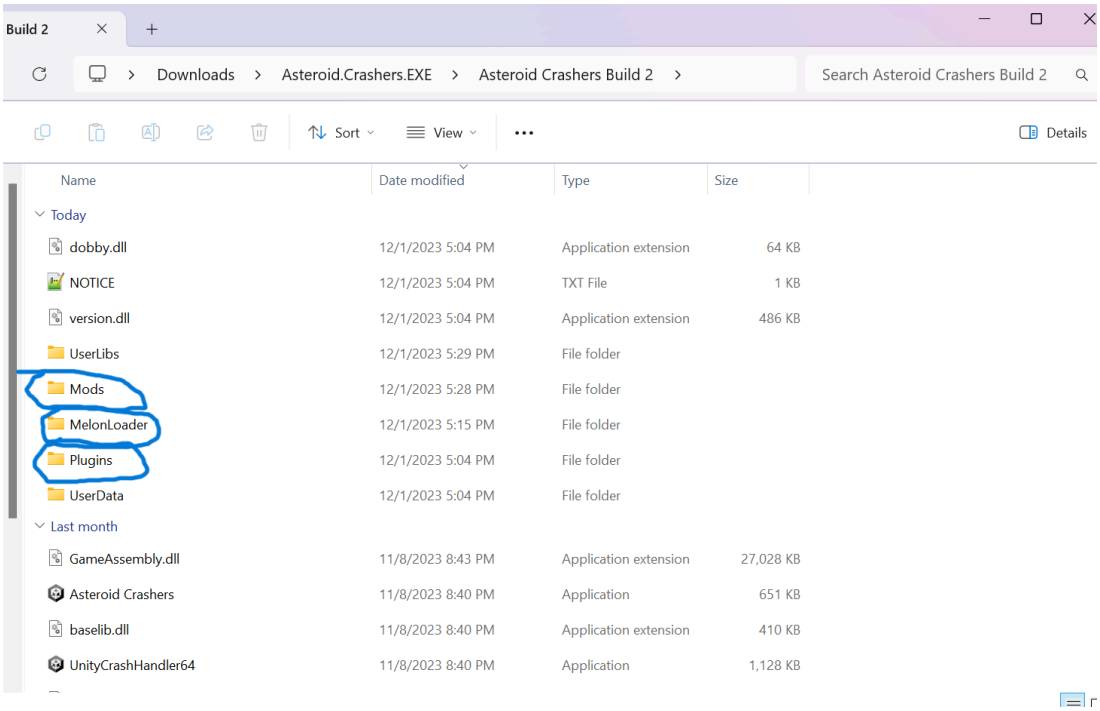
First: Download the Melonloader software from its GitHub repository. The software will be downloaded as an installer. Upon clicking on the installer, a menu will appear. In this menu, click on the “Select” option to target a Unity game. After selecting the game, click “Install.”

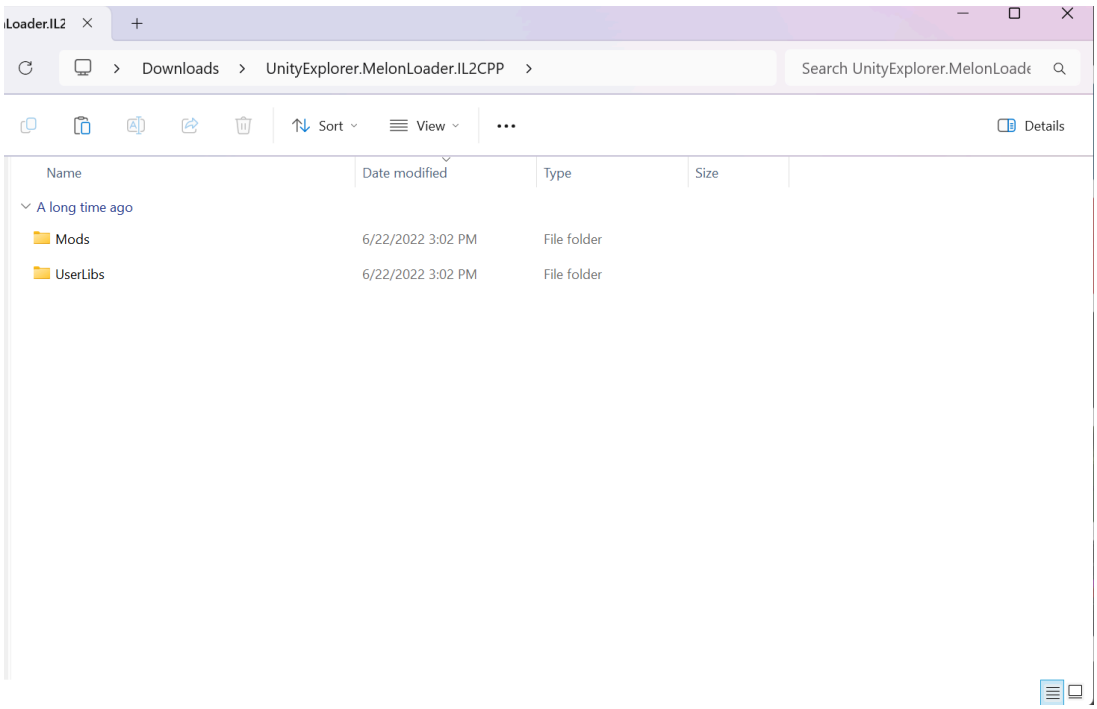
Second: The user must now run the game. A shell window will appear (if you run the software regularly without MelonLoader, this would not appear) that will show the installation of MelonLoader into the software.



Third: Download the UnityExplorer software from its GitHub repository. It is important to download the right version due to the compiler differences between Mono and IL2CPP. Since

our Unity application is compiled in IL2CPP, we will download the IL2CPP version of the UnityExplorer mod. The names of MelonLoader's installed directories are: "Mods" and "Plugins." Move the DLL files from the UnityExplorer folder into their respective folders in the game folder. Finally, run the game.





Fourth: Usually, the process would continue. However, due to changes in the compilation process, the software was unable to process the data. Therefore, we are unable to continue the process and must stop here.

Uploading onto a Videogame Distribution Service

We downloaded this game directly from our GitHub repository. By doing so, our game application did not need to go through testing and regulation checking. Many popular game distribution platforms that sell video games and other content, such as Steam and the Microsoft Store, require certain files to be implemented into the game's folder structure. Some platforms require more changes to the file structure than others, so this might cause some of the files that we were not able to access to become visible. This might allow for programs like MelonLoader to function, which could allow for more tampering. However, for the standalone game that we have developed, it is not an issue currently.

Conclusion:

The game we developed, *Asteroid Crashers*, is a secure and safe Unity game. Consequently, the only achievement that an attacker can gain by modifying our game is experience attacking a game. *Asteroid Crashers* is a locally run game that does not contain any monetization systems. All created files are stored locally on the user's PC and no data is sent over the network to a central server. Therefore, any hacking that a potential attacker could inflict is extremely low risk, due to our methods to secure the game, and low threat level, due to the lack of meaningful gains by attacking our game systems.

8. Conclusions and Lessons Learned

Advanced Encryption Standard (AES)

In this project, we learned how to encrypt and decrypt files using AES (Advanced Encryption Standard) encryption which utilizes the same key to encrypt and decrypt. Since it uses the same key, it's a symmetric encryption, the key is called the “secret” and is kept secret from third parties. We are using this encryption algorithm to encrypt the high scores, but we are unable to hide the private cryptographic key without the use of dedicated servers. Due to this, the key is hardcoded, but most users would be discouraged from modifying the scores file considering the effort required.

Intermediate Language to C++ Compiler

Using IL2CPP in this project has helped with obscuring the code from potential attackers. Initially during development, we thought that the IL2CPP compiled build was 700 MB compared to the Mono build’s size of 100 MB. However, we discovered that the IL2CPP build included a folder titled: “Asteroid Crashers_BackUpThisFolder_ButDontShipItWithYourGame” that included all of the converted C# to C++ code.

After removing this folder, the game size was reduced to a more reasonable size of 130 MB. If we had included this in the compressed game folder, hackers would have had access to the C++ source code directly. Therefore, it is important for developers to understand the file structure that comes with switching compilers and only include files in release builds that are absolutely necessary for the game to function properly.

Anti-Cheat

Creating an anti-cheat for the game required a lot of contemplation in order to understand potential attack vectors that attackers could use. By focusing on securing the high scores aspect of the game, we were able to design an anti-cheat system that would protect that section of the game while other sections of the game were left unprotected due to time constraints.

Although we worked to make sure that the number of asteroids destroyed was not modified, we overlooked securing the “time survived” variable. After discovering this exploit, we promptly patched the issue. Therefore, the anti-cheat system created for the game can be expanded in future versions to include other aspects of the game such as the lives, weapon, and movement systems. In the development of this project, we learned how to create a viable anti-cheat that discourages a majority of potential attackers.

Final Words

Secure software engineering is a continuous process that requires all developers to consider how their code can be used. Although we were able to add anti-cheat measures to our game to prevent attackers from influencing the high score directly, a determined and knowledgeable person could eventually find a way to influence the score determination process as shown in our penetration

testing. We were able to mostly secure the high score process to ensure that high scores are validated and that memory manipulation of the high score variables causes the game to end.

However, attackers could influence other aspects of the game if they were determined enough as demonstrated in our penetration testing. Without server validation, securing a local game is difficult since the attacker already has all of the resources that they need by downloading the game. Applying secure software principles requires dedication from developers in order to continuously improve their software to compete against committed attackers. Although it is not possible to completely thwart attackers, developers can design and develop their programs to be written as securely as possible using similar techniques that were used in this project.

9. References

1. “Understanding the Difference between DevOps and DevSecOps.” JFrog, 9 Mar. 2023, www.jfrog.com/devops-tools/article/difference-between-devops-and-devsecops/. Accessed 1 Dec. 2023.
2. Cheat Engine Features, Advantages, Disadvantages, Uses | Science Online. Soffar, Heba. 17 Jan.2016, www.online-sciences.com/games/cheat-engine-features-advantages-and-disadvantages/.Accessed 1 Dec. 2023.“Diagrams for Confluence and Jira.” Draw.Io, 7 Aug. 2023, www.drawio-app.com/.
3. Alan Zucconi. “A Practical Tutorial to Hack & Protect Unity Games.” Alan Zucconi, 1 Apr. 2020,www.alanzucconi.com/2015/09/02/a-practical-tutorial-to-hack-and
4. -protect-unity-games/.
5. “How to Hack Any Il2cpp Unity Game Easily.” UnKnoWnCheaTs, 15 Nov. 2021, www.unknowncheats.me/forum/unity/477927-hack-il2cpp-unity-game-easily.html.
6. “How to Hack Any Il2cpp Unity Game Easily.” *UnKnoWnCheaTs*, 15 Nov. 2021, www.unknowncheats.me/forum/unity/477927-hack-il2cpp-unity-game-easily.html.
7. “Hacking and Reverse Engineering Il2cpp Games with Ghidra.” Tomorrowisnew, 7 Feb. 2022, https://tomorrowisnew.com/posts/Hacking-and-reverse-engineering-il2cpp-games-with-ghidra/.
8. “Hacking Unity Games”, www.hypn.za.net/blog/2020/04/11/hacking-unity-games.Accessed 3 Dec. 2023.
9. Khan, Muhammad Faizan. “Difference between unity scripting backend IL2CPP and Mono2x” *Game Development Stack Exchange*, 16 Apr. 2019, www.gamedev.stackexchange.com/questions/140493/difference-between-unity-scripting-backend-il2cpp-and-mono2x.
10. “DnSpy/DnSpy.” *GitHub*, 13 May 2021, https://github.com/dnSpy/dnSpy.
11. ElektroKill. “DnSpyEx.” *GitHub*, 19 Aug. 2022, https://github.com/dnSpyEx/dnSpy.
12. Grunwald, Daniel, and David Srbecký. “ILSpy.” *GitHub*, 3 June 2022, https://github.com/icsharpcode/ILSpy.
13. LavaGang. “LavaGang/MelonLoader.” *GitHub*, 18 April. 2020, https://github.com/LavaGang/MelonLoader.
14. Lebreton, Sebastien. “Reflexil.” *GitHub*, 9 Aug. 2013, https://github.com/sailro/Reflexil.
15. Perfare. “Release Il2CppDumper V6.4.12 · Perfare/Il2CppDumper.” *GitHub*, 31 Dec. 2016, https://github.com/Perfare/Il2CppDumper/releases/tag/v6.4.12.
16. Sinai. “Releases.” *GitHub*, 7 Aug. 2020, https://github.com/sinai-dev/UnityExplorer.
17. “What Is .NET Framework? A Software Development Framework.” *Microsoft*, https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework.
18. Technologies, Unity. “Order of Execution for Event Functions.” *Unity*, docs.unity3d.com/Manual/ExecutionOrder.html.
19. Staff, Ars. “Unity at 10: For Better—or Worse—Game Development Has Never Been Easier.” Ars Technica, 27 Sept. 2016, www.arstechnica.com/gaming/2016/09/unity-at-10-

for-better-or-worse-game-development-has-never-been-easier/.
20. Dealessandri, Marie. “What Is the Best Game Engine: Is Unity Right for You?” GamesIndustry.biz, 16 Jan. 2020, www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you.
21. , Jeff. “24 Great Games That Use the Unity Game Engine.” TheGamer, 27 Feb. 2020, www.thegamer.com/unity-game-engine-great-games/#starship-troopers-terran-command. Accessed 3 Dec. 2023.
22. Ltd, Arm. “What Is a Gaming Engine?” Arm | The Architecture for the Digital World, www.arm.com/glossary/gaming-engines.
23. Technologies, Unity. “Unity - Manual: IL2CPP Overview.” Docs.unity3d.com, docs.unity3d.com/Manual/IL2CPP.html.
24. Technologies, Unity. “Unity - Manual: Mono Overview.” Docs.unity3d.com, docs.unity3d.com/Manual/Mono.html. Accessed 3 Dec. 2023.
25. Technologies, Unity. “Unity’s Protection Policies for Its Creators | Unity Security.” Unity.com, unity.com/security. Accessed 3 Dec. 2023.
26. Technologies, Unity. “Unity January 2023 Security Update Details.” Unity.com, unity.com/security/jan-2023-01. Accessed 3 Dec. 2023.

Appendix 1

10.1 Scripts & Release Build

Scripts: https://drive.google.com/drive/u/1/folders/1jlKf1AlXJ1RYLNYqhq3f3ky-ehKMnlhn/
Final Release Build (Windows, x86):
https://drive.google.com/file/d/1SKwOvSLh-OySLYHF7cLyvEKtP39of9dU/view?usp=drive_link

Note: Google Docs changes the code blocks to only extend to two indentations. For viewing the original code indentations and formatting, the Google Drive link includes the unaltered source code.

Additionally, the minimal use of third party code is not listed below, but is included in the Scripts folder at the location: Assets/Scripts/Third Party/JMO Assets/. This code was used for camera shake and particle effects and was slightly altered to comply with C# naming conventions.

10.2 Source Code

AI:

Agent.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Agent
 * Script Objective: Emulates the player in the main menu to make the menu stand out
more.
 *
 */

using UnityEngine;
using UnityEngine.AI;


namespace TeamBracket.AI {
	public class Agent : MonoBehaviour {


	private NavMeshAgent agent;
	private bool lookingForDestination = false;

	[SerializeField] private Animator animator;
	[SerializeField] private SpriteRenderer character;
```

```
// Start is called before the first frame update
private void Start( ) {

agent = GetComponent<NavMeshAgent>( );

agent.updateRotation = false;

Invoke( nameof( GoToRandomPosition ), 0.5f );

}

private void GoToRandomPosition( ) {

if ( gameObject.activeInHierarchy ) {

Vector3 newPosition = new( Random.Range( -15, 15 ), 1.5f, 0f );
_ = agent.SetDestination( newPosition );

lookingForDestination = true;

}

}

private void AnimateMovement( ) {

int moveDirection = ( agent.velocity.x > 0 ) ? 1 : -1;

if ( Mathf.Abs( agent.velocity.x ) < 0.1f )
moveDirection = 0;

bool isMoving = moveDirection != 0;
animator.SetInteger( "MoveDirection", moveDirection );
animator.SetBool( "IsMoving", isMoving );

character.flipX = moveDirection == 1;

}

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

AnimateMovement( );
```

```
        if ( lookingForDestination && agent.remainingDistance <
agent.stoppingDistance ) {

        lookingForDestination = false;
        Invoke( nameof( GoToRandomPosition ), Random.Range( 0.1f, 2f ) );

        }

        }
        }

}
```

Game:

BoundsDetection.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Bounds Detection
 * Script Objective: Destroy out-of-bounds objects as well as prevent the player from
falling forever if they fall off the platform.
 *
 */

using UnityEngine;

namespace TeamBracket {
        public class BoundsDetection : MonoBehaviour {

                [SerializeField] private PlayerHealth healthScript;

                private void OnTriggerEnter( Collider other ) {

                        if ( other.gameObject.CompareTag( "Player" ) ) {

                                healthScript.PlayerDied( true );

                        }
                        else {
```

```
                        Destroy( other.gameObject );

                    }

                }

        }

}
```

ContextualNotifications.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Contextual Notifications
 * Script Objective: Shows/hides on screen text based on the current state of the game.
Used to show "Reloading..." and "Respawning.." text to the player.
 *
 */

using UnityEngine;

namespace TeamBracket {
        public class ContextualNotifications : MonoBehaviour {

        [SerializeField] private GameObject animatedTextContainer;
        [SerializeField] private GameObject respawnText;
        [SerializeField] private GameObject reloadText;

        void ToggleVisibility( bool isVisible ) {

        animatedTextContainer.SetActive( isVisible );

        }

        public void ToggleTextVisiblity( string textName, bool isVisible ) {

        if ( ( isVisible && !animatedTextContainer.activeInHierarchy ) || ( !isVisible
&& animatedTextContainer.activeInHierarchy ) ) {

        ToggleVisibility( isVisible );
```

```
        }

        switch ( textName ) {

        case "Respawn":

               respawnText.SetActive( isVisible );
               reloadText.SetActive( false );
               break;

        case "Reload":
               reloadText.SetActive( isVisible );
               break;

        }

        }

        // Start is called before the first frame update
        void Start( ) {

        ToggleVisibility( false );

        }

        }

}
```

GameManager.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Game Manager
 * Script Objective: Handles game conditions and transitions the player to the
high score scene/menu when the game is over.
 *
```

```
 */

using TeamBracket.HighScores;
using TeamBracket.WeaponSystem;
using TMPro;
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;
using UnityEngine.SceneManagement;

namespace TeamBracket {

	public class GameManager : MonoBehaviour {


	[Header( "Settings" )]
	[SerializeField] private int scoreIncrement;

	[Header( "References" )]
	[SerializeField] private TextMeshProUGUI scoreText;
	[SerializeField] private Stopwatch stopwatch;
	[SerializeField] private GameObject gameOverScreen;
	[SerializeField] private HighScoreManager highScoreManager;

	[SerializeField] private TextMeshProUGUI finalScoreText;

	[SerializeField] private Weapon weaponScript;
	[SerializeField] private Spawner spawnerScript;

	// Grayscale Settings
	[SerializeField] private Volume globalSettings;
	private ColorCurves grayScale;

	private int asteroidsDestroyed;
	private int totalAsteroidsSpawned;

	private int prevAsteroidsDestroyed;

	private float gameRunningTime;
	private bool gameOver = false;

	private readonly TimeStringCreator timeStringCreator = new( );

	private const float ToleranceEpsilon = 0.0001f;
```

```
	private void Start( ) {

		foreach ( VolumeComponent component in
globalSettings.profile.components ) {

		if ( component is ColorCurves ) {

			grayScale = component as ColorCurves;

		}

		}

		asteroidsDestroyed = 0;
		totalAsteroidsSpawned = 0;

	}

	private void Update( ) {

		if ( asteroidsDestroyed > totalAsteroidsSpawned ||
asteroidsDestroyed > prevAsteroidsDestroyed + 3 ) {

		SceneManager.LoadScene( "Error" );

		}

		if ( !gameOver )
		gameRunningTime++;

		if ( stopwatch.GetStopwatchTime( ) > gameRunningTime + 5f ) {

		SceneManager.LoadScene( "Error" );

		}

	}

	public void GoToScene( string sceneName ) {

		ToggleGrayScale( false );
		Time.timeScale = 1.0f;
```

```
            SceneManager.LoadScene( sceneName );

        }

        public void AsteroidsSpawned( int asteroidsSpawned ) {

            totalAsteroidsSpawned += asteroidsSpawned;

        }

        private void UpdateUI( ) {

            scoreText.text = "<b>Hit</b>\n<mspace=25>" + asteroidsDestroyed +
"</mspace>";

        }

        private void ToggleGrayScale( bool isActive ) {

            grayScale.active = isActive;

        }
        public void AddScore( ) {

            // Impossible condition detected, end game.
            if ( Mathf.Abs( asteroidsDestroyed - prevAsteroidsDestroyed ) > 2
) {

            SceneManager.LoadScene( "Error" );

            }

            asteroidsDestroyed += scoreIncrement;
            prevAsteroidsDestroyed = asteroidsDestroyed;

            UpdateUI( );

        }

        private bool ScoreValidation( float timeSurvived ) {

            bool isScoreValid = true;
```

```
            // If the number of destroyed asteroids is greater than the amount
spawned, invalidate the score.
            if ( ( asteroidsDestroyed > 0 ) && ( asteroidsDestroyed >
totalAsteroidsSpawned ) )
            isScoreValid = false;


            // If the time between asteroids destroyed is less than the
weapon's actual fire delay, invalidate the score.
            if ( ( Mathf.Abs( timeSurvived / Mathf.Max( 1, asteroidsDestroyed
) ) - weaponScript.FireDelay ) < ToleranceEpsilon )
            isScoreValid = false;

            // If the total number of asteroids spawned exceeds the amount
possible per interval, invalide the score.
            if ( ( ( totalAsteroidsSpawned / timeSurvived ) *
spawnerScript.SpawnInterval ) > spawnerScript.MaxObjectsPerSpawn )
            isScoreValid = false;

            // If the total time survived is greater than the time the game
has been running plus 5 seconds, invalidate the score.
            if ( timeSurvived > gameRunningTime + 5f )
            isScoreValid = false;

            return isScoreValid;

      }

      public void GameOver( ) {

            ToggleGrayScale( true );

            stopwatch.ToggleStopwatch( false );
            gameOver = true;

            float timeSurvived = stopwatch.GetStopwatchTime( );

            bool isScoreValid = ScoreValidation( timeSurvived );

            if ( !isScoreValid ) {

            SceneManager.LoadScene( "Error" );
            return;
            }
```

```
            // Add Score to List
            highScoreManager.AddScore( asteroidsDestroyed, timeSurvived );

            // Get Overall Score
            HighScore currentScore = highScoreManager.GetScore(
asteroidsDestroyed, timeSurvived );

            // Display Final Score

            finalScoreText.text = string.Format( "<b>Final
Score:</b>\n<size=60>{0:0.00}</size>\n\n<b>Asteroids
Destroyed:</b>\n{1}\n<b>Time Survived:</b>\n{2}",
                                                currentScore.overallScore,
currentScore.asteroidsDestroyed, timeStringCreator.GetNewTimeString(
currentScore.timeSurvived ) );
            // Show Game Over Screen
            gameOverScreen.SetActive( true );

            // Slow Down Time
            Time.timeScale = 0.4f;



      }

      private void OnApplicationQuit( ) {

            ToggleGrayScale( false );

      }

      }

}
```

PlayerController.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
```

```
*
* Name: Player Controller
* Script Objective: Allows for player movement in X/Y directions.
*
*/

using UnityEngine;
namespace TeamBracket.Movement {
	public class PlayerController : MonoBehaviour {
	// Start is called before the first frame update

	[Header( "Movement" )]
	[SerializeField] private float walkSpeed;
	[SerializeField] private float currentSpeed;
	[SerializeField] private float acceleration;
	[SerializeField] private float deceleration;

	[SerializeField] private LayerMask groundLayers;
	[SerializeField] private float playerHeight;

	[Header( "Jumping" )]
	[SerializeField] private float jumpForce;
	[SerializeField] private float jumpCooldown;

	[Header( "References" )]
	[SerializeField] private Rigidbody rb;
	[SerializeField] private Animator anim;
	[SerializeField] private SpriteRenderer character;

	private bool canJump;
	[SerializeField] private bool grounded;

	private int moveDirection;

	private void Start( ) {

	canJump = true;

	}

	private void MovePlayer( ) {

	rb.velocity = new Vector3( currentSpeed, rb.velocity.y, 0 );

	}
```

```
        private void Jump( ) {

        anim.SetTrigger( "Jump" );
        rb.AddForce( new Vector3( 0, jumpForce, 0 ), ForceMode.Impulse );

        Invoke( nameof( ResetJump ), jumpCooldown );

        }

        private void ResetJump( ) {

        canJump = true;

        }

        private void UpdateAnimationState( ) {

        bool isMoving = moveDirection != 0;
        anim.SetInteger( "MoveDirection", moveDirection );
        anim.SetBool( "IsMoving", isMoving );
        anim.SetBool( "Grounded", grounded );

        character.flipX = moveDirection == 1;

        }

        private void GetUserInput( ) {

        float horizontalInput = Input.GetAxis( "Horizontal" );

        moveDirection = 0;

        grounded = Physics.Raycast( transform.position, Vector3.down, ( playerHeight
* 0.5f ) + 0.2f, groundLayers );

        // Move Right
        if ( horizontalInput > 0 ) {

        currentSpeed += acceleration * Time.deltaTime;
        moveDirection = 1;

        }
        // Move Left
        else if ( horizontalInput < 0 ) {

        currentSpeed -= acceleration * Time.deltaTime;
```

```
moveDirection = -1;

}
// Decelerate
else if ( currentSpeed != 0 ) {

int decelerateDirection = ( currentSpeed > 0 ) ? 1 : -1;

currentSpeed -= deceleration * decelerateDirection * Time.deltaTime;

// Stop Player Movement
if ( Mathf.Abs( currentSpeed ) < 0.1f )
       currentSpeed = 0;


}

// Player Presses Jump Key
if ( Input.GetKeyDown( KeyCode.Space ) && canJump && grounded ) {

canJump = false;
Jump( );

}

}

private void ClampSpeed( ) {

// Clamp Walk Speed
if ( currentSpeed > walkSpeed )
currentSpeed = walkSpeed;
else if ( currentSpeed < -walkSpeed )
currentSpeed = -walkSpeed;

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

GetUserInput( );
ClampSpeed( );
UpdateAnimationState( );

}

private void FixedUpdate( ) {
```

```
        MovePlayer( );

        }

        }

}
```

PlayerHealth.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Player Health
 * Script Objective: Handle the player lives system by respawning the player if they die
as well as updating the UI of the player's life count.
 *
 */

using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.SceneManagement;

namespace TeamBracket {
        public class PlayerHealth : MonoBehaviour {
        // Start is called before the first frame update

        [Header( "Player Settings" )]
        [SerializeField] private int totalLives = 3;
        private int livesLeft;

        [SerializeField] private float playerInvincibilityTime;
        private float invincibilityTimeLeft;

        [SerializeField] private Vector3 respawnPoint = Vector3.zero;
        [SerializeField] private float respawnTime = 3f;

        [Header( "References" )]
        [SerializeField] private List<GameObject> UILifeCounter = new( );
```

```
        [SerializeField] private Transform player;
        private Rigidbody playerRb;
        private Animator playerAnimator;

        [SerializeField] private AudioSource playerDeathSound;

        [SerializeField] private GameManager gameManager;

        [SerializeField] private ContextualNotifications notificationScript;

        private void EndImmortality( ) {

        playerAnimator.SetBool( "IsFading", false );

        }

        public void RespawnPlayer( ) {

        player.gameObject.SetActive( true );

        // Add Immortality
        playerAnimator.SetBool( "IsFading", true );
        Invoke( nameof( EndImmortality ), playerInvincibilityTime );

        invincibilityTimeLeft = playerInvincibilityTime;

        // Respawn Player
        player.transform.position = respawnPoint;
        playerRb.velocity = Vector3.zero;
        playerRb.angularVelocity = Vector3.zero;

        if ( notificationScript != null )
        notificationScript.ToggleTextVisiblity( "Respawn", false );


        }
        public void PlayerDied( bool playerFell ) {

        if ( !playerFell && invincibilityTimeLeft > 0 ) // Player is currently
immortal, no need to subtract lives.
        return;

        player.gameObject.SetActive( false );
        playerDeathSound.Play( );
```

```
        // Make sure that the current scene is not the main menu scene. That
scene features an AI agent, but it has infinite lives.
        if ( SceneManager.GetActiveScene( ).name != "Menu" ) {

        UpdateLifeCountUI( );

        livesLeft--;

        if ( livesLeft < 0 ) { // No lives remaining, send player to the game over
screen.

                gameManager.GameOver( );
                return;

        }

        // Show "Respawning..." Text on UI
        if ( notificationScript != null )
                notificationScript.ToggleTextVisiblity( "Respawn", true );

        }

        Invoke( nameof( RespawnPlayer ), respawnTime );

        }

        public void UpdateLifeCountUI( ) {

        if ( livesLeft - 1 < 0 )
        return;

        UILifeCounter.ElementAt( livesLeft - 1 ).SetActive( false );


        }

        private void Start( ) {

        livesLeft = totalLives;

        playerRb = player.gameObject.GetComponent<Rigidbody>( );
        playerAnimator = player.gameObject.GetComponent<Animator>( );

        }

        // Update is called once per frame
```

```
        private void Update( ) {

        if ( invincibilityTimeLeft > 0 ) {

        invincibilityTimeLeft -= Time.deltaTime;

        }

        }
        }

}
```

ProjectileImpact.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Projectile Impact
 * Script Objective: Handles projectile interactions and special effects. Asteroids
and rockets both use this script to handle collisions.
 *
 */

using UnityEngine;
using UnityEngine.SceneManagement;

namespace TeamBracket {
        public class ProjectileImpact : MonoBehaviour {

        [SerializeField] private GameObject explosionPrefab;
        [SerializeField] private GameObject specialEffects;
        private GameManager gameManager;

        private AudioSource explosionSound;
        private SpriteRenderer spriteRenderer;

        private bool hasHitTarget = false;
        private bool isAsteroid;

        private Rigidbody rb;
```

```
        private void DestroyObject( ) {

        Destroy( this.gameObject );

        }
        private void Explode( ) {

        explosionSound.pitch = Random.Range( 0.9f, 1.3f );
        explosionSound.Play( );

        // Hide Missle
        spriteRenderer.enabled = false;
        specialEffects.SetActive( false );

        // Create Explosion Effect
        GameObject newExplosion = Instantiate( explosionPrefab );
        newExplosion.transform.position = transform.position;

        // Schedule Destruction of Projectile After Explosion Plays
        Invoke( nameof( DestroyObject ), 4f );

        }

        private void OnCollisionEnter( Collision collision ) {

        // Current Object is Asteroid. If the other object is an asteroid or a
missile, early return here.
        if ( isAsteroid )
        if ( collision.collider.CompareTag( "Asteroid" ) ||
collision.collider.CompareTag( "Missile" ) )
              return;

        // If this collision has already been handled, early return.
        if ( hasHitTarget )
        return;

        hasHitTarget = true;

        // Hit Asteroid
        if ( collision.collider.CompareTag( "Asteroid" ) ) {

        gameManager.AddScore( );
        Destroy( collision.gameObject );

        }
```

```
        // Hit Player
        else if ( collision.collider.CompareTag( "Player" ) ) {

        if ( collision.gameObject.TryGetComponent<PlayerHealth>( out PlayerHealth
playerHealth ) ) {

              playerHealth.PlayerDied( false );

        }

        }

        // Explode Asteroid/Missile
        Explode( );


        }

        private void Start( ) {

        isAsteroid = this.gameObject.CompareTag( "Asteroid" );


        if ( SceneManager.GetActiveScene( ).name != "Menu" )
        gameManager = GameObject.Find( "GameManager" ).GetComponent<GameManager>( );

        spriteRenderer = GetComponent<SpriteRenderer>( );
        explosionSound = GetComponent<AudioSource>( );
        rb = GetComponent<Rigidbody>( );

        rb.sleepThreshold = 0;

        }

        private void Update( ) {

        // Prevent Rigidbodies from Pausing.
        if ( rb.IsSleeping( ) ) {

        rb.WakeUp( );

        }

        }

        }
```

```
}
```

RandomizeSprite.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Randomize Sprite
 * Script Objective: Randomly chooses from a list of sprites provided in the Unity
Inspector. The game uses this script to randomize the asteroid appearances.
 *
 */

using System.Collections.Generic;
using System.Linq;
using UnityEngine;

namespace TeamBracket {
	public class RandomizeSprite : MonoBehaviour {

	[SerializeField] private List<Sprite> spriteList = new( );
	private SpriteRenderer spriteRenderer;

	// Start is called before the first frame update
	private void Start( ) {

	// Randomize Sprite & Assign Choice to Sprite Renderer Component
	spriteRenderer = GetComponent<SpriteRenderer>( );
	spriteRenderer.sprite = spriteList.ElementAt( Random.Range( 0,
spriteList.Count ) );

	}

	}

}
```

Spawner.cs:

C/C++

```
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Spawner
 * Script Objective: Spawns a specified object with a list of modifiable parameters.
These parameters are set in the Unity Inspector and will not be modified during
runtime.
 *
 */

using System.Collections;
using UnityEngine;

namespace TeamBracket {
	public class Spawner : MonoBehaviour {

	[SerializeField] private float spawnInterval;
	[SerializeField] private int maxObjectsPerSpawn;
	[SerializeField] private GameObject spawnObjectPrefab;
	[SerializeField] private float maxSpeed;
	[SerializeField] private float spawnHorizontalSpread;

	[SerializeField] private GameManager gameManager;


	public float SpawnInterval => spawnInterval;
	public float MaxObjectsPerSpawn => maxObjectsPerSpawn;

	private IEnumerator spawnCoroutine;

	// Spawn objects forever until the current game ends.
	private IEnumerator SpawnObject( ) {

	while ( this.enabled ) {

	yield return new WaitForSeconds( spawnInterval );

	int numObjectsToSpawn = Random.Range( 1, maxObjectsPerSpawn );

	// Create each object based on numObjectsToSpawn's value.
	for ( int i = 0; i < numObjectsToSpawn; i++ ) {

		// Set random position based on horizontal spread and a random
velocity based on the maxSpeed.
```

```
                Vector3 newSpawnPosition = transform.position + new Vector3(
Random.Range( -spawnHorizontalSpread, spawnHorizontalSpread ), 0, 0 );

                GameObject newSpawnedObject = Instantiate( spawnObjectPrefab,
newSpawnPosition, Quaternion.identity );
                Rigidbody rb = newSpawnedObject.GetComponent<Rigidbody>( );

                rb.velocity = new( Random.Range( -maxSpeed, maxSpeed ), rb.velocity.y,
Random.Range( -maxSpeed, maxSpeed ) );

                rb.angularVelocity = new( 0f, 0f, Random.Range( -1f, 1f ) );

        }

        if ( gameManager != null )
                gameManager.AsteroidsSpawned( numObjectsToSpawn );

        } // End of while loop

        } // End of SpawnObject( )

        // Start is called before the first frame update
        private void Awake( ) {

        spawnCoroutine = SpawnObject( );
        _ = StartCoroutine( spawnCoroutine );


        } // End of Awake( )


        } // End of Spawner class

} // End of namespace
```

StartGame.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Start Game
```

```
 * Script Objective: Starts the game and handles the countdown sequence.
 *
 */

using System.Collections;
using System.Collections.Generic;
using TeamBracket.Movement;
using TMPro;
using UnityEngine;

namespace TeamBracket {
	public class StartGame : MonoBehaviour {

	[SerializeField] private PlayerController movementScript;

	[SerializeField] private GameObject CountdownUI;
	[SerializeField] private TextMeshProUGUI countdownText;
	[SerializeField] private Animator animator;

	[SerializeField] private Stopwatch stopwatch;

	[SerializeField] private List<string> countdownSequence = new( );

	[SerializeField] private GameObject asteroidSpawner;

	private int currentIndex = 0;

	private bool hasCountdownStarted;

	// Start is called before the first frame update
	private void Start( ) {

	CountdownUI.SetActive( false );

	StartCountdown( );
	movementScript.enabled = false;

	}

	private IEnumerator CountdownSequence( ) {

	yield return new WaitForSeconds( 0.25f );

	CountdownUI.SetActive( true );

	while ( currentIndex < countdownSequence.Count ) {
```

```
        countdownText.text = "<mspace=mspace=35>" + countdownSequence[ currentIndex
] + "</mspace>";
        animator.SetBool( "IsPlaying", true );

        currentIndex++;
        yield return new WaitForSeconds( 0.75f );
        animator.SetBool( "IsPlaying", false );

        }

        CountdownUI.SetActive( false );

        movementScript.enabled = true;

        stopwatch.SetStopwatchType( "Stopwatch" );
        stopwatch.SetStopwatchTime( 0f );
        stopwatch.ToggleStopwatch( true );

        asteroidSpawner.SetActive( true );

        }

        public void StartCountdown( ) {

        if ( hasCountdownStarted )
        return;

        hasCountdownStarted = true;

        _ = StartCoroutine( CountdownSequence( ) );

        }

        }

}
```

Stopwatch.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
```

```
 *
 * Name: Stopwatch
 * Script Objective: Keeps track of the player's time spent alive.
 *
 */
using TMPro;
using UnityEngine;

namespace TeamBracket {
	public class Stopwatch : MonoBehaviour {

	[SerializeField] private TextMeshProUGUI stopwatchText;

	private float currentTime = 0f;

	private bool isPlaying;
	private readonly TimeStringCreator timeStringCreator = new( );


	private int direction = 1;

	public void ToggleStopwatch( bool newState ) {

	isPlaying = newState;

	}

	public void SetStopwatchType( string stopwatchType ) {

	direction = stopwatchType == "Stopwatch" ? 1 : -1;


	}

	public void SetStopwatchTime( float newTime ) {

	currentTime = newTime;

	}

	public float GetStopwatchTime( ) {

	return currentTime;

	}
```

```
        private void UpdateUI( ) {

        stopwatchText.text = "<mspace=mspace=25>" +
timeStringCreator.GetNewTimeString( currentTime ) + "</mspace>";

        }

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

        if ( !isPlaying )
        return;


        currentTime += Time.deltaTime * direction;

        }

        private void FixedUpdate( ) {

        UpdateUI( );

        }
        }

}
```

TimeStringCreator.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Time String Creator
 * Script Objective: Used to create a formatted time string by taking a float value as an
argument and returning a string.
 *
 */
using UnityEngine;

public class TimeStringCreator
```

```
{

	private readonly float sixtyDivisor = 1f / 60f; // Doing this calculation
here since multiplication is more performant than division.

	public string GetNewTimeString( float newTime ) {


	int hours = Mathf.FloorToInt( newTime * sixtyDivisor * sixtyDivisor );
	int minutes = Mathf.FloorToInt( newTime * sixtyDivisor );
	int seconds = Mathf.FloorToInt( newTime % 60 );
	int milliseconds = Mathf.FloorToInt( newTime * 100f % 100 );

	return string.Format( "{0:00}:{1:00}:{2:00}.{3:00}", hours, minutes,
seconds, milliseconds % 100 );

	}

}
```

Weapon.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Weapon
 * Script Objective: Handles weapon interactions and movement. Allows for firing
projectiles with customizable settings.
 *
 */
using TMPro;
using UnityEngine;

namespace TeamBracket.WeaponSystem {
	public class Weapon : MonoBehaviour {

	// References
	[Header( "References" )]
	[SerializeField] private Transform character;
```

```
    [SerializeField] private Transform firePoint;
    [SerializeField] private GameObject currentWeapon;
    [SerializeField] private ContextualNotifications notificationScript;

    [Header( "Audio" )]
    [SerializeField] private AudioSource weaponEmptySound;
    [SerializeField] private AudioSource errorSound;
    [SerializeField] private AudioSource bulletFireSound;

    // Internal References
    private Transform weaponTransform;
    private SpriteRenderer currentWeaponSprite;

    // Prefabs
    [Header( "Prefabs" )]
    [SerializeField] private GameObject bulletPrefab;

    // UI
    [Header( "User Interface" )]
    [SerializeField] private TextMeshProUGUI remainingBulletsText;
    [SerializeField] private TextMeshProUGUI ammoText;
    [SerializeField] private GameObject reloadPrompt;

    [Header( "Weapon Settings" )]
    [SerializeField] private float orbitRotationSpeed = 5f;


    // Weapon Attributes
    [Header( "Weapon Attributes" )]
    private int ammo;
    private int remainingBullets;

    [SerializeField] private bool infiniteAmmo;



    [SerializeField] private int maxBulletsPerClip;

    [SerializeField] private int bulletsPerTap;
    [SerializeField] private float fireDelay;
    [SerializeField] private float timeBetweenBullets;
    [SerializeField] private float bulletForce;

    [SerializeField] private float reloadTime;

    private float currentDelay;
```

```
        private bool canFire;
        private bool isReloading = false;

        public float FireDelay => fireDelay;

        private void SetWeaponTransform( ) {

        weaponTransform = currentWeapon.transform;
        currentWeaponSprite = currentWeapon.GetComponent<SpriteRenderer>( );

        } // End of SetWeaponTransform( )

        // Start is called before the first frame update
        private void Start( ) {

        SetWeaponTransform( );

        Reload( );

        canFire = true;

        UpdateWeaponUI( );

        } // Endof Start( )

        private void LookAtMousePosition( ) { // Using this for Rotating Weapon
Angle: https://www.youtube.com/watch?v=Geb_PnF1wOk

        // Set Weapon Position
        Vector3 characterPosition = character.position;
        transform.position = characterPosition;

        // Calculate Facing Direction
        Vector3 mousePosition = Camera.main.ScreenToWorldPoint( Input.mousePosition
);
        Vector3 direction = mousePosition - characterPosition;

        float pointAngle = Mathf.Atan2( direction.y, direction.x ) * Mathf.Rad2Deg;


        // Point Weapon Towards Mouse
        weaponTransform.rotation = Quaternion.AngleAxis( pointAngle,
Vector3.forward );

        // Orbit Weapon Around Player
```

```
        transform.rotation = Quaternion.Slerp( transform.rotation,
Quaternion.Euler( 0, 0, pointAngle ), orbitRotationSpeed * Time.deltaTime );

        // Flip Weapon Based on Orientation
        currentWeaponSprite.flipY = transform.localEulerAngles.z > 90f &&
transform.localEulerAngles.z < 270;


        } // Endof LookAtMousePosition( )

        private void ShootBullet( ) {

        bulletFireSound.pitch = Random.Range( 0.9f, 1.25f );
        bulletFireSound.Play( );

        GameObject newBullet = Instantiate( bulletPrefab );
        newBullet.transform.position = firePoint.position;

        Vector3 shootDirection = ( firePoint.position - transform.position
).normalized;
        float shootAngle = Mathf.Atan2( shootDirection.y, shootDirection.x ) *
Mathf.Rad2Deg;


        newBullet.transform.rotation = Quaternion.Euler( 0, 0, shootAngle - 90f );

        Rigidbody rigidbody = newBullet.GetComponent<Rigidbody>( );

        Vector3 bulletDirection = firePoint.right * bulletForce;
        rigidbody.AddForce( bulletDirection, ForceMode.Force );

        } // Endof ShootBullet( )

        private void UpdateWeaponUI( ) {

        remainingBulletsText.text = isReloading ? "<b>-</b>" : "<b>" +
remainingBullets.ToString( ) + "</b>";
        ammoText.text = infiniteAmmo ? "∞" : ammo.ToString( );

        }

        private void Reload( ) {


        if ( infiniteAmmo ) {
```

```
        remainingBullets = maxBulletsPerClip;

        }
        else {

        // If current ammo exceeds the required bullets, subtract
maxBulletsPerClip from ammo.
        if ( ammo > maxBulletsPerClip ) {

                int bulletsToSubtract = maxBulletsPerClip - remainingBullets;

                remainingBullets = maxBulletsPerClip;
                ammo -= bulletsToSubtract;

        }
        else { // Set remaining bullets equal to the rest of the ammo left.

                remainingBullets = ammo;
                ammo = 0;

        }

        }

        isReloading = false;
        UpdateWeaponUI( );

        notificationScript.ToggleTextVisiblity( "Reload", false );

        } // Endof Reload( )

        private void StartReload( ) {

        reloadPrompt.SetActive( false );

        // Weapon does not need to be reloaded.
        if ( remainingBullets == maxBulletsPerClip )
        return;

        notificationScript.ToggleTextVisiblity( "Reload", true );

        isReloading = true;
        UpdateWeaponUI( );

        Invoke( nameof( Reload ), reloadTime );
```

```
}

private void Fire( ) {

canFire = true;

if ( isReloading ) {

errorSound.Play( );
return;

}

int bulletsToFire = bulletsPerTap;

if ( remainingBullets < bulletsPerTap )
bulletsToFire = remainingBullets;



remainingBullets -= bulletsToFire;


if ( remainingBullets <= 0 ) {

reloadPrompt.SetActive( true );

canFire = false;
remainingBullets = 0;

}

UpdateWeaponUI( );


if ( !canFire ) {

weaponEmptySound.Play( );
return;

}

if ( bulletsToFire > 1 ) {

for ( int i = 0; i < bulletsToFire; i++ ) {

       Invoke( nameof( ShootBullet ), timeBetweenBullets * i );
```

```
        }

        }
        else {

        ShootBullet( );

        }

        currentDelay = fireDelay;

        } // Endof Fire( )


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

        // If weapon is attached to the player, allow weapon movement.
        if ( weaponTransform ) {

        LookAtMousePosition( );

        }

        // Cannot fire gun, so do an early return here.
        if ( currentDelay > 0f ) {

        currentDelay -= Time.deltaTime;
        return;
        }

        // Fire Button
        if ( Input.GetMouseButtonDown( 0 ) )
        Fire( );

        // Reload Button
        if ( Input.GetKeyDown( KeyCode.R ) )
        StartReload( );

        } // Endof Update()

        } // Endof Weapon

} // Endof namespace
```

High Scores

HighScoreLister.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: High Score Lister
 * Script Objective: Lists all high scores in the "Scores" scene in a UI list.
 *
 */
using System.Collections.Generic;
using System.Linq;
using TMPro;
using UnityEngine;

namespace TeamBracket.HighScores {
	public class HighScoreLister : MonoBehaviour {

	[Header( "References" )]
	[SerializeField] private GameObject listingPrefab;
	[SerializeField] private GameObject scoreContainer;
	[SerializeField] private HighScoreManager highScoreManager;

	private readonly TimeStringCreator timeStringCreator = new( );

	// Start is called before the first frame update
	void Start( ) {

	LoadScores( );
	}

	void LoadScores( ) {

	if ( highScoreManager.IsLoaded ) { // Scores loaded, show them in the list.

	ShowScores( );

	} else { // Keep waiting until the file is loaded completely.

	Invoke( nameof( LoadScores ), 0.1f );

	}

	}
```

```
        void ShowScores( ) {

        // Get all high scores.
        List<HighScore> highScores = highScoreManager.HighScores;

        for ( int i = 0; i < 10; i++ ) {

        // If current index exceeds the list of high scores, break from the list
so that the loop doesn't go out of bounds.
        if ( i > highScores.Count - 1 )
                break;

        GameObject newListing = Instantiate( listingPrefab );

        // Update Rank Text
        newListing.transform.GetChild( 0 ).GetComponent< TextMeshProUGUI >( ).text =
( i + 1 ).ToString( );

        // Update Date Text
        newListing.transform.GetChild( 1 ).GetComponent<TextMeshProUGUI>( ).text =
highScores.ElementAt( i ).date;

        // Update Overall Score Text
        newListing.transform.GetChild( 2 ).GetComponent<TextMeshProUGUI>( ).text =
string.Format( "{0:0.00}", highScores.ElementAt( i ).overallScore );

        // Update Time Survived Text
        newListing.transform.GetChild( 3 ).GetComponent<TextMeshProUGUI>( ).text =
timeStringCreator.GetNewTimeString( highScores.ElementAt( i ).timeSurvived );

        // Update Asteroids Destroyed Text
        newListing.transform.GetChild( 4 ).GetComponent<TextMeshProUGUI>( ).text =
highScores.ElementAt( i ).asteroidsDestroyed.ToString( );

        // Add Listing to High Scores List
        newListing.transform.SetParent( scoreContainer.transform );

        }

        }

        }

}
```

HighScoreManager.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: High Score Manager
 * Script Objective: Manages Saving/Loading High Scores List and Adding New Scores
 *
 * Source(s):
 *
https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.crypt
ostream?view=net-8.0
 *      https://videlais.com/2021/02/28/encrypting-game-data-with-unity/
 *     https://www.youtube.com/watch?v=32zOlthWX2s
 *
 */
using System;
using System.Collections.Generic;
using System.IO;
using System.Security.Cryptography;
using System.Text;
using UnityEngine;

namespace TeamBracket.HighScores {

	[System.Serializable]
	public class HighScore {

	public float timeSurvived;
	public int asteroidsDestroyed;
	public float overallScore;
	public string date;

	public HighScore( int asteroidsDestroyed, float timeSurvived ) {

	this.asteroidsDestroyed = asteroidsDestroyed;
	this.timeSurvived = timeSurvived;
	date = DateTime.Now.ToString( "MM/dd/yy" );

	}

	}

	[System.Serializable]
```

```
        public class HighScoresWrapper {

        public List<HighScore> scores;

        public HighScoresWrapper( List<HighScore> scores ) {

        this.scores = scores;

        }

        }

        public class HighScoreManager : MonoBehaviour {

        // FIXME: Unserialize this after testing is completed.
        [SerializeField] private List<HighScore> sortedScores = new( );

        [Header( "Points Allocation" )]
        [SerializeField] private float survivalPointsPerSecond = 1f;
        [SerializeField] private float pointsPerAsteroidDestroyed = 10f;


        private readonly string saveFileName = "scores.json";
        private bool isLoaded = false;

        // Readonly Public Variables. The targeted variables should NOT be
changed during runtime and only should be assigned via the Unity Inspector
window.
        public bool IsLoaded => isLoaded;
        public float SurvivalPointsPerSecond => survivalPointsPerSecond;
        public float PointsPerAsteroidDestroyed => survivalPointsPerSecond;
        public List<HighScore> HighScores => sortedScores;

        private void Start( ) {

        LoadScores( );


        }


        // Create HighScore object and calculate overall score based on the
specified formula.
        private HighScore CreateHighScore( int asteroidsDestroyed, float
timeSurvived ) {
```

```
        HighScore newHighScore = new( asteroidsDestroyed, timeSurvived ) {
        overallScore = ( timeSurvived * survivalPointsPerSecond ) + (
asteroidsDestroyed * pointsPerAsteroidDestroyed )
        };

        return newHighScore;

        }

        // This function is called by GameManager after the game is over.
        public void AddScore( int asteroidsDestroyed, float timeSurvived ) {

        // Add Score to List
        sortedScores.Add( CreateHighScore( asteroidsDestroyed, timeSurvived ) );

        // Sort List From Highest To Lowest
        sortedScores.Sort( ( a, b ) => b.overallScore.CompareTo( a.overallScore ) );

        if ( sortedScores.Count > 10 ) {

        sortedScores.RemoveRange( 10, sortedScores.Count - 10 );

        }

        // Save Scores to File
        SaveScores( );

        }

        public HighScore GetScore( int asteroidsDestroyed, float timeSurvived ) {

        HighScore foundScore = sortedScores.Find( score => score.timeSurvived ==
timeSurvived && score.asteroidsDestroyed == asteroidsDestroyed );

        return foundScore;

        }

        // Save Score List to File
        private void SaveScores( ) {

        try {
        string filePath = Path.Combine( Application.persistentDataPath, saveFileName
);

        //Convert data into json which hold all info and structure of data
```

```
        string json = JsonUtility.ToJson( new HighScoresWrapper( sortedScores ), true
);

        //Encrypt json
        string encryptedJson = EncryptString( json, "hardcodedKey" );

        //Write to encrypted json while open
        using ( StreamWriter writer = new( filePath ) ) {
               writer.Write( encryptedJson );
        }

        Debug.Log( "Scores saved successfully." );

        }
        catch ( Exception e ) {
        Debug.LogError( "Error saving scores: " + e.Message );
        }
        }

        private string EncryptString( string plainText, string key ) {

        //Advanced Encryption Standard
        //creating new instance of aes class and aesAlg variable
        using Aes aesAlg = Aes.Create( );
        //key is converting key string to byte array using UTF-8
        aesAlg.Key = Encoding.UTF8.GetBytes( key.PadRight( 32 ) );
        aesAlg.IV = new byte[ 16 ];

        // Create an encryptor to perform the stream transform.
        ICryptoTransform encryptor = aesAlg.CreateEncryptor( aesAlg.Key, aesAlg.IV
);

        //Using MS CryptoStream to encrypt the string with Key and IV defined.
        using MemoryStream msEncrypt = new( );
        using ( CryptoStream csEncrypt = new( msEncrypt, encryptor,
CryptoStreamMode.Write ) ) {
        using StreamWriter swEncrypt = new( csEncrypt );
        // Write all data to the stream.
        swEncrypt.Write( plainText );
        }

        return Convert.ToBase64String( msEncrypt.ToArray( ) );

        }
```

```
        // Load Score List From File
        private void LoadScores( ) {

        string filePath = Application.persistentDataPath + "/" + saveFileName;


        if ( File.Exists( filePath ) ) {

        try {

                using StreamReader reader = new( filePath );

                //Read encrypted JSON from file
                string encryptedJson = reader.ReadToEnd( );

                //Decrypt JSON before sending any info to screen
                string json = DecryptString( encryptedJson, "hardcodedKey" );

                HighScoresWrapper wrapper = JsonUtility.FromJson<HighScoresWrapper>(
json );

                if ( wrapper != null ) {

                sortedScores = wrapper.scores;

                }

        }
        catch ( Exception e ) {

                Debug.Log( "Error when loading saved scores: " + e.Message );

        }

        }

        isLoaded = true;

        }

        private string DecryptString( string cipherText, string key ) {

        using Aes aesAlg = Aes.Create( );
        aesAlg.Key = Encoding.UTF8.GetBytes( key.PadRight( 32 ) );
        aesAlg.IV = new byte[ 16 ];

        // Create a decryptor to perform the stream transform.
```

```
        ICryptoTransform decryptor = aesAlg.CreateDecryptor( aesAlg.Key, aesAlg.IV
);

        //Using MS CryptoStream to decrypt the data
        using MemoryStream msDecrypt = new( Convert.FromBase64String( cipherText ) );
        using CryptoStream csDecrypt = new( msDecrypt, decryptor,
CryptoStreamMode.Read );
        using StreamReader srDecrypt = new( csDecrypt );
        //read decrypted bytes and return to string format to be returned
        return srDecrypt.ReadToEnd( );

        }

        }

}
```

Menu

CreditsHandler.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Credits Handler
 * Script Objective: Shows all game credits and then returns to the main menu.
 *
 */

using UnityEngine;
using TMPro;
using UnityEngine.SceneManagement;

namespace TeamBracket.Credits {

        [System.Serializable]
        public class Credit {

        public string title;
        public string source;
```

```
        }

        public class CreditsHandler : MonoBehaviour {

        [SerializeField] private Credit[ ] creditsList;
        [SerializeField] private float fadeDuration = 1f;
        [SerializeField] private float displayTextDuration = 1.5f;

        private bool completed = false;
        private int currentIndex = 0;

        private float m_Timer;


        public TextMeshProUGUI titleText;
        public TextMeshProUGUI sourceText;

        private void GoToMenu( ) {

        SceneManager.LoadScene( "Menu" );

        }

        private void SetText( Credit currentElement ) {

        titleText.text = currentElement.title;
        sourceText.text = currentElement.source;

        }

        // Start is called before the first frame update
        private void Start( ) {

        SetText( creditsList[ currentIndex ] );

        }

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

        if ( Input.GetKeyDown( KeyCode.Escape ) || Input.GetKeyDown( KeyCode.Space ) )
{

        if ( !completed ) {

               completed = true;
```

```
            GoToMenu( );
            return;

        }

        }

        m_Timer += Time.deltaTime;

        if ( completed )
        return;

        // Fade Text
        titleText.alpha = m_Timer / fadeDuration;
        sourceText.alpha = m_Timer / fadeDuration;

        if ( m_Timer < fadeDuration + displayTextDuration ) {
        return;
        }

        // Reset Credit Display
        titleText.alpha = 0;
        sourceText.alpha = 0;
        m_Timer = 0;

        // Credits Finished. Go to Main Menu
        if ( currentIndex >= creditsList.Length - 1 ) {

        completed = true;
        titleText.alpha = 1;
        sourceText.alpha = 1;

        GoToMenu( );
        return;

        }

        // Credits Not Finished. Go to Next Credit
        currentIndex++;
        SetText( creditsList[ currentIndex ] );

        }

        }

}
```

InGameMenuHandler.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: In-Game Menu Handler
 * Script Objective: Toggles the visibility of the in-game menu.
 *
 */

using UnityEngine;

namespace TeamBracket.UI {

	public class InGameMenuHandler : MonoBehaviour {

	[SerializeField] private GameObject menu;

	private void Start( ) {

	menu.SetActive( false );

	}

	private void Update( ) {

	if ( Input.GetKeyDown( KeyCode.Escape ) ) {

	menu.SetActive( !menu.activeInHierarchy );

	}

	}

	}

}
```

MenuHandler.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Menu Handler
 * Script Objective: Handles common methods between scenes.
 *
 * Source(s):
https://stackoverflow.com/questions/56145437/how-to-make-textmesh-pro-input-fie
ld-deselect-on-enter-key
 *
 */

using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.SceneManagement;

namespace TeamBracket.UI {

	public class MenuHandler : MonoBehaviour {

	public void GoToScene( string sceneName ) {

	SceneManager.LoadScene( sceneName );

	}

	public void QuitGame( ) {

	Application.Quit( );

	}

	public void DeselectButton( ) {

	var eventSystem = EventSystem.current;
	if ( eventSystem.alreadySelecting )
	eventSystem.SetSelectedGameObject( null );

	}

	}

}
```

SoundHandler.cs:

```C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Sound Handler
 * Script Objective: Allows the player to change their audio settings in the "Settings"
scene.
 *
 * Source(s):
https://docs.unity3d.com/2018.2/Documentation/ScriptReference/UI.Slider-onValue
Changed.html
 *
 */

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Audio;
using UnityEngine.UI;

namespace TeamBracket.Audio {
	public class SoundHandler : MonoBehaviour {

	[SerializeField] private List<Slider> sliderList = new( );
	[SerializeField] private List<AudioMixer> audioMixerList = new( );

	// Dictionaries can't be serialized in Unity, so lists are used to
populate and save the references,
	// and then the dictionaries are created in Start().
	private readonly Dictionary<string, Slider> musicSliders = new( );
	private readonly Dictionary<string, AudioMixer> audioMixers = new( );

	private void Start( ) {

	// Populate dictionaries.
	foreach ( var musicGroup in audioMixerList ) {

	audioMixers[ musicGroup.name ] = musicGroup;

	}

	foreach ( var slider in sliderList ) {

	string sliderName = slider.gameObject.name;
	musicSliders[ sliderName ] = slider;
```

```
        // Add Slider Callback
        slider.onValueChanged.AddListener( delegate { OnSliderValueChanged(
sliderName ); } );

        float storedVolumeValue = PlayerPrefs.GetFloat( sliderName );

        SetSliderValue( sliderName, DecibelToPercentage( storedVolumeValue, -60f,
10f ) );

        }

        }

        private void SetSliderValue( string sliderName, float value ) {
        musicSliders[ sliderName ].value = value;
        }

        // min + (value * (max - min)) = Decibel (-80db - 20db)
        private float PercentageToDecibel( float value, float min, float max ) {
        return Mathf.Clamp( min + value * ( max - min ), min, max );
        }

        // (value - min) / (max - min) = Percentage (0 - 1)
        private float DecibelToPercentage( float value, float min, float max ) {
        return Mathf.Clamp( Mathf.Abs( value - min ) / Mathf.Abs( max - min ), 0f, 1f );
        }

        private void OnSliderValueChanged( string sliderName ) {

        float newVolumeValue = PercentageToDecibel( musicSliders[ sliderName
].value, -60f, 10f );

        audioMixers[ sliderName ].SetFloat( sliderName, newVolumeValue );

        }

        // When the scene changes, save the volume settings.
        private void OnDestroy( ) {

        SaveVolumeSettings( );

        }

        private void SaveVolumeSettings( ) {
```

```
        foreach ( AudioMixer mixer in audioMixerList ) {

        mixer.GetFloat( mixer.name, out float currentValue );
        PlayerPrefs.SetFloat( mixer.name, currentValue );

        }

        }

        }

}
```

SoundPrefsLoader.cs:

```
C/C++
/*
 * Team: Team Bracket (Team 1)
 * Course: CSC-440-101
 *
 * Name: Sound Prefs Loader
 * Script Objective: Allows the game to save and load audio settings.
 *
 * Source(s):
https://stackoverflow.com/questions/68902203/dont-destroy-on-load-creates-multi
ple-objects
 *      https://discussions.unity.com/t/how-do-i-use-audiomixer-getfloat/141433
 *
 */

using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Audio;

namespace TeamBracket.Audio {
        public class SoundPrefsLoader : MonoBehaviour {

        [SerializeField] private List<AudioMixer> audioMixers = new( );

        public static SoundPrefsLoader instance;

        void Start( ) {
```

```
        // Load volume settings once the game starts.
        LoadVolumeSettings( );

        }

        private void Awake( ) {

        // Tell Unity to not destroy this object as we'll need it to save the
volume settings once the application quits.
        if ( instance == null ) {

        instance = this;
        DontDestroyOnLoad( this.gameObject );

        }
        else {

        Destroy( this.gameObject );

        }

        }

        // When the game closes, save the settings in case they haven't been
saved earlier.
        private void OnApplicationQuit( ) {

        SaveVolumeSettings( );

        }

        private void LoadVolumeSettings( ) {

        foreach ( AudioMixer mixer in audioMixers ) {

        // If no value is found, it defaults to 0 db.
        float savedValue = PlayerPrefs.GetFloat( mixer.name );
        mixer.SetFloat( mixer.name, savedValue );

        }

        }

        private void SaveVolumeSettings( ) {

        foreach ( AudioMixer mixer in audioMixers ) {
```

```
        mixer.GetFloat( mixer.name, out float currentValue );
        PlayerPrefs.SetFloat( mixer.name, currentValue );

        }

        }

        }

}
```