CS-485: Advanced Game Development Technical Report – Asteroid Belt

Brendan Sting

Professor Turini

Kettering University

Last Revision: 8-18-2024

(Changes to initial or past designs and versions are highlighted in blue.)

Description To Player

"Asteroid Belt" poses as a 3-dimensional spin-off of the classic game, "Asteroids", which is a 1979 arcade top-down space shooter game whose objective is destroy all enemy asteroids without colliding with any of them, else you lose the game. In "Asteroid Belt", you won't just be playing top-down as a spaceship shooting incoming asteroids, rather, you'll mainly play inside the cockpit of a 3D spaceship to shoot asteroids from a first-person perspective. Now, instead of just being able to control your spacecraft side to side and forward, you'll gain full zero-gravity maneuverability to go not only forward, but roll, turn, and tilt your spaceship! With a whole new dimension added to the base "Asteroids", you can shoot and dodge asteroid rocks with much more precision. But now, you'll have blind-spots to worry about, which begs the question, can you survive and destroy the oncoming asteroids with your newfound controls?

Description To Developer

Getting into the technical side of "Asteroid Belt", it is a 3D game that is meant to be played on a typical US keyboard layout, which means that its target platform is for PC. As previously mentioned, this game will take on the paradigm of being a first-person shooter video game, but all player movements will be restricted to vehicular movements (a.k.a., the player moves by driving the spaceship, and is confined to the spaceship). The spacecraft will be capable of 4 aircraft DOF (Degrees Of Freedom): "yaw", "pitch", "roll", and "drive/throttle". Since these DOF should have bi-directional axis values, the standard WASD keyboard controls will not work standalone and thus mouse positions will be brought in to support the extra degrees of freedom for ship controls. Firing mechanics, however, will remain relatively standard as the left-mouse button (LMB) will be the primary fire trigger button to shoot projectiles from the ship. Note that a boost button will be added in development to give the player more fast-paced control over the spacecraft; this would most likely be bound to the SHIFT key if implemented.

Moving outside the scope of the player's ship, the outer environment of the game will of course be a chosen galaxy-themed skybox to immerse the player in a galactic setting. In turn, the theme will be reminiscent of the original Asteroids' space shooter theme, just with more advanced graphics than plain sprite outlines. Furthermore, to add a more 3D effect to the space environment, a particle system can be used to generate "space dust" around the ship, with its quantity adjusted later on in case it obstructs field-of-view (FOV) in any way. Additionally, for terrain management, the environment will be semi open-world with the player being confined to a 3D teleportation treadmill. This will allow for a confined playable roaming space that avoids reaching a breakable transform position in the game world.

Lastly, enemies will of course be floating asteroids hurdling towards the ship at a reasonable speed that the player can dodge or shoot at and can be further improved with highlighting shaders when the player is aiming at their hitbox collider. These asteroid enemies will spawn relative to the player's location to keep pace with their position in the playable area treadmill.

Visuals

To give a better visual representation of what "Asteroid Belt" will look like, here are some graphics of its concepts as well as the original base "Asteroids" game it will build off of:

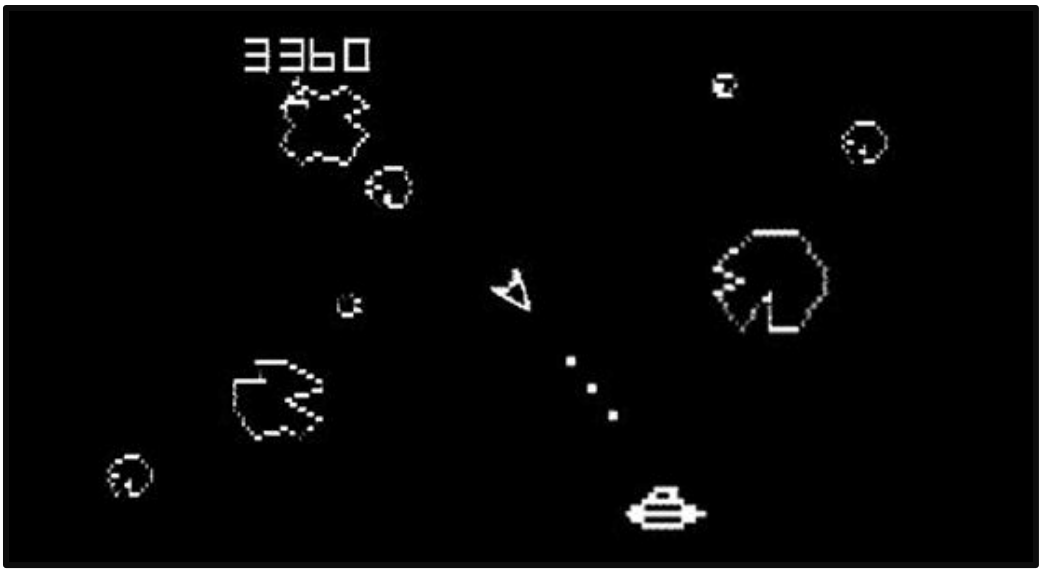

Figure 1. A screenshot of the original 1979 "Asteroids" arcade game, ported on the Atari.

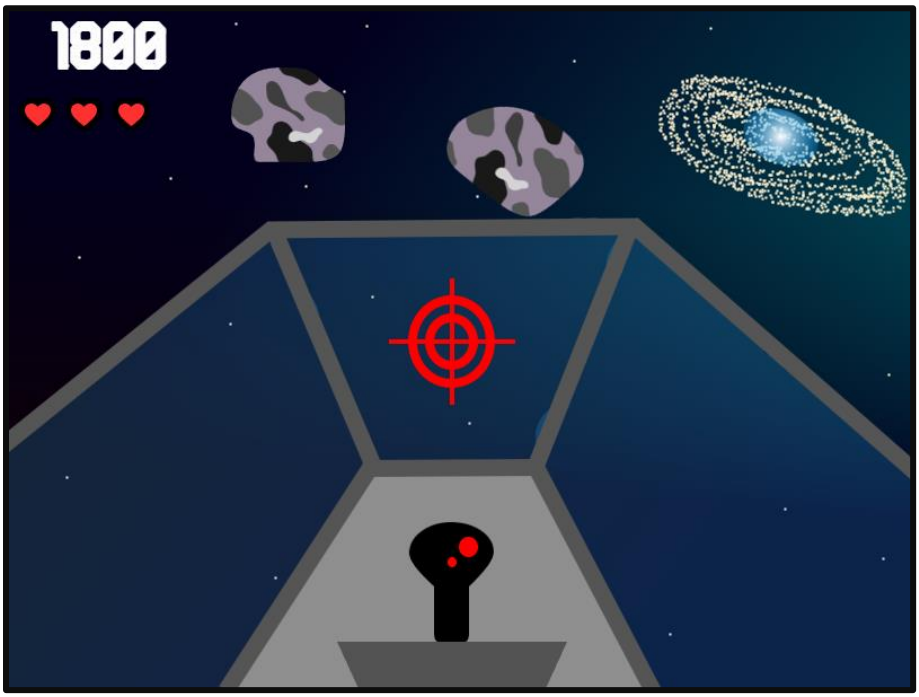

Figure 2. A graphic sketch of the "Asteroid Belt" spin-off game, from cockpit perspective (roof removed to show space skybox.

Figure 3. A graphic sketch of the UI pause screen in "Asteroid Belt", displayed as an overlay.

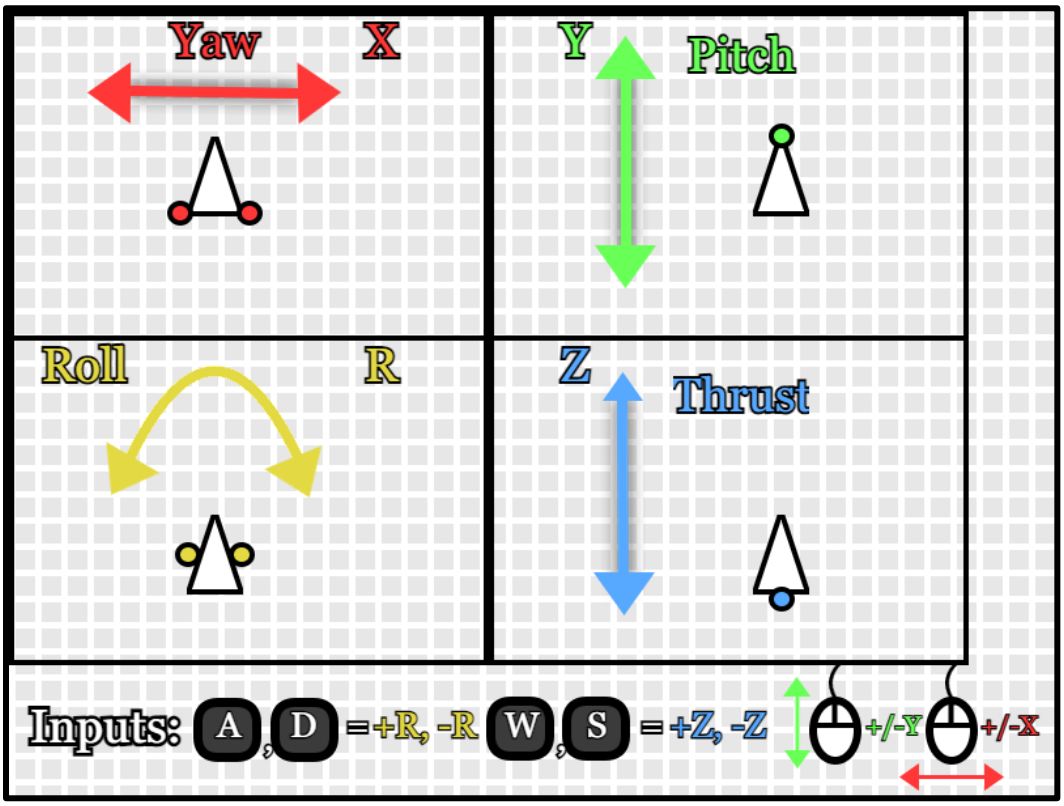

Figure 4. A graphic sketch of the default controls schema in "Asteroid Belt" for the ship.

Modules

ESSENTIAL (YELLOW), NON-ESSENTIAL (REGULAR)

- Throttle Engine SFX (sound effects)
- Fire Ship Cannons SFX
- Background Music/Soundtrack (.Wav file[s])
- Player Ship Model (3D) Asset
- Asteroid Model (3D) Asset
- UFO Model (3D) Asset
    - Optional, but would be an advanced enemy unit
- Text Mesh Pro/TMP Unity Asset (UI-extension)

- Heart UI Icon (16-bit preferred)
- Emery Font Asset (.TTF or .OTF)
    - Can also interchange for another bold and/or retro font if not open-source
    - Currently using Brose (TTF) font as Emery is confirmed not open-source.
- Crosshair Icon
- Universal Render Pipeline Asset
- Particle FX (effects)
    - E.g., ship cannons/blasters, thrusters, space dust, etc.
- Galaxy Skybox(es)
- Space UI SFX (sci-fi clicking sounds for UI buttons, etc.)
- Planet Assets (static environment)
- Space Station + Satellite Asset(s) (static environment)
- Sci-fi/Space UI Icons (image assets)

Development

Diving into development, the implementation of "Asteroid Belt" first began with focusing on the player's input and controller, which would be their spaceship in the game. The thought of starting with the player component was driven by the plan to first get the foundation for the player's aircraft movement working and then build around and off of this core feature in terms of environment, UI, interactions, etc. This is a common development approach, but in hindsight after doing ~50% of developmental tasks, it may have been better to work-out the finer details about managing the playable area since a "player treadmill" feature would inherently affect player input; i.e., controlling the terrain's movement instead of the player's ship to give the illusion of movement but keeping them in a static position. Regardless, the game is already very far into development and a re-work of the foundational controls seems infeasible with the remaining deliverables still due in the time given. However, some VFX and juice will be added to the current state of player movement to still smooth out player experience where possible. That being said, let's get into what's currently been contributed dev-wise to the game, "Asteroid Belt", starting with the player asset.

Player Asset & Controls

Beginning with the player component of the game, this is designed to be a bundled prefab that contains all the meshes, models, and scripts to foster the core in-game input. The process of building up this prefab first started with getting a spaceship asset from the Unity Asset Store. The initial search for a player spaceship asset was daunting, but a little help from a YouTube channel, Midnite Oil Software LLC, who helped recommend and showcase a free modular spaceship asset sample pack provided by Ebal Studios. From the downloaded spaceship sample pack, the red ship was chosen as the player's ship for the reason it pops out the most and has a well-balanced

model with where its wings and blasters are positioned. Here is a figure of the ship's render upon importing and adding to a Unity scene:

Figure 5. Red modular ship asset used for player's prefab and controls.

Note that some of the red ship model's materials were converted and adjusted into various URP material equivalents to optimize it with the project's URP setup and to allow any player cameras to adequately see through the glass cockpit. Once the asset was fully unpacked and ready to go, the next action item was to figure out how to get the ship moving nicely with Unity's PhysX (physics) engine. Keeping with Midnite Oil Software's *How To Make a 3D Space Shooter Game in Unity – Tutorial*, the video advised a top-down approach to making a ship controller where the controller script would be made first, containing all the physic forces to be applied to the ship asset's RigidBody, then expand scripting downwards to add in a ship movement initialization script (now called "PlayerShipMovementInput.cs" in the "Asteroid Belt" project), an input handler middleman to oversee setting the type of input control, and a player-specific ship movement script to implement a wide range of interface properties that fetch ship

inputs (currently called “PlayerShipMovement.cs” script for the player’s asset). Below in Figure 6 is a sketch of the player’s ship input pipeline altogether.

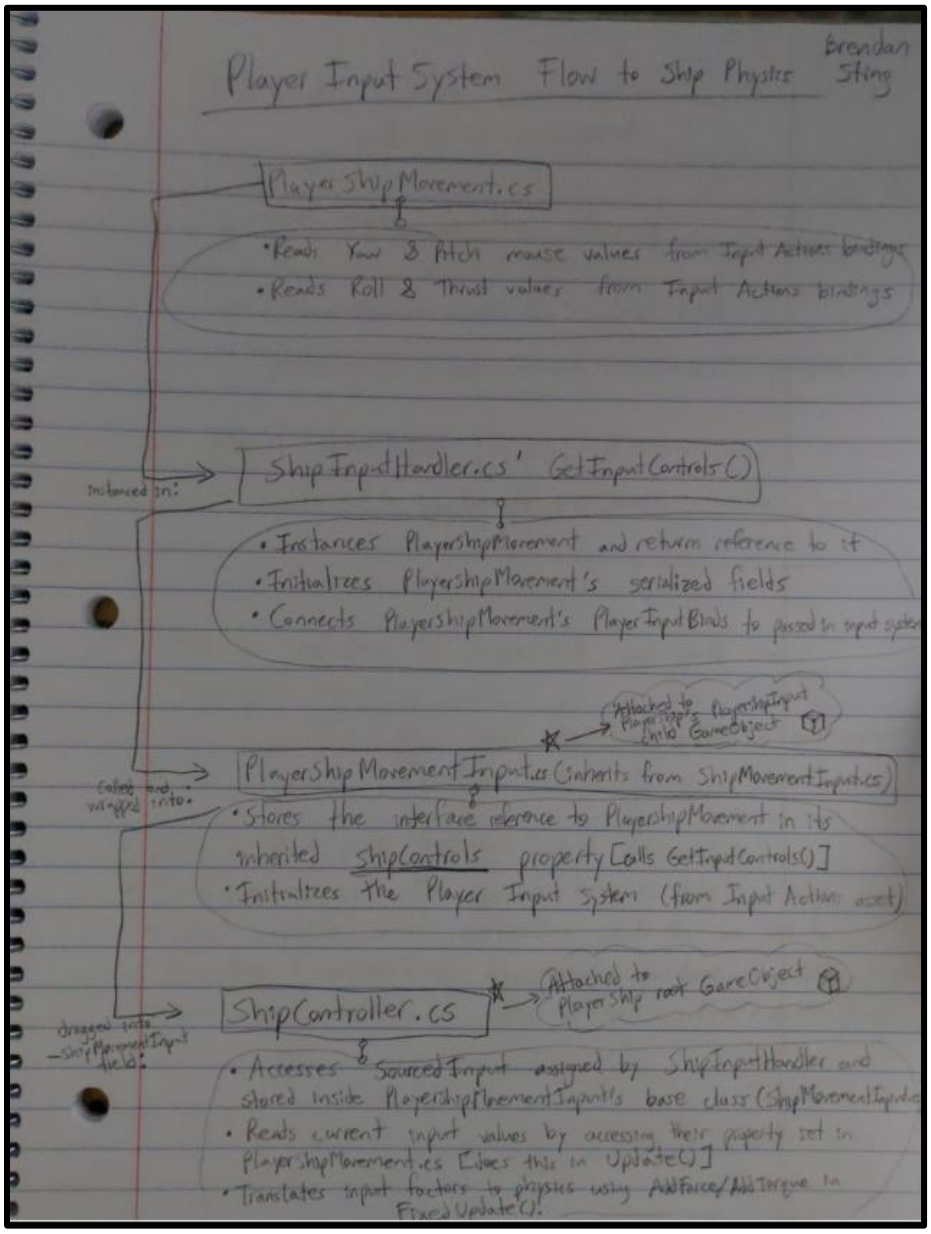

Figure 6. Initial notebook sketch up of player’s input system pipeline that translates input actions to spaceship physics.

Last, but not least for the player input scripts, the interface for the ship's physics inputs were declared in "IShipMovement.cs" script. Later on, the input management scripts were refactored to have base classes that the player ship scripts would inherit from; the base scripts were called "ShipMovementBase.cs" and "ShipMovementInput.cs" and modeled after "PlayerShipMovement.cs" and "PlayerShipMovementInput.cs", respectively. Motivation for this refactoring change was to generalize the controls so that other types of spaceships, like enemy/AI ships for example, could be added later on if development allows. The final code snippets of what currently drives the player's ship movements are given below.

```
/*-----------------------------------
 Custom script made by Brendan Sting
 Date: 7/21/2024
-----------------------------------*/

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

namespace Bsting.Ship
{
    /// <summary>
    /// Base interface to declare ship movement properties that are to be
initialized by PlayerInput.
    /// </summary>
    public interface IShipMovement
    {
        public float PitchFactorInput { get; }
        public float YawFactorInput { get; }
        public float ThrustFactorInput { get; }
        public float RollFactorInput { get; }
        public float HyperspeedFactorInput { get; }
    }
}
```

Code 1. Ship movement interface that declares values to be stored from reading input actions. (Refer to *Figure 4*'s graphic for what pitch, yaw, thrust, and roll refer to on the spaceship.)

```
/*-----------------------------------
 Custom script made by Brendan Sting
 Date: 7/21/2024
-----------------------------------*/

using System.Collections;
```

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

namespace Bsting.Ship.Player
{
    /// <summary>
    /// Class that inherits from ShipMovementBase, in which it reads from a hooked up
PlayerInputActionsAsset "PlayerInputBinds", to determine ship movement values/factors.
    /// </summary>
    public class PlayerShipMovement : ShipMovementBase
    {
        // Public fields:
        [SerializeField] public PlayerInputSystem PlayerInputBinds;
        [SerializeField] public float HyperspeedBoostAmount = 2.0f;
        [HideInInspector, SerializeField] public bool _isPitchInverted = false;
        [HideInInspector, SerializeField] public bool _isMouseFilteredToGameWindow = false;
        [HideInInspector, SerializeField] public float _stepValueToEaseRoll = 1f;
        [HideInInspector, SerializeField] public float _stepValueToEaseThrust = 1f;

        // Private fields:
        [SerializeField] private float _deadzoneRadius = 0.1f;

        // Private var's:
        private Vector2 _foundScreenCenter => new Vector2(Screen.width * 0.5f, Screen.height *
0.5f); // lambda eval.
        private float _easedRollFactor = 0f;
        private float _easedThrustFactor = 0f;

        #region MAIN SPACESHIP INPUT VALUE ACCESSORS
        /* ========   MAIN SPACESHIP INPUT CONNECTORS/ACCESSORS ======== */
        public override float PitchFactorInput
        {
            get
            {
                // ...
                float currentMousePitchPos =
PlayerInputBinds.Player.AircraftPitch.ReadValue<float>();
                float currentMouseYawPos = PlayerInputBinds.Player.AircraftYaw.ReadValue<float>();
                float finalPitchFactor;

                if (_isMouseFilteredToGameWindow)
                {
                    // Added if condition with else { return no pos; } so doesn't keep spinning
ship after moving cursor outside active game window:
                    if ((!IsMouseYPositionOutOfGameWindow(currentMousePitchPos)) &&
(!IsMouseXPositionOutOfGameWindow(currentMouseYawPos)))
                    {
                        finalPitchFactor = FindDeltaPitch(currentMousePitchPos);
                    }
                    else
                    {
                        finalPitchFactor = 0f;
                    }
                }
                else
                {
                    finalPitchFactor = FindDeltaPitch(currentMousePitchPos);
                }

                return finalPitchFactor;
            }
        }

        public override float YawFactorInput
        {
            get
            {
```

```
                // ...
                float currentMouseYawPos = PlayerInputBinds.Player.AircraftYaw.ReadValue<float>();
                float currentMousePitchPos =
PlayerInputBinds.Player.AircraftPitch.ReadValue<float>();
                float finalYawFactor;

                if (_isMouseFilteredToGameWindow)
                {
                    // Added if condition with else { return no pos; } so doesn't keep spinning
ship after moving cursor outside active game window:
                    if ((!IsMouseXPositionOutOfGameWindow(currentMouseYawPos)) &&
(!IsMouseYPositionOutOfGameWindow(currentMousePitchPos)))
                    {
                        finalYawFactor = FindDeltaYaw(currentMouseYawPos);
                    }
                    else
                    {
                        finalYawFactor = 0f;
                    }
                }
                else
                {
                    finalYawFactor = FindDeltaYaw(currentMouseYawPos);
                }

                return finalYawFactor;
            }
        }

        public override float ThrustFactorInput
        {
            get
            {
                // ...
                float currentThrustSignedVal =
PlayerInputBinds.Player.AircraftThrust.ReadValue<float>();

                // ADDED
                float currentEstimatedThrust =
Mathf.Approximately(Mathf.Abs(currentThrustSignedVal), 0f) ? 0f : currentThrustSignedVal;
                _easedThrustFactor = Mathf.Lerp(_easedThrustFactor, currentEstimatedThrust,
Time.deltaTime * _stepValueToEaseThrust);

                // If there's nothing sensed on the bind, then return no factor on thrust:
                return _easedThrustFactor;
            }
        }

        public override float RollFactorInput
        {
            get
            {
                // ...
                float currentRollSignedVal =
PlayerInputBinds.Player.AircraftRoll.ReadValue<float>();

                // ADDED
                float currentEstimatedRoll = Mathf.Approximately(Mathf.Abs(currentRollSignedVal),
0f) ? 0f : currentRollSignedVal;
                _easedRollFactor = Mathf.Lerp(_easedRollFactor, currentEstimatedRoll,
Time.deltaTime * _stepValueToEaseRoll);

                // If there's nothing sensed on the bind, then return no factor on thrust:
                return _easedRollFactor;
            }
        }

        public override float HyperspeedFactorInput
        {
```

```
            get
            {
                // ...
                float currentBoostSignedVal =
PlayerInputBinds.Player.AircraftHyperSpeed.ReadValue<float>();

                // ADDING
                float currentEstimatedBoost = Mathf.Approximately(Mathf.Abs(currentBoostSignedVal),
0f) ? 1.0f : (currentBoostSignedVal * HyperspeedBoostAmount);

                return currentEstimatedBoost;
            }
        }
        /* ========   END OF MAIN SPACESHIP INPUT CONNECTORS/ACCESSORS ======== */
        #endregion

        #region Helper Function(s)
        private bool IsMouseXPositionOutOfGameWindow(float currentMouseXPos)
        {
            bool result = false;

            if ((currentMouseXPos < 0) || (currentMouseXPos > Screen.width))
            {
                result = true;
            }

            return result;
        }

        private bool IsMouseYPositionOutOfGameWindow(float currentMouseYPos)
        {
            bool result = false;

            if ((currentMouseYPos < 0) || (currentMouseYPos > Screen.height))
            {
                result = true;
            }

            return result;
        }

        private float FindDeltaPitch(float currentPitchPos)
        {
            float deltaPitchRatioFromCenter = (currentPitchPos - _foundScreenCenter.y) /
_foundScreenCenter.y;
            if (!_isPitchInverted) { deltaPitchRatioFromCenter *= -1; }

            // If position (usually mouse) has been moved outside deadzone, then apply the new
pitch:
            return Mathf.Abs(deltaPitchRatioFromCenter) > _deadzoneRadius ?
deltaPitchRatioFromCenter : 0f;
        }

        private float FindDeltaYaw(float currentYawPos)
        {
            float deltaYawRatioFromCenter = (currentYawPos - _foundScreenCenter.x) /
_foundScreenCenter.x;

            // If position (usually mouse) has been moved outside deadzone, then apply the new yaw:
            return Mathf.Abs(deltaYawRatioFromCenter) > _deadzoneRadius ? deltaYawRatioFromCenter :
0f;
        }
        #endregion
    }
}
```

Code 2. The root PlayerShipMovement script that parses raw input values for the player's ship.

```
/*-----------------------------------
 Custom script made by Brendan Sting
 Date: 7/21/2024
-----------------------------------*/

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Bsting.Ship.Player;

namespace Bsting.Ship
{
    /// <summary>
    /// Class to handle configuring ShipMovement input values and how their InputActionMaps are
hooked up.
    /// </summary>
    public class ShipInputHandler : MonoBehaviour
    {
        #region Enumeration Over Input Types
        public enum InputType
        {
            Player,
            EnemyAI
        }
        #endregion

        #region Helper Function(s)
        public static IShipMovement GetInputControls(InputType inputType,
                                                     PlayerInputSystem playerInputMap = null,
                                                     bool shouldInvertPitch = false,
                                                     bool shouldFilterMousePos = false,
                                                     float stepFactorToEaseRoll = 1.0f,
                                                     float stepFactorToEaseThrust = 1.0f,
                                                     float hyperspeedBoostFactor = 2.0f)
        {
            IShipMovement determinedSource = null;

            switch (inputType)
            {
                case InputType.Player:
                    // Instantiate:
                    PlayerShipMovement newPlayerMovement = new PlayerShipMovement();

                    // Setup:
                    if (playerInputMap != null) {
                         newPlayerMovement.PlayerInputBinds = playerInputMap;
                     }

                    // Override input configs:
                    newPlayerMovement._isPitchInverted = shouldInvertPitch;
                    newPlayerMovement._isMouseFilteredToGameWindow = shouldFilterMousePos;
                    newPlayerMovement._stepValueToEaseRoll = stepFactorToEaseRoll;
                    newPlayerMovement._stepValueToEaseThrust = stepFactorToEaseThrust;
                    newPlayerMovement.HyperspeedBoostAmount = hyperspeedBoostFactor; // ADDING

                    // Copy over interface:
                    determinedSource = newPlayerMovement;
                    break;
                case InputType.EnemyAI:
                    // Null for now; placeholder...
                    determinedSource = null;
                    break;
                default:
                    throw new ArgumentOutOfRangeException(nameof(inputType), inputType, null);
            }

            return determinedSource;
        }
```

```
        #endregion
    }
}
```

Code 3. "ShipInputHandler.cs" script that returns an interface ship movement source, which currently instances the PlayerShipMovement class to be sent upstream to the "PlayerMovementInput.cs" component for centralizing the player's input processing.

```
/*-----------------------------------
 Custom script made by Brendan Sting
 Date: 7/21/2024
-----------------------------------*/

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Bsting.Ship.Managers;
using Bsting.Ship.Weapons;

namespace Bsting.Ship.Player
{
    /// <summary>
    /// Class that sets up the PlayerInputSystem with a GameManager and initializes the ship
controls to be used in ShipController.
    /// </summary>
    public class PlayerShipMovementInput : ShipMovementInput
    {
        public PlayerInputSystem _playerInputMap { get; private set; }

        [SerializeField] private List<Blaster> _listOfConnectedBlasters;
        [SerializeField] private bool _realisticJoyStickEnabled = true;
        [SerializeField] private bool _filterMouseMovement = false;
        [SerializeField] private float _factorToEaseShipRollBy = 1.0f;
        [SerializeField] private float _factorToEaseShipThrustBy = 1.0f;
        [SerializeField] private float _factorToBoostShipHyperspeedBy = 3.0f;

        private bool _hasGameManagerBeenInit = false;
        private bool _hasCameraManagerBeenInit = false;

        #region MonoBehaviors
        protected override void Awake()
        {
            base.Awake();

            // Player-specific config:
            _playerInputMap = new PlayerInputSystem();
            _playerInputMap.Enable();

            ConnectAllShipBlasters();

        }

        void OnEnable()
        {
            // Send to Game Manager:
            SubscribeGameManagersInputSystem();

            SubscribeCameraManagersInputSystem();
        }

        protected override void Start()
        {
```

```
            if (!_hasGameManagerBeenInit)
            {
                SubscribeGameManagersInputSystem();
            }

            if (!_hasCameraManagerBeenInit)
            {
                SubscribeCameraManagersInputSystem();
            }

            // Apply player input action map to player config:
            ShipControls = ShipInputHandler.GetInputControls(_inputTypeForThisShip,
                                                             playerInputMap: _playerInputMap,
                                                             shouldInvertPitch:
_realisticJoyStickEnabled,
                                                             shouldFilterMousePos:
_filterMouseMovement,
                                                             stepFactorToEaseRoll:
_factorToEaseShipRollBy,
                                                             stepFactorToEaseThrust:
_factorToEaseShipThrustBy,
                                                             hyperspeedBoostFactor:
_factorToBoostShipHyperspeedBy);
        }

        protected override void OnDestroy()
        {
            base.OnDestroy();
        }
        #endregion

        private void ConnectAllShipBlasters()
        {
            foreach (Blaster connectedBlaster in _listOfConnectedBlasters)
            {
                if (connectedBlaster != null)
                {
                    connectedBlaster.SetPlayerInputInstance(_playerInputMap);
                }
            }
        }

        private void SubscribeGameManagersInputSystem()
        {
            if (GameManager.Instance != null)
            {
                GameManager.Instance.SetPlayerInputInstance(_playerInputMap);
                _filterMouseMovement = GameManager.Instance.IsMouseIgnoredOutsideGameWindow();
                _hasGameManagerBeenInit = true;
            }
        }

        private void SubscribeCameraManagersInputSystem()
        {
            if (CameraManager.Instance != null)
            {
                CameraManager.Instance.SetPlayerInputInstance(_playerInputMap);
                _hasCameraManagerBeenInit = true;
            }
        }

        public void DisableAllBlasters()
        {
            foreach (Blaster connectedBlaster in _listOfConnectedBlasters)
            {
                connectedBlaster.enabled = false;
            }
        }
```

```
        public void EnableAllBlasters()
        {
            foreach (Blaster connectedBlaster in _listOfConnectedBlasters)
            {
                connectedBlaster.enabled = true;
            }
        }
    }
}
```

Code 4. "PlayerShipMovementInput.cs" component that is responsible for creating a new Player Input System from the Player Input Actions asset and then using that to construct new ship controls, specifically for the player's ship, which will be accessed occasionally by the "ShipController.cs".

```
/*-----------------------------------
 Custom script made by Brendan Sting
 Date: 7/21/2024
-----------------------------------*/

using Bsting.Ship.Player;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;
using Bsting.Ship.Managers;

namespace Bsting.Ship
{
    /// <summary>
    /// Class that mainly pilots the ship via applying forces to it, which are affected by a
sourced ShipMovementInput.
    /// </summary>
    public class ShipController : MonoBehaviour
    {
        // Fields:
        [Header("Source Input To Control Aircraft")]
        [SerializeField] private ShipMovementInput _shipMovementInput; // this may be
PlayerShipMovementInput (inherits)

        [Header("Aircraft Movement Values")]
        [SerializeField][Range(1000f, 10000f)] private float _thrustForce = 7500f;
        [SerializeField][Range(1000f, 10000f)] private float _yawForce = 2000f;
        [SerializeField][Range(1000f, 10000f)] private float _pitchForce = 6000f;
        [SerializeField][Range(1000f, 10000f)] private float _rollForce = 1000f;

        [Header("Aircraft Movement Affectors")]
        [SerializeField][Range(-1f, 1f)] private float _thrustFactor;
        [SerializeField][Range(-1f, 1f)] private float _yawFactor;
        [SerializeField][Range(-1f, 1f)] private float _pitchFactor;
        [SerializeField][Range(-1f, 1f)] private float _rollFactor;
        [SerializeField] private float _hyperspeedFactor;

        [Header("Ship Model Components")]
        [SerializeField] List<ShipEngine> _engines;

        [Header("Ship Abilities")]
        [SerializeField] private float _hyperSpeedDuration = 3.0f;
        [SerializeField] private float _hyperSpeedCooldownTime = 5.0f;
```

```
        public UnityEvent OnHyperspeedActivated = new UnityEvent(); // Use this to trigger
hyperspeed VFX
        public UnityEvent OnHyperspeedExpired = new UnityEvent(); // Use this to turn off
hyperspeed VFX

        private Rigidbody _thisRigidBody;
        private PlayerInputSystem _connectedInputMap = null;
        private PlayerShipMovementInput _initPlayerInputControls = null;
        private Coroutine _hyperspeedRoutine = null;
        private bool _canUseHyperspeed = true;
        private bool _isHyperspeedPressed = false;
        private float _amountOfHyperspeedRemaining;
        private float _timeLeftOnHyperspeedCooldown;
        private float _MAX_SPEED_FACTOR = 0f;
        private float _currentBoostSpeed = 0f;
        private float _currentHyperspeedMultiplier = 0f;
        private bool _hasHyperSpeedJumpstarted = false;
        private bool _hasLevelManagerBeenInit = false;

        IShipMovement SourcedInput => _shipMovementInput.ShipControls;

        #region MonoBehaviors
        void Awake()
        {
            _thisRigidBody = GetComponent<Rigidbody>();

            // ADDING:
            if (_shipMovementInput != null)
            {
                if (_shipMovementInput.GetType() == typeof(PlayerShipMovementInput))
                {
                    _initPlayerInputControls = _shipMovementInput as PlayerShipMovementInput;
                    _connectedInputMap = _initPlayerInputControls._playerInputMap;
                }
            }
        }

        void OnEnable()
        {
            SubscribeShipTransformToLevelManager();
        }

        void OnDisable()
        {
            if (_hyperspeedRoutine != null)
            {
                InterruptHyperspeedRoutine();
            }
        }

        void OnDestroy()
        {
            if (_hyperspeedRoutine != null)
            {
                InterruptHyperspeedRoutine();
            }
        }

        void Start()
        {
            if (!_hasLevelManagerBeenInit)
            {
                SubscribeShipTransformToLevelManager();
            }

            foreach (ShipEngine engine in _engines)
            {
                engine.Init(SourcedInput);
            }
```

```
            if (Mathf.Approximately(0f, _MAX_SPEED_FACTOR))
            {
                _MAX_SPEED_FACTOR = (_thrustForce * 1); // 1 is to represent a pressed Thrust input
value
            }
        }

        void Update()
        {
            // At the start of each frame, send updated transform info to Level Manager:
            LevelManager.Instance.SetPlayerShipTransform(GetTransformOfShip());

            // Get processed input values from interface to Input Actions:
            _thrustFactor = SourcedInput.ThrustFactorInput;
            _yawFactor = SourcedInput.YawFactorInput;
            _pitchFactor = SourcedInput.PitchFactorInput;
            _rollFactor = SourcedInput.RollFactorInput;
            _hyperspeedFactor = SourcedInput.HyperspeedFactorInput;

            if (_isHyperspeedPressed)
            {
                Debug.Log("Time left in Hyperspeed: " + _amountOfHyperspeedRemaining);
                _amountOfHyperspeedRemaining -= Time.deltaTime;
            }
            else if (!_canUseHyperspeed && !_isHyperspeedPressed)
            {
                Debug.Log("Cooldown time remaining on Hyperspeed: " +
_timeLeftOnHyperspeedCooldown);
                _timeLeftOnHyperspeedCooldown -= Time.deltaTime;
            }
        }

        void FixedUpdate()
        {
            // Physics-update YAW:
            if (!Mathf.Approximately(0f, _yawFactor))
            {
                // Affect Y-axis constant:
                _thisRigidBody.AddTorque(transform.up * (_yawForce * _yawFactor *
Time.fixedDeltaTime));
            }

            // Physics-update PITCH:
            if (!Mathf.Approximately(0f, _pitchFactor))
            {
                // Affect X-axis constant:
                _thisRigidBody.AddTorque(transform.right * (_pitchForce * _pitchFactor *
Time.fixedDeltaTime));
            }

            // Physics-update ROLL:
            if (!Mathf.Approximately(0f, _rollFactor))
            {
                // Affect Z-axis constant:
                _thisRigidBody.AddTorque(transform.forward * (_rollForce * _rollFactor *
Time.fixedDeltaTime));
            }

            // "If we can use hyperspeed ability, and it has not been pressed yet":
            if (_canUseHyperspeed && !_isHyperspeedPressed)
            {
                if ((_hyperspeedFactor > 1.0f) && (_hyperspeedRoutine == null)) // Hyperspeed input
is being pressed
                {
                    _currentHyperspeedMultiplier = _hyperspeedFactor;
                    _hyperspeedRoutine = StartCoroutine(UseHyperspeed()); // ADDING
                }
            }
```

```
            // Physics-update THRUST:
            if (Mathf.Approximately(0f, _thrustFactor) && _isHyperspeedPressed)
            {
                CheckIfHyperspeedHasBeenJumpstarted();
                _thisRigidBody.velocity = transform.forward * (_currentBoostSpeed *
_currentHyperspeedMultiplier * Time.fixedDeltaTime);
                CheckIfBoostedSpeedExceedsMaxSpeed();
            }
            else if (!_connectedInputMap.Player.AircraftThrust.IsPressed() && _isHyperspeedPressed)
            {
                CheckIfHyperspeedHasBeenJumpstarted();
                _thisRigidBody.velocity = transform.forward * (_currentBoostSpeed *
_currentHyperspeedMultiplier * Time.fixedDeltaTime);
                CheckIfBoostedSpeedExceedsMaxSpeed();
            }
            else if (!Mathf.Approximately(0f, _thrustFactor) && _isHyperspeedPressed)
            {
                CheckIfHyperspeedHasBeenJumpstarted();
                _currentBoostSpeed = _MAX_SPEED_FACTOR;
                _thisRigidBody.velocity = transform.forward * (_currentBoostSpeed *
_currentHyperspeedMultiplier * Time.fixedDeltaTime);
                CheckIfBoostedSpeedExceedsMaxSpeed();
            }
            else if (!Mathf.Approximately(0f, _thrustFactor) && !_isHyperspeedPressed)
            {
                _thisRigidBody.AddForce(transform.forward * (_thrustForce * _thrustFactor *
Time.fixedDeltaTime));
            }

        }
        #endregion

        #region Coroutine(s)
        IEnumerator UseHyperspeed()
        {
            ResetDurationAndCooldownBarValues();
            _canUseHyperspeed = false;
            _isHyperspeedPressed = true;
            if (_initPlayerInputControls != null) { _initPlayerInputControls.DisableAllBlasters();
} // Disable blasters while in boost mode
            OnHyperspeedActivated.Invoke();
            _currentBoostSpeed = (_thrustForce / 2);
            yield return new WaitForSeconds(_hyperSpeedDuration);

            _isHyperspeedPressed = false;
            if (_initPlayerInputControls != null) { _initPlayerInputControls.EnableAllBlasters(); }
            OnHyperspeedExpired.Invoke();
            Debug.Log("!!! Hyperspeed is out of fuel. Cooldown engaged. !!!");

            yield return new WaitForSeconds(_hyperSpeedCooldownTime); // Hand off control back to
main loop while cooldown runs

            // Reset vars:
            _hyperspeedRoutine = null;
            ResetDurationAndCooldownBarValues();
            _currentBoostSpeed = 0f;
            _currentHyperspeedMultiplier = 0f;
            _hasHyperSpeedJumpstarted = false;
            _canUseHyperspeed = true;
            Debug.Log("=== Cooldown expired. Hyperspeed is ready! ===");
        }
        #endregion

        #region Helper Function(s)
        private void ResetDurationAndCooldownBarValues()
        {
            _amountOfHyperspeedRemaining = _hyperSpeedDuration;
            _timeLeftOnHyperspeedCooldown = _hyperSpeedCooldownTime;
```

```
        }

        // Backup Function in case things go south later in development:
        private void MakeDirectionOfRigidBodyGoForward(Rigidbody usingThisRigidbody)
        {
            Vector3 currentVelocity = usingThisRigidbody.velocity;
            var localVelocity =
usingThisRigidbody.transform.InverseTransformDirection(currentVelocity);

            bool isRBMovingForward = (localVelocity.z > 0);

            if (!isRBMovingForward)
            {
                // Force/override velocity to go forward on RB:
                Vector3 normalizedLocalForwardVel =
usingThisRigidbody.transform.InverseTransformDirection(transform.forward);
                usingThisRigidbody.velocity =
usingThisRigidbody.transform.InverseTransformDirection(transform.forward);
            }
        }

        private void CheckIfBoostedSpeedExceedsMaxSpeed()
        {
            if ((_currentBoostSpeed > _MAX_SPEED_FACTOR) || (Mathf.Approximately(_MAX_SPEED_FACTOR,
_currentBoostSpeed)))
            {
                _currentBoostSpeed = _MAX_SPEED_FACTOR;
            }
            else
            {
                _currentBoostSpeed = Mathf.Lerp(_currentBoostSpeed, _MAX_SPEED_FACTOR,
Time.fixedDeltaTime);
            }
        }

        private void CheckIfHyperspeedHasBeenJumpstarted()
        {
            if (!_hasHyperSpeedJumpstarted)
            {
                _thisRigidBody.AddForce(transform.forward * (_currentBoostSpeed * _hyperspeedFactor
* Time.fixedDeltaTime));
                _hasHyperSpeedJumpstarted = true;
            }
        }

        private void SubscribeShipTransformToLevelManager()
        {
            // Send initial Transform to Level Manager:
            if (LevelManager.Instance != null)
            {
                LevelManager.Instance.SetPlayerShipTransform(GetTransformOfShip());
                _hasLevelManagerBeenInit = true;
            }
        }
        #endregion

        #region Public Accessor(s)
        public float GetRemainingHyperspeedAmount()
        {
            float currentAmtRemaining = _amountOfHyperspeedRemaining;
            return currentAmtRemaining;
        }

        public float GetTimeLeftOnHyperspeedCooldown()
        {
            float currentTimeLeft = _timeLeftOnHyperspeedCooldown;
            return currentTimeLeft;
        }
```

```
        public Vector3 GetPositionOfShip()
        {
            return this.gameObject.transform.position;
        }

        public Transform GetTransformOfShip()
        {
            return this.gameObject.transform;
        }
        #endregion

        public void SetPositionOfShip(Vector3 newPos)
        {
            this.gameObject.transform.position = newPos;
        }

        public void InterruptHyperspeedRoutine()
        {
            StopCoroutine(_hyperspeedRoutine);
            _hyperspeedRoutine = null;
        }
    }
}
```

Code 5. The player ship's top-level "ShipController.cs" script component that accesses the processed source player input values and translates those values as factors to affect the physics of the player's ship in Unity's Fixed Update method.

Note that if you're wondering as to where the ship controls that the "PlayerShipMovementInput.cs" script constructs and "ShipController.cs" accesses is declared, it can be found inside one of the refactored base classes, "ShipMovementInput.cs", whose base code is also given below.

```
/*-----------------------------------
 Custom script made by Brendan Sting
 Date: 7/21/2024
-----------------------------------*/

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

namespace Bsting.Ship
{
    /// <summary>
    /// Base class to hook up a ShipControls (part of IShipMovement interface) property with a
provided ship input type.
    /// </summary>
    public class ShipMovementInput : MonoBehaviour
    {
        // Properties:
        public IShipMovement ShipControls { get; protected set; }

        // Fields:
```

```
        // (Default is Player type)
        [SerializeField] protected ShipInputHandler.InputType _inputTypeForThisShip =
ShipInputHandler.InputType.Player;

        #region MonoBehaviors
        protected virtual void Awake()
        {
            // ...
        }

        // Start is called before the first frame update
        protected virtual void Start()
        {
            ShipControls = ShipInputHandler.GetInputControls(_inputTypeForThisShip);
        }

        protected virtual void OnDestroy()
        {
            ShipControls = null;
        }
        #endregion
    }
}
```

Code 6. The base ShipMovementInput class that generalizes the setup framework for initializing ship controls for various types of spaceships.

After the foundational player input management scripts were laid out, it was time to hookup the main input configuration scripts to the Player's ship prefab (which was created by dragging and dropping the current model into Unity's Project window). This would allow for design-level adjustments to be made to the look and feel of the player ship controls without having to go into the backend of the player's controls all that much. The first config script to expose on the player ship asset was the ShipController class and this was placed on the root parent object of the prefab since it contained all the main physics-related values and properties, shown below in Figure 7.

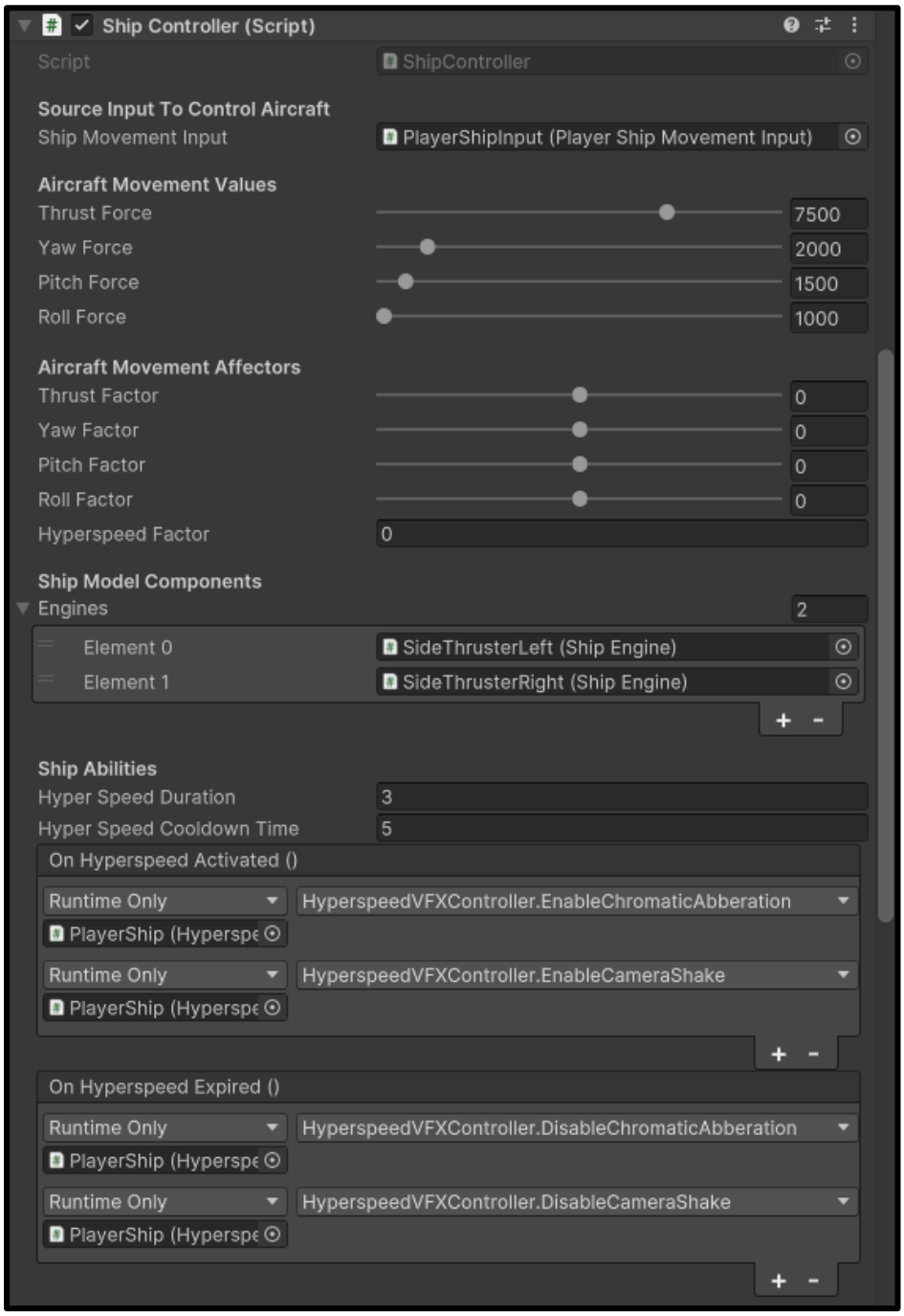

Figure 7. ShipController's configurable serialized fields and properties that affect the physics of the player's ship asset.

Lastly for the player controls and asset, was hooking up the second config script on the player ship prefab which was the PlayerShipMovement class. This script was put on a child

object of the player's ship, named "PlayerShipInput", and it allows for filters and easing to be applied to the processed input being handed to the ShipController. Currently, its customized to produce eased values for smoother animations on the player's cockpit components, like the joystick or throttle handle. Its exposed fields and Inspector layout can be seen in the following figure.

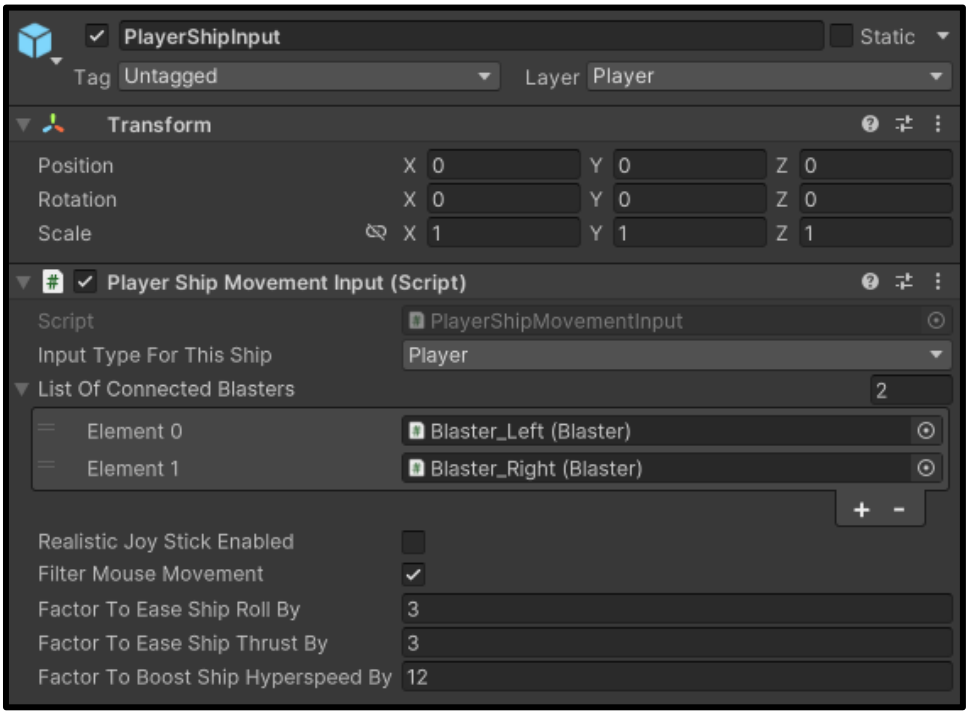

Figure 8. PlayerShipMovementInput's exposed fields and properties which allow to mainly customize processed ship inputs before they are handed off to the ship controller.

It's worth noting that some advanced mechanics have already worked their way onto PlayerShipMovementInput's script, such as support for communicating the initialized input actions map to the ship's blasters (to sync up detection when the player presses FIRE) or adjusting hyper speed's boost velocity.

Before moving on to any player abilities, we'll also examine the current progress on the player's camera and its associated setup on the player's ship asset.

Player Camera

To achieve the reminiscent style of the 1979 "Asteroids" top-down arcade game while also incorporating a new first-person view mechanic, a special camera setup is required to merge both of these effects into one game. Breaking down the problem, it was decided that one camera should be responsible for displaying the first-person perspective in-game, while a second camera would be responsible for displaying the top-down third-person perspective in "Asteroid Belt". To allow the player to access these two separate cameras, a toggle mechanic was designed where the player would press an input, say 'Q' on the keyboard, and then the game would switch to the other camera as the currently active in-game view. Events could also be added later on and listening to this "toggle camera" event which would allow for some cool post-processing effects and camera control. To support this ambitious camera system, Unity's "Cinemachine" package was chosen to help better foster this setup and make the configuration process of the player camera a lot smoother. Plus, "Cinemachine" came with built-in camera activated events which made it a whole lot easier to listen to the toggle event in the scripting backend.

Implementation-wise, after the "Cinemachine" package was imported into the "Asteroid Belt" Unity project, the sample scene's initial main camera was converted into a Cinemachine Brain camera, via adding a Cinemachine Brain component in the Inspector. This would allow the main camera to act as a bridge for selecting which child/virtual camera to render on screen. See in the below figure for how the current main camera is configured with Cinemachine Brain.

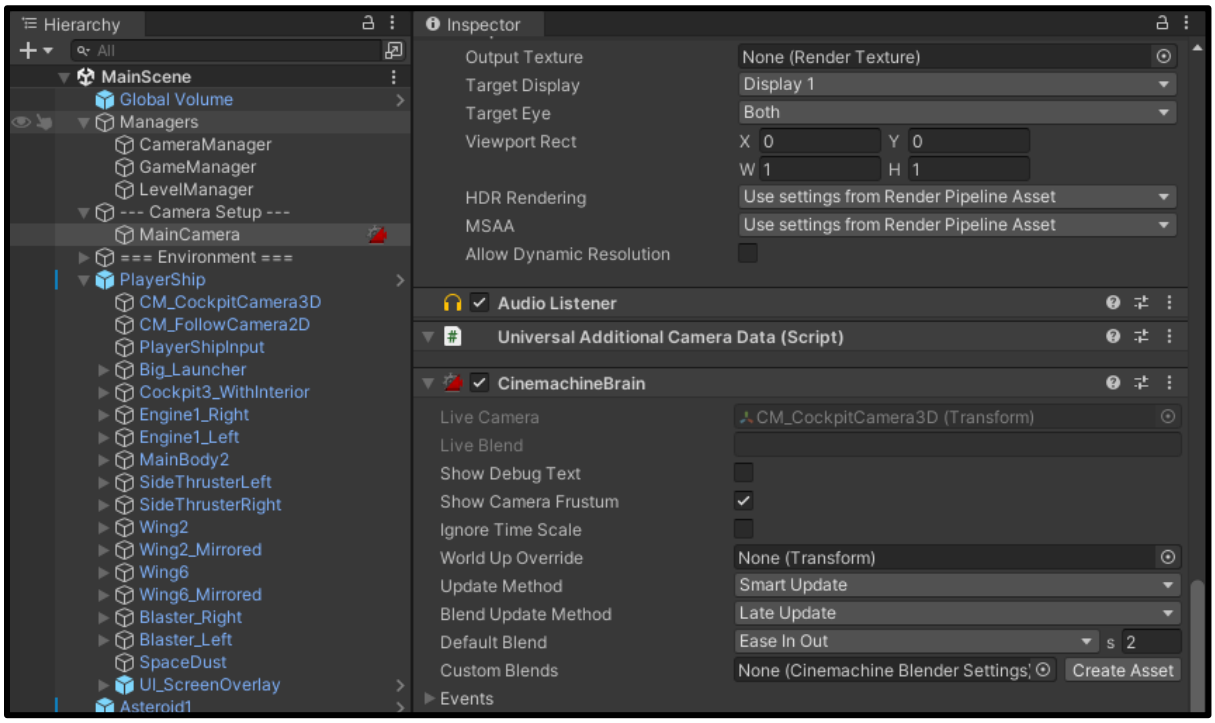

Figure 9. New main camera setup that now uses a Cinemachine Brain component.

You may have noticed in the above Figure 9 that there's also two child game objects called "CM_CockpitCamera3D" and "CM_FollowCamera2D" within the Player Ship prefab. These two objects are the two separate first-person and third-person cameras discussed earlier, with the cockpit camera being an alias for the first-person view, and the follow camera being a name for the third-person top-down perspective. Further, these two child cameras are Cinemachine Virtual Cameras which just represent different virtual viewpoints for the main camera to choose from. Note that the follow camera isn't actually in 2D despite what its object name suggests, it's just labeled 2D as it uses an orthographic-like transposer body and top-down follow constraints that gives its rendering a 2D look. Both are under the Player Ship prefab so that their position can be consistently updated with the ship's, and this removes the need for camera damping/movement compensation. You can view both of these two virtual cameras and their properties in the following figures.

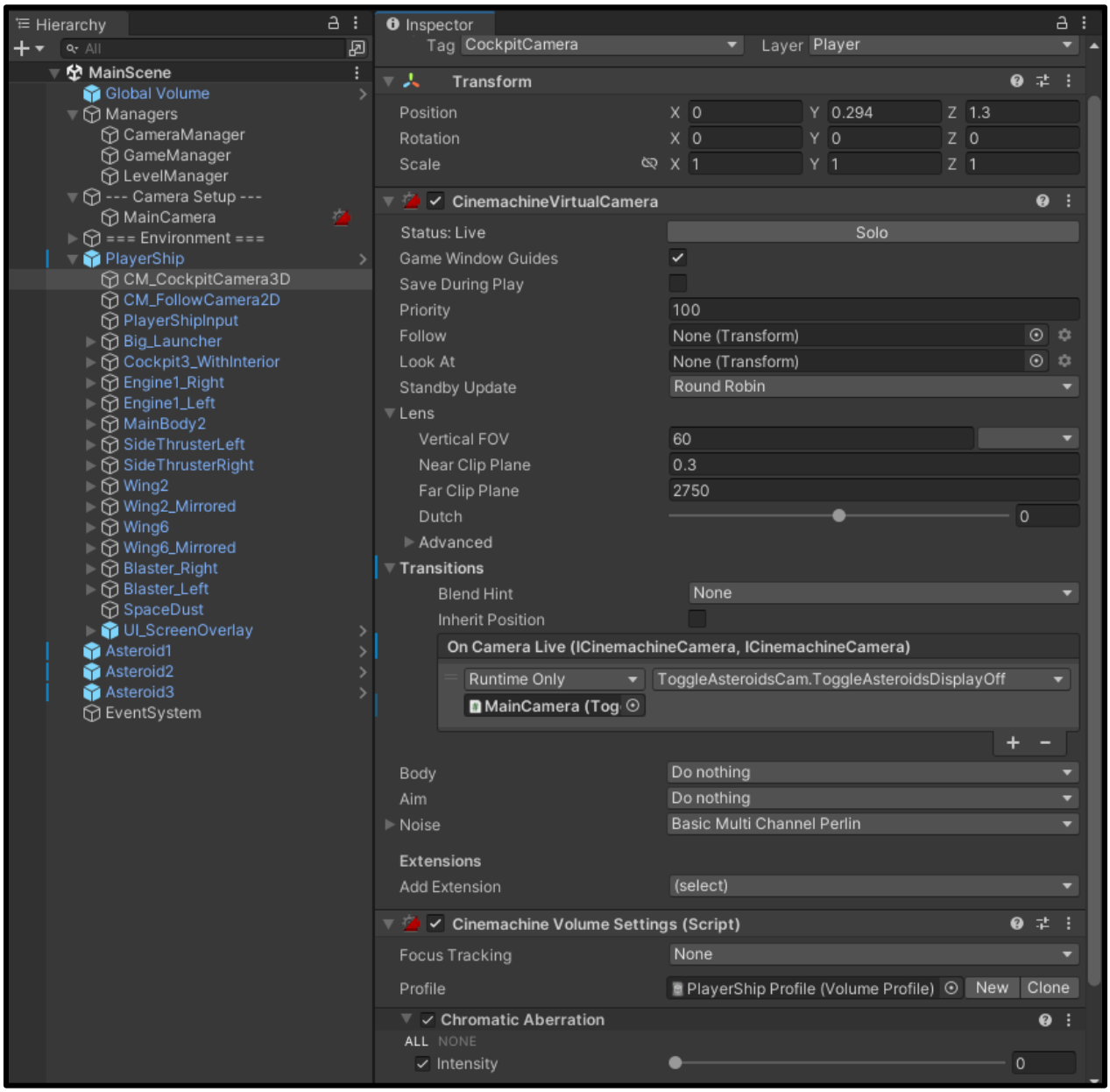

Figure 10. Inspector view of the first-person virtual camera, “CM_CockpitCamera3D”.

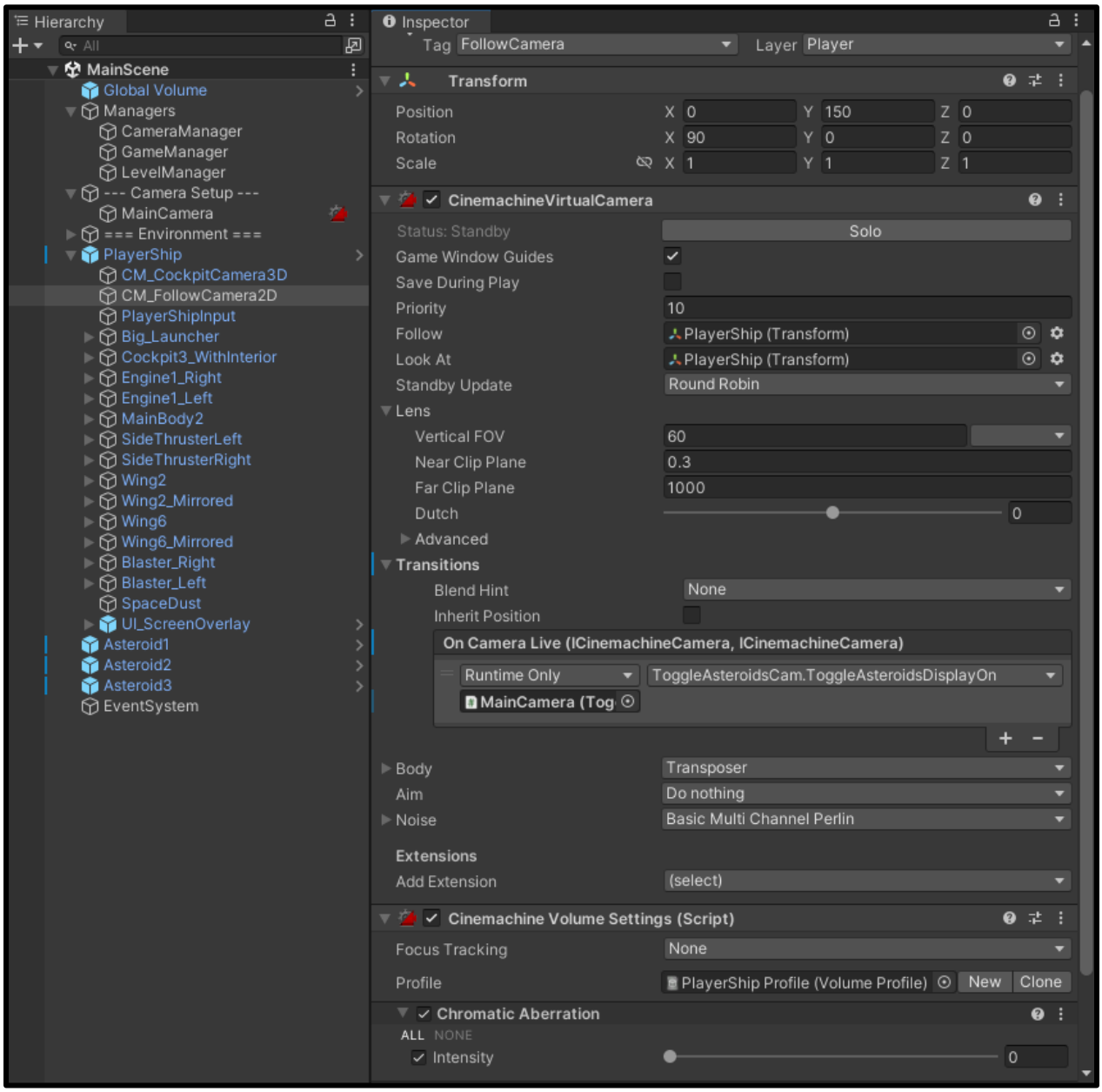

Figure 11. Inspector view of the top-down virtual camera, “CM_FollowCamera2D”.

You may have noticed that in Figure 11, the follow camera has a lot more transform data to follow and look at than the first-person camera, which is mainly due to the need to keep the top-down camera steady and stable since it views the ship at a relatively far distance. Additionally, you might’ve also seen the “Chromatic Aberration” and “Basic Multi Channel

Perlin" components added on to the virtual cameras. These are post-processing camera effects mainly used for simulating a hyper speed ability (which we'll look into more in the player abilities section) but are all currently set to their zero values by default.

With all the virtual cameras independently set up in the scene, a camera manager was then needed to coordinate which virtual camera should currently be active and sent to the main camera to be rendered on screen. This is the basis for the "CameraManager" component you may have observed in the past figures under the Managers group in the Unity Hierarchy. Its script would basically take in a list of virtual cameras to toggle through, and when prompted by player input in the backend, would go and determine which camera is next in line to index to and activate. The current CameraManager script described is given in the below code snippet.

```
/*-----------------------------------
 Custom script made by Brendan Sting
 Date: 7/27/2024
-----------------------------------*/

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

namespace Bsting.Ship.Managers
{
    /// <summary>
    /// A camera manager that goes through a list of
    /// virtual cameras and decides which one to activate.
    /// </summary>
    public class CameraManager : Manager<CameraManager>, IVirtualCameras
    {
        [Header("List Of Virtual Cameras To Activate")]
        [SerializeField] private List<GameObject> _virtualCameras;

        private PlayerInputSystem _currentPlayerInputSystem;
        private int _virtualCamerasIndex = -1;

        public VirtualCameras CameraKeyPressed
        {
            get
            {
                if (_virtualCameras != null)
                {
                    if (_virtualCameras.Count > 0)
                    {
                        if (_virtualCamerasIndex == -1) { _virtualCamerasIndex = 0; } // Default
starting index

                        if (_currentPlayerInputSystem != null
                            &&
                            _currentPlayerInputSystem.Player.ToggleCamera.WasPressedThisFrame())
```

```
                        {
                            ToggleThruCameraIndices(_virtualCameras.Count - 1);
                        }
                    }
                }

                return (VirtualCameras)_virtualCamerasIndex;
            }
        }

        protected override void Awake()
        {
            base.Awake();
        }


        // Start is called before the first frame update
        void Start()
        {
            // Set CockpitCamera to be active camera by startup:
            SetActiveCamera(VirtualCameras.CockpitCamera);
        }

        // Update is called once per frame
        void Update()
        {
            VirtualCameras toggledCam = CameraKeyPressed;
            SetActiveCamera(toggledCam);
        }

        private void SetActiveCamera(VirtualCameras newActiveCamera)
        {
            if (newActiveCamera == VirtualCameras.NoCamera)
            {
                Debug.LogWarning("No camera available to toggle to.");
                return;
            }

            foreach (GameObject camera in _virtualCameras)
            {
                string strNewCamera = newActiveCamera.ToString();
                camera.SetActive(camera.CompareTag(strNewCamera));
            }
        }

        private void ToggleThruCameraIndices(int tailIndex)
        {
            if (_virtualCamerasIndex < tailIndex)
            {
                _virtualCamerasIndex++;
            }
            else
            {
                _virtualCamerasIndex = VirtualCameras.CockpitCamera.GetHashCode();
            }
        }

        public void SetPlayerInputInstance(PlayerInputSystem newInputInstance)
        {
            _currentPlayerInputSystem = newInputInstance;
        }

        public List<GameObject> GetListOfManagedCameras()
        {
            List<GameObject> managedCameras = _virtualCameras;
            return managedCameras;
        }
    }
}
```

Code 7. The current backend of CameraManager.cs which rotates through a list of virtual cameras and determines which one should be active to render the current scene.

Looking closer at the above code snippet of CameraManager, you'll see a lot of references to a custom type called "VirtualCameras". This from CameraManager's IVirtualCameras interface it implements as seen at the top of its script. The IVirtualCameras basically declares and sets up an enumeration group of virtual camera indices along with a camera pressed property to implement with the Player Input Actions asset. All this does is help make a template for CameraManager as well as also making its script easier to read. We can view how simple the code behind IVirtualCameras is in its code provided below.

```
/*-----------------------------------
 Custom script made by Brendan Sting
 Date: 7/27/2024
-----------------------------------*/

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

namespace Bsting.Ship
{
    public enum VirtualCameras
    {
        NoCamera = -1,
        CockpitCamera = 0,
        FollowCamera = 1,
    }

    /// <summary>
    /// Base camera interface to declare the input toggle
    /// property as well as the mapping of vcam indices in an enum.
    /// </summary>
    public interface IVirtualCameras
    {
        VirtualCameras CameraKeyPressed
        {
            get;
        }
    }
}
```

Code 8. The interface code from IVirtualCameras that's referenced in CameraManager's script.

After scripting up CameraManager, its resulting Inspector panel is setup to the following in the below figure so it may reference all current virtual cameras in the Unity scene.

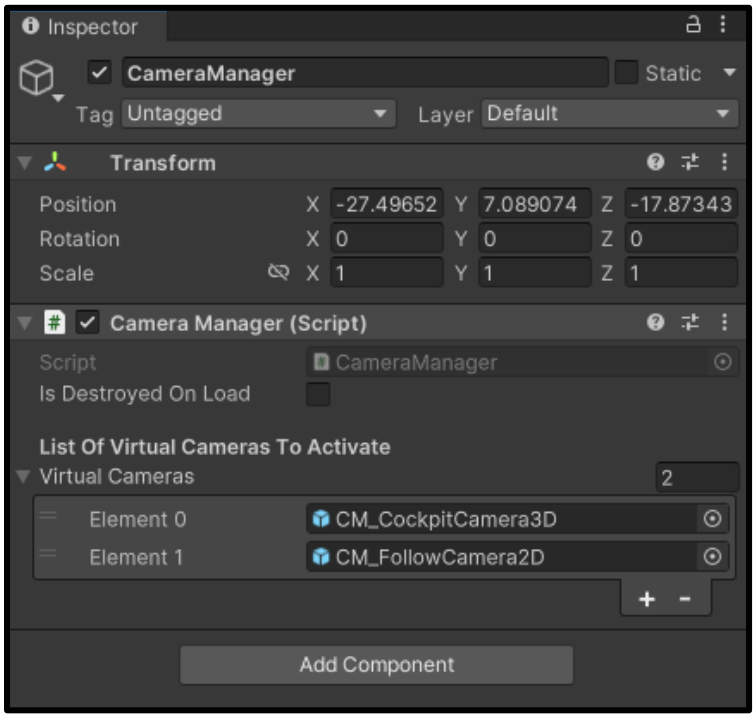

Figure 12. The current Inspector values applied to CameraManager's script component.

We now have both the Cinemachine camera objects and base scripts to alternate between two different camera views in our game's scene. Next up was just positioning them correctly in the scene and storing their relative positions as saved overrides on the Player Ship prefab. For the first-person camera, it had to be far up enough in front of the pilot seat, but low enough so it wouldn't clip through the glass material. As for the third-person top-down virtual camera, it had to be positioned high enough to make the ship seem of similar size to the one in "Asteroids", while also being centered and rotated correctly. The resulting views from positioning both of these virtual cameras can be visualized in the next figure.

Figure 13. Side-by-side visualization of the two player cameras, with one being posed in FP and the other being configured and positioned top-down in a TP view.

While these two views are just what we want in “Asteroid Belt”, the third-person one looks a little bland and not too reminiscent of the original “Asteroids” graphics style. So, taking advantage of the fact that this project is made in the Universal Render Pipeline (where a lot of shaders can be advanced), a Fullscreen Shader that highlights the edges and outlines of pixels was made to give the top-down camera an exclusive “Asteroids” look. This shader was developed by following a short article to full screen outline shaders, titled “Outline Post Process in Unity Shader Graph (URP)- Fun With Fullscreen Graphs” which was written by Daniel Ilett. The great thing about a full screen shader here is that you can save a lot of time by not needing to create and apply a bunch of individual materials from one shader, and instead, you can just create one material and assign it to the post-processing of your camera(s) for a universal effect/style. After going through Daniel’s guide, the resulting shader graph for the top-down camera can be seen in the below visual.

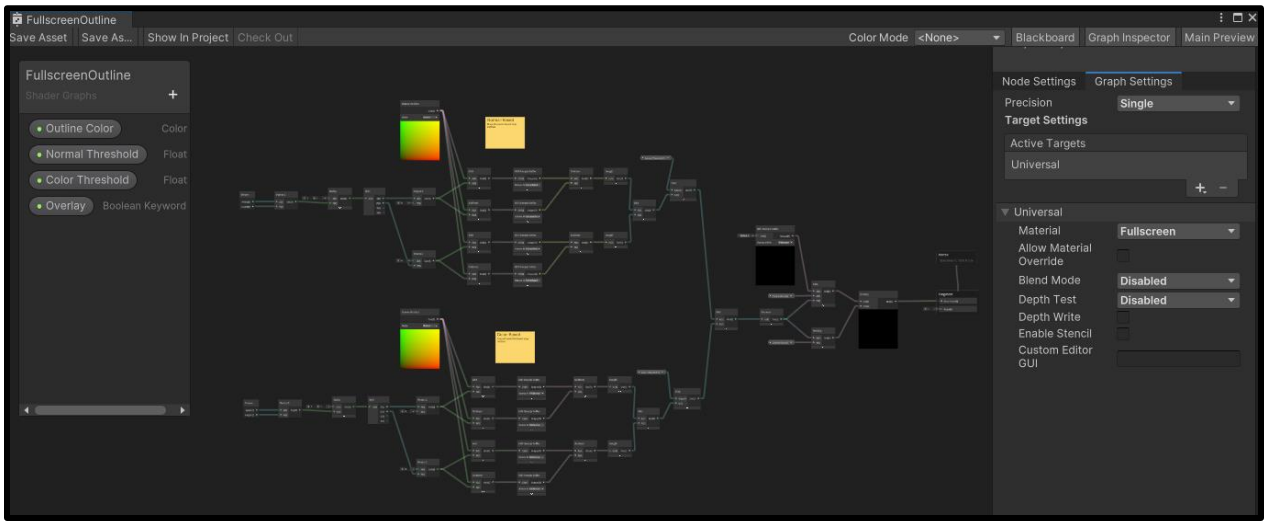

Figure 14. Abstract overview of the Full Screen Outline shader made for the top-down camera.

Least to say the outline shader's graph is a little complicated, but all you need to really know is that it uses screen position vectors and URP sample buffers to calculate the distance between each bordering pixel and shade those pixels and edges in accordingly to the set properties on its material.

Speaking of the outline shader's material, the next step was to create its material via right-clicking on the shader in the project window and going to > "Create" > select "Material". Then, the material was set to have that white outline we see in "Asteroids" and its overlay flag set to false so we can just have the outline rendered in the top-down view. To fully visualize the changes, refer to the below figure where its final properties are set in the full screen outline material's Inspector.

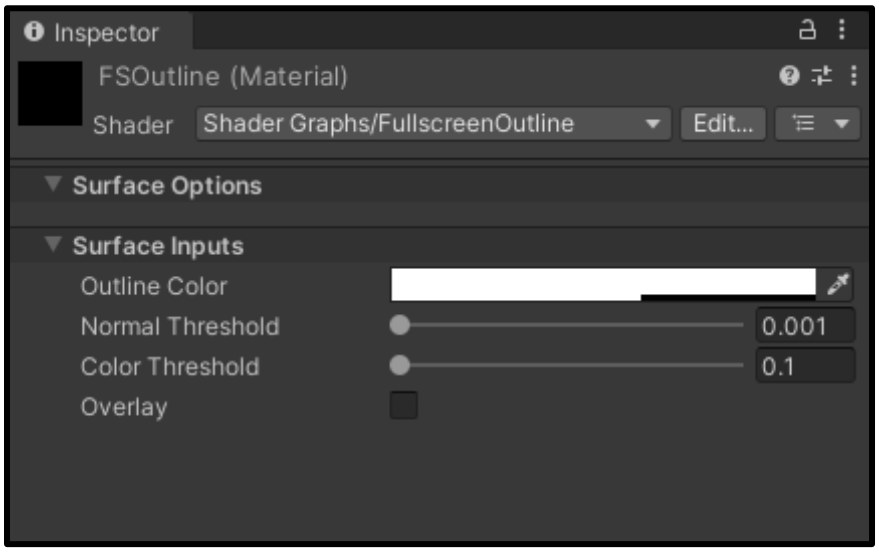

Figure 15. Final material properties set currently for the full screen outline shader.

All that's left to do to get the shader work on the entire screen is to add a Full Screen Pass Renderer Feature and then assign the newly made outlines material shader to its "Pass Material". Then, by keeping it disabled by default, we can then enable it via script whenever we change to the top-down camera and re-disable it when we are back to the first-person one. To add this feature to URP setup of "Asteroid Belt", we just have to locate the currently used renderer URP asset inside "Assets" > "Settings" folder. Once found, we just scroll down its Inspector and click on the "Add Renderer Feature" and add it in as seen in the following visual below.

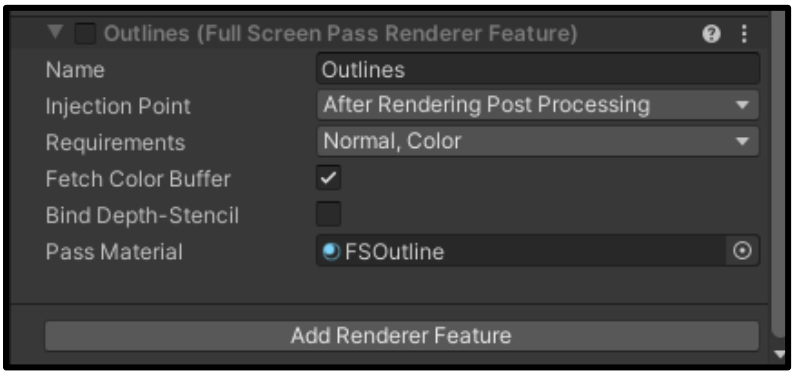

Figure 16. Inspector visual on adding Outlines Full Screen Pass Renderer Feature to URP asset.

The final step to achieving the desired “Asteroids” look we want in the top-down camera is to create a toggle script and attach that to one of our camera objects in the scene. For now, we’ll select the main camera for testing purposes, but it’s worth noting that it may be better to have the toggle script on the top-down “CM_FollowCamera2D” object under the Player Ship prefab since all the main object fields are a child game object of the ship currently. Regardless, here’s the code in the currently developed toggle camera script, called “ToggleAsteroidsCam.cs”.

```
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.Rendering;
using UnityEngine.Rendering.Universal;

namespace Bsting.Ship.Player.Camera
{
    public class ToggleAsteroidsCam : MonoBehaviour
    {

        public UniversalRenderPipelineAsset pipelineAsset;
        private FullScreenPassRendererFeature _outlineFullscreenShader; // Toggle this ON when in
the air
        [SerializeField] private GameObject _spaceDustDome; //Toggle this OFF when in the air

        void Start()
        {
            pipelineAsset.GetRenderer(0);

            // Access the Renderer Data using Reflection
            var renderer = (GraphicsSettings.currentRenderPipeline as
UniversalRenderPipelineAsset).GetRenderer(0);
            var property = typeof(ScriptableRenderer).GetProperty("rendererFeatures",
BindingFlags.NonPublic | BindingFlags.Instance);
            List<ScriptableRendererFeature> features = property.GetValue(renderer) as
List<ScriptableRendererFeature>;

            // Find the FullScreenPassRendererFeature
            foreach (var feature in features)
            {
                if (feature is FullScreenPassRendererFeature)
                {
                    _outlineFullscreenShader = (FullScreenPassRendererFeature)feature;
                    break;
                }
            }
        }

        void OnApplicationQuit()
        {
            ToggleAsteroidsDisplayOff();
        }

        public void ToggleAsteroidsDisplayOn()
        {
            TryToTurnOnOutline();
            TryToTurnOffSpaceDust();
```

```
        }

        public void ToggleAsteroidsDisplayOff()
        {
            TryToTurnOffOutline();
            TryToTurnOnSpaceDust();
        }

        private void TryToTurnOnOutline()
        {
            if (_outlineFullscreenShader != null)
            {
                _outlineFullscreenShader.SetActive(true);
            }
        }

        private void TryToTurnOffOutline()
        {
            if (_outlineFullscreenShader != null)
            {
                _outlineFullscreenShader.SetActive(false);
            }
        }

        private void TryToTurnOnSpaceDust()
        {
            if (_spaceDustDome != null)
            {
                _spaceDustDome.SetActive(true);
            }
        }

        private void TryToTurnOffSpaceDust()
        {
            if (_spaceDustDome != null)
            {
                _spaceDustDome.SetActive(false);
            }
        }
    }
}
```

Code 9. The "ToggleAsteroidsCam.cs" script that enables the outline shader and disables any other component (like the space dust) that's not supposed to be rendered while in this mode.

After all is said and done with the outline shader, and it compiles correctly, we get the new following display from our top-down camera when we switch to it in-game (using 'Q'):

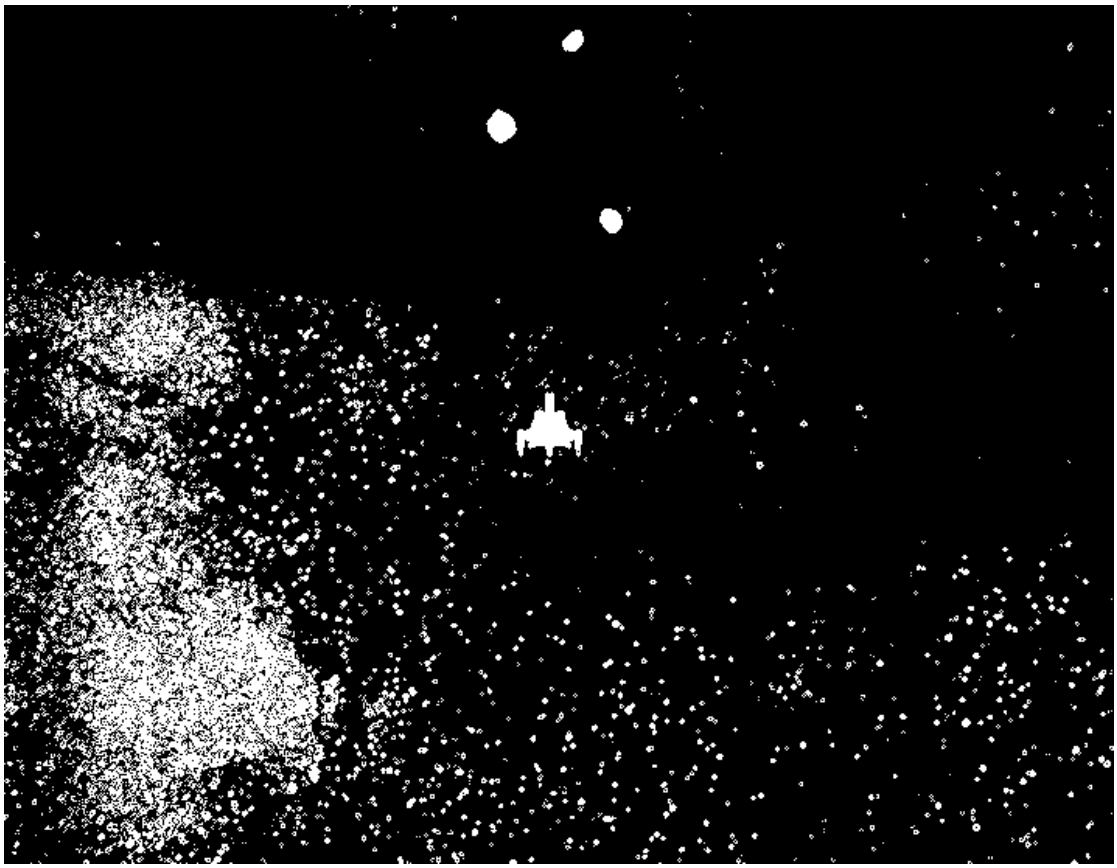

Figure 17. The shaded version of "Asteroid Belt's" top-down follow camera.

Player Abilities

As hinted at earlier in the discussion about post-processing effects applied to the cameras, there's really only one player "ability" currently in the project, "Asteroid Belt": *hyper-speed*!

Hyper-speed is one of the chosen advanced mechanics to add into the game and is a triggered ability, where once activated, will launch the player in the current direction at "higher than normal speeds". This could be used by the player to evade enemies, asteroids, or just as fun tactical maneuver. However, it will have a certain duration or fuel timer that will trigger a cooldown once the ship runs out of juice, so to speak. After the cooldown time period is up, the player should then be able to trigger their hyper-speed again!

With the project's design of the hyper-speed mechanic in mind, let's get in as to how it was developed. First, the base mechanic had to be implemented to where the player's ship would just sling-shot in a certain direction upon pressing the hyper-speed input action key, which is set to Q currently. This was added by modifying both the "PlayerShipMovementInput.cs" and "ShipController.cs" scripts mentioned in past code snippets. The main addition to these previous scripts is the hyper-speed coroutine which directs the entire ability in the background and can be seen in the following snippet (taken from ShipController's script).

```
// ...
#region Coroutine(s)
        IEnumerator UseHyperspeed()
        {
            ResetDurationAndCooldownBarValues();
            _canUseHyperspeed = false;
            _isHyperspeedPressed = true;
            if (_initPlayerInputControls != null) { _initPlayerInputControls.DisableAllBlasters();
} // Disable blasters while in boost mode
            OnHyperspeedActivated.Invoke();
            _currentBoostSpeed = (_thrustForce / 2);
            yield return new WaitForSeconds(_hyperSpeedDuration);

            _isHyperspeedPressed = false;
            if (_initPlayerInputControls != null) { _initPlayerInputControls.EnableAllBlasters(); }
            OnHyperspeedExpired.Invoke();
            Debug.Log("!!! Hyperspeed is out of fuel. Cooldown engaged. !!!");

            yield return new WaitForSeconds(_hyperSpeedCooldownTime); // Hand off control back to
main loop while cooldown runs

            // Reset vars:
            _hyperspeedRoutine = null;
            ResetDurationAndCooldownBarValues();
            _currentBoostSpeed = 0f;
            _currentHyperspeedMultiplier = 0f;
            _hasHyperSpeedJumpstarted = false;
            _canUseHyperspeed = true;
            Debug.Log("=== Cooldown expired. Hyperspeed is ready! ===");
        }
#endregion


// ...
```

Code 10. Code snippet taken from ShipController.cs that showcases the hyper-speed routine.

In the above code, the hyper-speed coroutine will essentially coordinate Boolean flags to declare when the speed should be increased and for how long using the WaitForSeconds() time counters. Additionally, some game components, like the blasters on the ship, are disabled in this

routine to prevent weird movement with Unity's LookAt method that especially gets weird under high velocity conditions like hyper-speed. Then, once the ability is over, flags are reset, and player ship components are re-enabled back to normal function.

In addition to the scripting modifications made to support the hyper-speed input in the player's controls, another separate script was developed to coordinate the post-processing effects discussed earlier with the cameras. This script, called "HyperspeedVFXController.cs", is attached to the Player Ship prefab, and using a list of virtual camera component references on the child cameras, will go through and enable each one of the post-processing overrides (like "Chromatic Aberration") when its appropriate methods are called. To hookup these calls to the ability's actual mechanic, they are thus referenced inside an OnHyperspeedActivated() and OnHyperspeedExpired() Unity Event that are declared and invoked inside "ShipController.cs", as seen in *Code 10* snippet above with the hyper-speed coroutine. Having everything hooked up and polished, here's what the final hyper-speed ability and effect looks like currently:

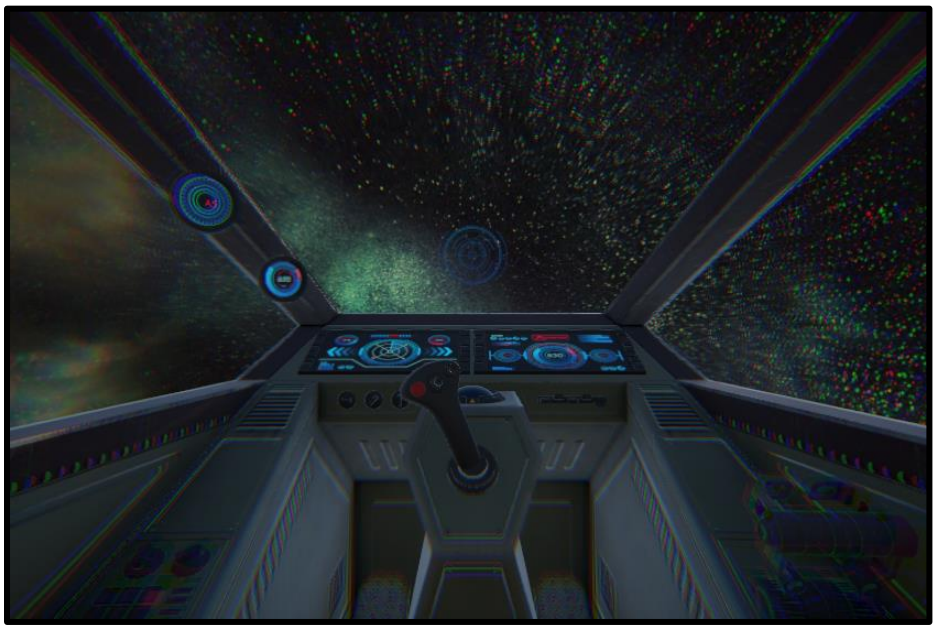

Figure 18. Visual of current hyper-speed ability, with its post-processing effects applied.

Environment

Moving outside of the player's scope and going into what was developed for the game environment, the first element to be added for the space surroundings was the skybox system. Again, going back to Midnite Oil Software's *How To Make a 3D Space Shooter Game in Unity – Tutorial* video, Greg (who runs Midnite Oil Software) recommended using Pulsar Bytes' "SpaceSkies Free" asset store package as it came with different resolutions and three different variants of the nebula-like skybox. After importing the skyboxes using Unity's Package Manager, a helper skybox handler script was made to select and apply one of the skyboxes to the current scene. It's code can be seen below.

```
/*-------------------------------------
 Custom script made by Brendan Sting
 Date: 7/21/2024
-------------------------------------*/

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

namespace Bsting.Skybox
{
    using Skybox = UnityEngine.Skybox;

    /// <summary>
    /// Class that handles assigning an overriding skybox to a URP camera.
    /// </summary>
    [RequireComponent(typeof(Skybox))] // Will add comp. type if not assigned
    public class SkyboxHandler : MonoBehaviour
    {
        // Serialized fields:
        [SerializeField] private List<Material> _skyboxMaterials;
        [SerializeField] private int _defaultMaterialIndex = 0;

        // Private var's:
        private Skybox _skybox;

        #region MonoBehaviors
        void Awake()
        {
            _skybox = GetComponent<Skybox>();
        }

        void OnEnable()
        {
            ChangeSkybox(_defaultMaterialIndex);
        }
        #endregion

        #region Helper Function(s)
        private void ChangeSkybox(int skyboxIndex)
```

```
        {
            if ((_skybox != null) && (skyboxIndex >= 0) && (skyboxIndex <= _skyboxMaterials.Count))
            {
                _skybox.material = _skyboxMaterials[skyboxIndex];
            }
        }
        #endregion
    }
}
```

Code 11. SkyboxHandler script the selects the skybox for the scene based on the passed in field values in the Inspector. Also adds a Skybox override component to render the chosen skybox on its attached camera if it isn't there already.

Once the skybox handler script was written out, it was attached with a "Skybox" component to the main camera so the skybox override would render for all subsequent virtual cameras connected to the Cinemachine brain. These new components on the main camera object can be seen initialized in the below Inspector figure.

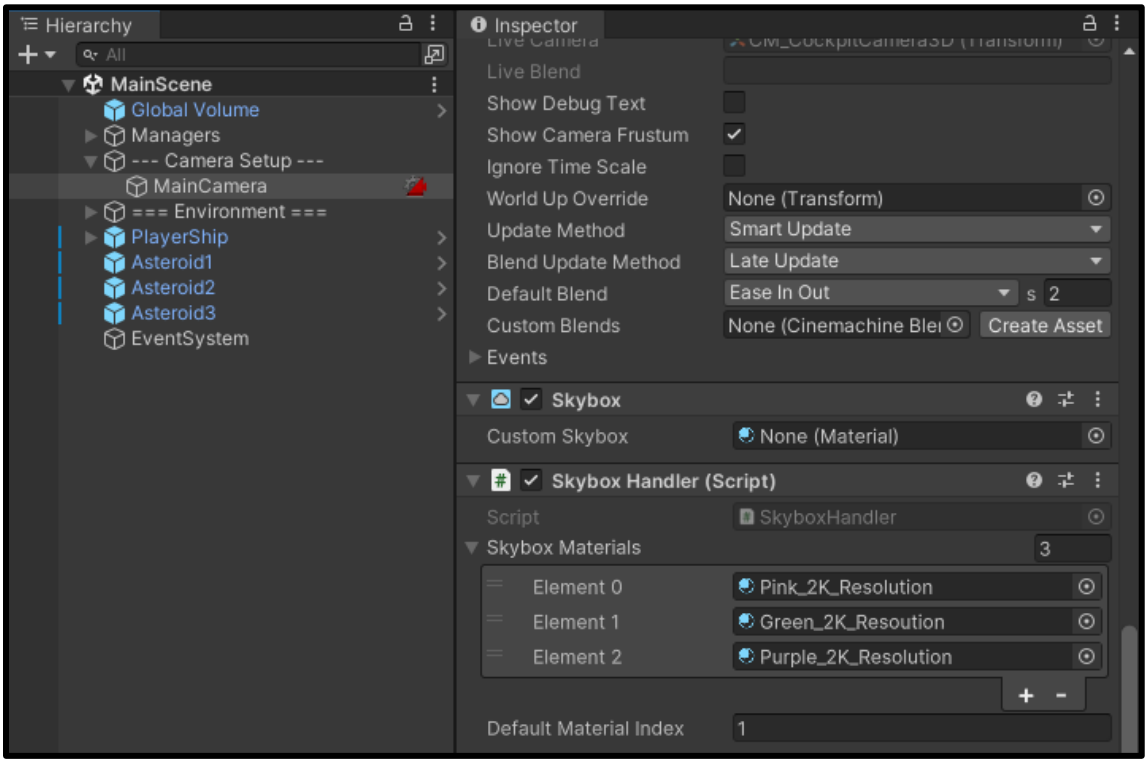

Figure 19. Main Camera's Inspector panel with the skybox components added.

The second main addition to the environment of "Asteroid Belt" was the immersive "space dust dome" element. This idea was taken from Midnite Oil Software's *How To Make a 3D Space Shooter Game in Unity – Tutorial* video, and the premise was to surround the player in particles that they could fly through to add on to the appearance of the nebula skybox. The dome of space particles was implemented via putting a child particle system under the Player prefab, which would make the space dust spawn as the player moves through it, creating an infinite space dust traversal effect. The resulting particle space dust dome can be seen below.

Figure 20. Visualization of the "space dust dome" to enhance the immediate environment.

Moving on to the third main addition to the project's environment, which affects the player's experience the most, is the playable area treadmill. A playable area treadmill is essentially a zone for the player's asset/controller to be confined in the scene in order to avoid reaching floating point imprecision with the player's transform data. There are two main approaches to this player treadmill: one is to surround them in some sort of border, that when crossed, would teleport them back inside, while the other is to have all player inputs control the

player's surrounding or generated terrain to give the illusion that the player is moving, when really, they are not. The former of these two options was chosen for the project's playable area treadmill since a lot of the player's inputs and controller have already been built out and configured to move the player's actual spaceship. However, if time allows later on in development, the latter option may be revisited if the end results are undesirable, and time still remains to make the necessary new input setup. Nonetheless, the initial design of the current teleport treadmill system was defined to be a sphere-like playable area that would reset the player back to the opposite edge of the treadmill sphere. Later on, it was determined in development that teleporting to the center would have better results since teleporting to the edge could lead to back-to-back teleports with the boundary still being relatively close to the player. To implement this teleport sphere treadmill, a radius and the player's vector distance from a center point are defined. Now, if their vector distance ever exceeds the radius' length, then the teleport will trigger. Below is the current code for the player's teleport treadmill, managed under the Level Manager game object.

```
using Bsting.Ship.Managers;
using Cinemachine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

namespace Bsting.Ship.Managers
{
    public class LevelManager : Manager<LevelManager>
    {
        // Fields:

        // Feature: Treadmill will be a sphere that will reset player back facing inside the
playable zone
        // Teleport - should warp player to opposite edge but facing inside normal direction to
playable field
        // NEW Teleport - should warp player to center of treadmill sphere but facing normal
direction

        [Header("Spherical Teleport-Based Treadmill Settings")]
        [SerializeField] private Vector3 _sphereCenterPosition = Vector3.zero;
        // Unity Forum posts says most game devs keep run distance under/at 10,000 units (> 100,000
= scuffed):
        [SerializeField][Range(100f, 10000f)] private float _sphereRadius = 100.0f;
```

```
        [SerializeField][Tooltip("If unchecked, then Player will spawn at the center of the
treadmill sphere.")]
        private bool _isPlayerTeleportedToEdgeOfMap = true;
        [SerializeField] public UnityEvent OnTeleportPlayerReady = new UnityEvent();
        [SerializeField] public UnityEvent OnPlayerTeleported = new UnityEvent();
        [SerializeField] public UnityEvent OnPlayerOutOfTeleport = new UnityEvent();
        [SerializeField][Range(0f, 10f)] private float _teleportDelayTime = 2.0f;
        [SerializeField][Range(0f, 10f)] private float _teleportSicknessTime = 1.0f;

        [Header("Environment Shading Settings")]
        [SerializeField] public List<GameObject> TargetShadedObjectsToFacePlayerDirection = new
List<GameObject>();
        [SerializeField] private GameObject _firstPerson3DCamera = null;

        // Private var(s):
        private Transform _connectedPlayerShipTransform = null;
        private Vector3 _playerPos = Vector3.zero;
        private Vector3 _directionFromCenter = Vector3.zero;
        private Coroutine _entryTeleportFXRoutine = null;
        private Coroutine _exitTeleportFXRoutine = null;
        private bool _canWarpPlayer = false;
        private bool _canPlayerExitThroughWarp = false;
        private bool _teleportProcessStarted = false;

        #region MonoBehaviors
        protected override void Awake()
        {
            base.Awake();
        }

        void OnEnable()
        {
            _sphereCenterPosition = this.gameObject.transform.position;
            _canPlayerExitThroughWarp = true;
            _teleportProcessStarted = false;
            if (_exitTeleportFXRoutine != null)
            {
                InterruptTeleportExitRoutine();
            }
        }

        void OnDisable()
        {
            if (_entryTeleportFXRoutine != null)
            {
                InterruptTeleportEnterRoutine();
            }
            if (_exitTeleportFXRoutine != null)
            {
                InterruptTeleportExitRoutine();
            }
        }

        void OnDestroy()
        {
            if (_entryTeleportFXRoutine != null)
            {
                InterruptTeleportEnterRoutine();
            }
            if (_exitTeleportFXRoutine != null)
            {
                InterruptTeleportExitRoutine();
            }
        }

        // Update is called once per frame
        void Update()
        {
            // Treadmill Teleport Update:
```

```
            CheckIfPlayerNeedsToBeTeleportedBack(_connectedPlayerShipTransform);

            // Lit Environmental Shader Materials Update:
            UpdateLitMaterialsToFacePlayer();
        }

        void OnDrawGizmos()
        {
            // Draw playable area sphere:
            Gizmos.color = Color.yellow;
            Gizmos.DrawWireSphere(_sphereCenterPosition, _sphereRadius);

            // Draw first-person in-range of view sphere (anything past this is culled w/o
workarounds):
            if (_firstPerson3DCamera != null)
            {
                Color semiTransparentGizmoColor = Color.green;
                semiTransparentGizmoColor.a = 0.2f;
                Gizmos.color = semiTransparentGizmoColor;
                float renderRadius =
_firstPerson3DCamera.GetComponent<CinemachineVirtualCamera>().m_Lens.FarClipPlane;
                Vector3 centerPos = GetCurrentPlayerPos();
                Gizmos.DrawSphere(centerPos, renderRadius);
                Gizmos.DrawWireSphere(centerPos, renderRadius);
            }
        }

        void OnApplicationQuit()
        {
            ResetAllLitMaterialsFacingPosition();
        }
        #endregion



        #region Coroutine(s)
        IEnumerator WaitOnTeleportEntryFX()
        {
            Debug.Log("Started Teleport Treadmill Entry Coroutine.");
            _canWarpPlayer = false;
            yield return new WaitForSeconds(_teleportDelayTime);
            _canWarpPlayer = true;
            Debug.Log("Ending Teleport Treadmill Entry Coroutine...");
            _entryTeleportFXRoutine = null;
        }

        IEnumerator WaitOnTeleportExitFX()
        {
            Debug.Log("Started Teleport Treadmill Exit Coroutine.");
            _canPlayerExitThroughWarp = false;
            yield return new WaitForSeconds(_teleportSicknessTime);
            _canPlayerExitThroughWarp = true;
            Debug.Log("Ending Teleport Treadmill Exit Coroutine...");
            _exitTeleportFXRoutine = null;
        }
        #endregion

        #region Helper Function(s)
        private void CheckIfPlayerNeedsToBeTeleportedBack(Transform currentPlayerTransform)
        {
            if (currentPlayerTransform != null)
            {
                _playerPos = GetCurrentPlayerPos(currentPlayerTransform);
                _directionFromCenter = _playerPos - _sphereCenterPosition;

                // Determine if player is out of bounds:
                // (Vector length to sphere center > radius of sphere)
                if ((_directionFromCenter.magnitude > _sphereRadius) && _canPlayerExitThroughWarp)
                {
```

```
                    // ...

                    if (_entryTeleportFXRoutine == null && _exitTeleportFXRoutine == null &&
!_teleportProcessStarted)
                    {
                        OnTeleportPlayerReady.Invoke();
                        _teleportProcessStarted = true;
                        _entryTeleportFXRoutine = StartCoroutine(WaitOnTeleportEntryFX());
                    }
                    if (_canWarpPlayer)
                    {
                        TeleportToOppositeEnd(currentPlayerTransform);

                        if (_exitTeleportFXRoutine == null)
                        {
                            OnPlayerTeleported.Invoke();
                            _exitTeleportFXRoutine = StartCoroutine(WaitOnTeleportExitFX());
                            _teleportProcessStarted = false;
                            OnPlayerOutOfTeleport.Invoke();
                        }
                    }
                }
            }
        }

        private void TeleportToOppositeEnd(Transform playerTransform)
        {
            _directionFromCenter.Normalize();

            if (_isPlayerTeleportedToEdgeOfMap)
            {
                // To opposite edge (Asteroids-style):
                playerTransform.position = _sphereCenterPosition - _directionFromCenter *
_sphereRadius;
            }
            else
            {
                // To center of treadmill sphere in same direction:
                playerTransform.position = _sphereCenterPosition - _directionFromCenter;
            }
        }

        private void UpdateLitMaterialsToFacePlayer()
        {
            Vector3 playerPos = GetCurrentPlayerPos();

            foreach (GameObject shadedObj in TargetShadedObjectsToFacePlayerDirection)
            {
                if (shadedObj != null)
                {
                    // Get Unity properties:
                    Material targetMat = shadedObj.GetComponent<Renderer>().material;
                    Transform targetTransform = shadedObj.transform;

                    if ((targetMat != null) && (targetTransform != null))
                    {
                        // Process params:
                        Vector3 targetObjPos = targetTransform.position;

                        // Set shader properties on this shaded object:
                        targetMat.SetVector("_PlayerPositionToIlluminate", playerPos);
                        targetMat.SetVector("_CenterWSPosition", targetObjPos);
                    }
                }
            }
        }

        private void ResetAllLitMaterialsFacingPosition()
        {
```

```
            foreach (GameObject shadedObj in TargetShadedObjectsToFacePlayerDirection)
            {
                if (shadedObj != null)
                {
                    Material targetMat = shadedObj.GetComponent<Renderer>().material;

                    if (targetMat != null)
                    {
                        targetMat.SetVector("_PlayerPositionToIlluminate", Vector3.zero);
                    }
                }
            }
        }

        private Vector3 GetCurrentPlayerPos(Transform usingThisTransform = null)
        {
            Vector3 foundPos = Vector3.zero;

            if (usingThisTransform != null)
            {
                foundPos = usingThisTransform.position;
            }
            else if (_connectedPlayerShipTransform != null)
            {
                foundPos = _connectedPlayerShipTransform.position;
            }

            return foundPos;
        }
        #endregion

        #region Coroutine Helper(s)
        public void InterruptTeleportEnterRoutine()
        {
            StopCoroutine(_entryTeleportFXRoutine);
            _entryTeleportFXRoutine = null;
            _canWarpPlayer = false;
        }

        public void InterruptTeleportExitRoutine()
        {
            StopCoroutine(_exitTeleportFXRoutine);
            _exitTeleportFXRoutine = null;
            _canPlayerExitThroughWarp = true;
        }
        #endregion

        #region Public Mutator(s)
        public void SetPlayerShipTransform(Transform updatedPlayerTransform)
        {
            if (updatedPlayerTransform != null)
            {
                _connectedPlayerShipTransform = updatedPlayerTransform;
            }
        }
        #endregion
    }
}
```

Code 12. The “LevelManager.cs” script that contains all treadmill teleport functionality.

Additionally, some treadmill sphere gizmos were made to visualize the sphere of the playable area along with the render distance of the player, shown in the below figure.

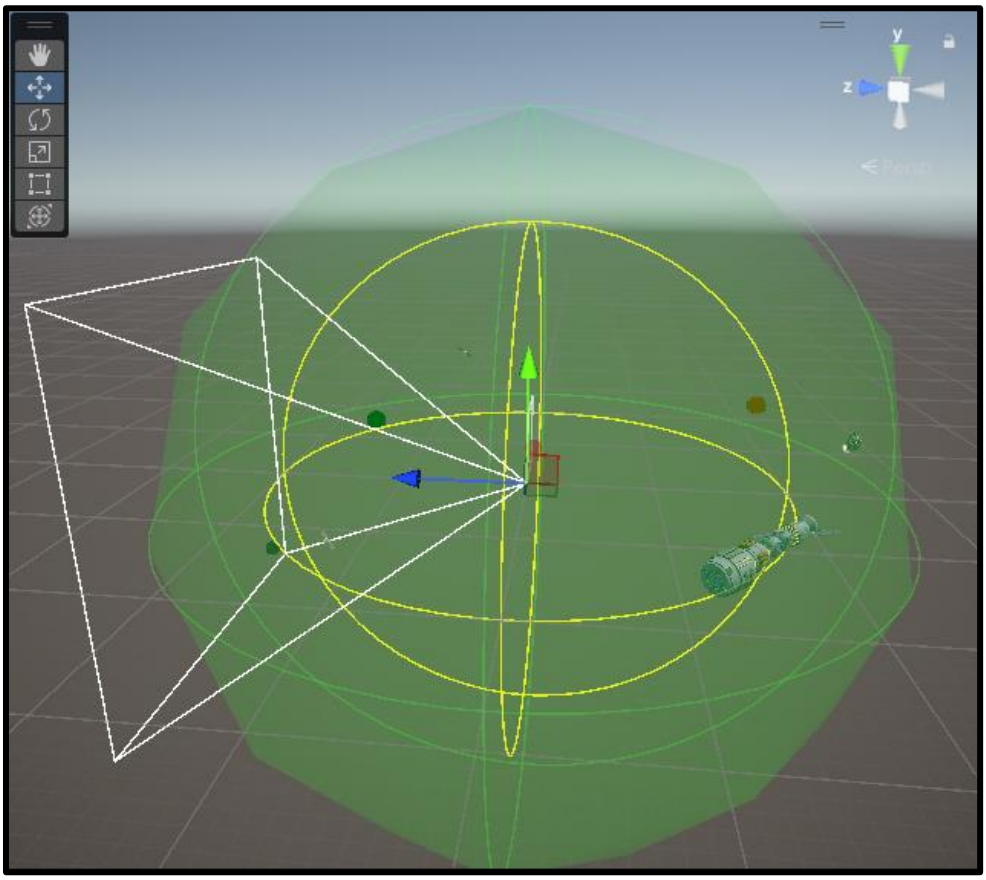

Figure 21. Visual of treadmill sphere gizmos to show teleport boundary (yellow) and rendering boundary (green).

You may have noticed that in the above "LevelManager.cs" script that there are a couple of coroutines as part of the teleporting process. These coroutines essentially help manage the teleport effect by adding timers to know when to apply certain visual effects or player cues, as well as some post-teleport timers to add "cooling down/returning to normal" effects. Overall, these placeholders should be able to allow the current teleport mechanic to be less abrupt and sudden to the player's experience.

Next in line in terms of major game environment changes are the destructible asteroids, which also serve as the main "enemy" or "obstacle" of the game. Since creating a full-fledged

destructible system can be quite time-consuming, this project will backend off of the "Breakable Asteroids" asset store package that was made by Chadderbox. After importing a bunch of the sample asteroid prefabs into the "Asteroid Belt" project, they were able to be unpacked completely and later equipped with supplementary collision-detection scripts developed to support receiving hits from the player's blaster projectiles. Here are the following scripts attached or included in each new asteroid and fractured asteroid prefab:

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

public class FracturedAsteroid : MonoBehaviour
{
    [SerializeField][Range(1f, 60f)] private float _duration = 2f;

    private void OnEnable()
    {
        Destroy(gameObject, _duration);
    }
}
```

Code 13. The FracturedAsteroid script attached to each destructible version of the asteroid.

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

public class Asteroid : MonoBehaviour, IDamageable
{
    [SerializeField] private FracturedAsteroid _fracAsteroidPrefab;
    [SerializeField] private Detonator _explosionPrefab;

    private Transform _transform;

    void Awake()
    {
        _transform = transform;
    }

    public void TakeDamage(int damage, Vector3 hitPosition)
    {
        FractureTheAsteroid(hitPosition);
    }

    private void FractureTheAsteroid(Vector3 hitPosition)
    {
        // Instance the fragmented pieces
        if (_fracAsteroidPrefab != null)
        {
            Instantiate(_fracAsteroidPrefab, _transform.position, _transform.rotation);
        }
```

```
        // Instance the "explosion" where we hit the thing
        if (_explosionPrefab != null)
        {
            Instantiate(_explosionPrefab, hitPosition, Quaternion.identity);
        }

        Destroy(this.gameObject);
    }
}
```

Code 14. The Asteroid script hooked to each initially instanced asteroid to allow it to take damage and become fractured (destroyed which then splits into pieces).

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

public interface IDamageable
{
    public void TakeDamage(int damage, Vector3 hitPosition);
}
```

Code 15. The IDamageable interface that declares the TakeDamage() method to implement on every asteroid prefab.

Currently, there is no asteroid spawner in the game yet, but the plan is to spawn asteroids and throw them at and near the player at reasonable speeds for them to shoot at until a win or loss condition is met. Despite this spawner still yet to be implemented, you can still see the current destructible asteroids in-action with the player ship's fire mechanic in the below figure.

Figure 22. Before and after visual of shooting a destructible asteroid instance.

For some extra juice on the asteroids, a “Detonator Explosion Framework” asset was imported from Midnite Oil Software’s GitHub repository as a Unity Package file. It couldn’t be retrieved from the official Unity Asset Store as it has been marked deprecated and no longer supported on Unity’s page (original store publisher for the framework was Ben Throop). Despite its deprecation, the asset still works great in the current URP project and has been referenced in the previous Asteroid script module (“Detonator” type) to be displayed upon destruction.

Figure 23. Demo of "Detonator Explosion Framework"; using 'ignitor' effect in this example.

The remaining environment additions to the project up until this revision were just adding in dynamic planets, that had their own Fresnel-illumination shaders, and some static environment objects for decoration such as satellites and space stations to add to the sci-fi feel. These final static environmental additions are shown in the below figure for reference.

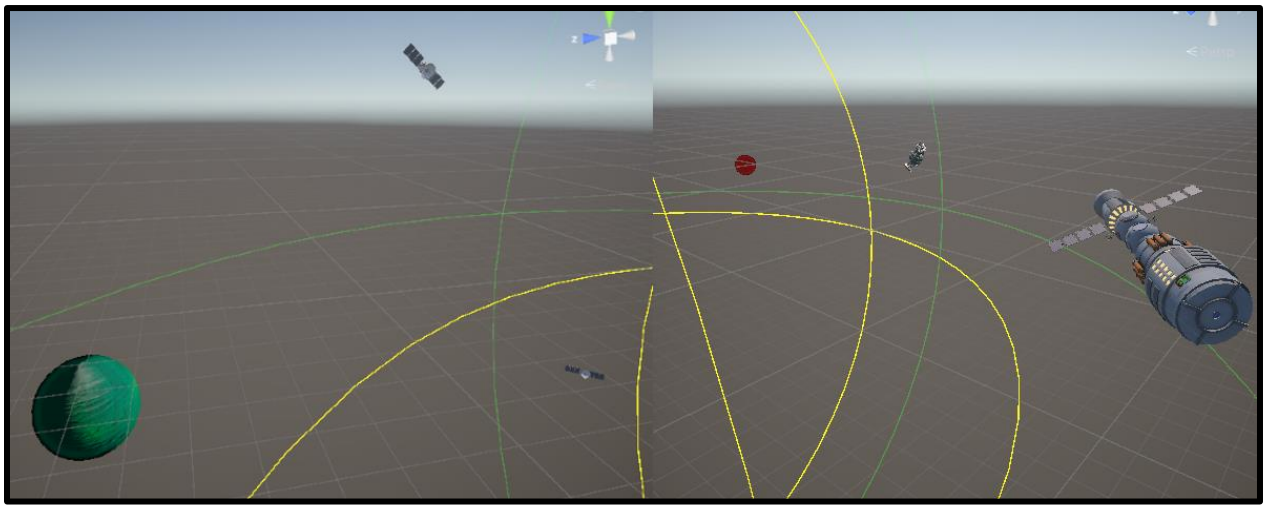

Figure 24. Visual showcasing the environment's planet shader props and the static sci-fi models.

Note that the above planets in Figure 24 were just scaled up Unity spheres while the space station and satellites were imported from FlexUnit's "Sci-Fi Space Station" and "Animated Sci-Fi Satellite" packages, respectively, on the Unity Asset Store.

User-Interface

The UI of "Asteroid Belt" is currently barebones/static, and no pause or main menu screens have been implemented yet into the game. However, there is in-game UI for the ship's HP bar and score counter but only as static elements for now. Below is a screenshot of the current in-game UI, both in scene and game view.

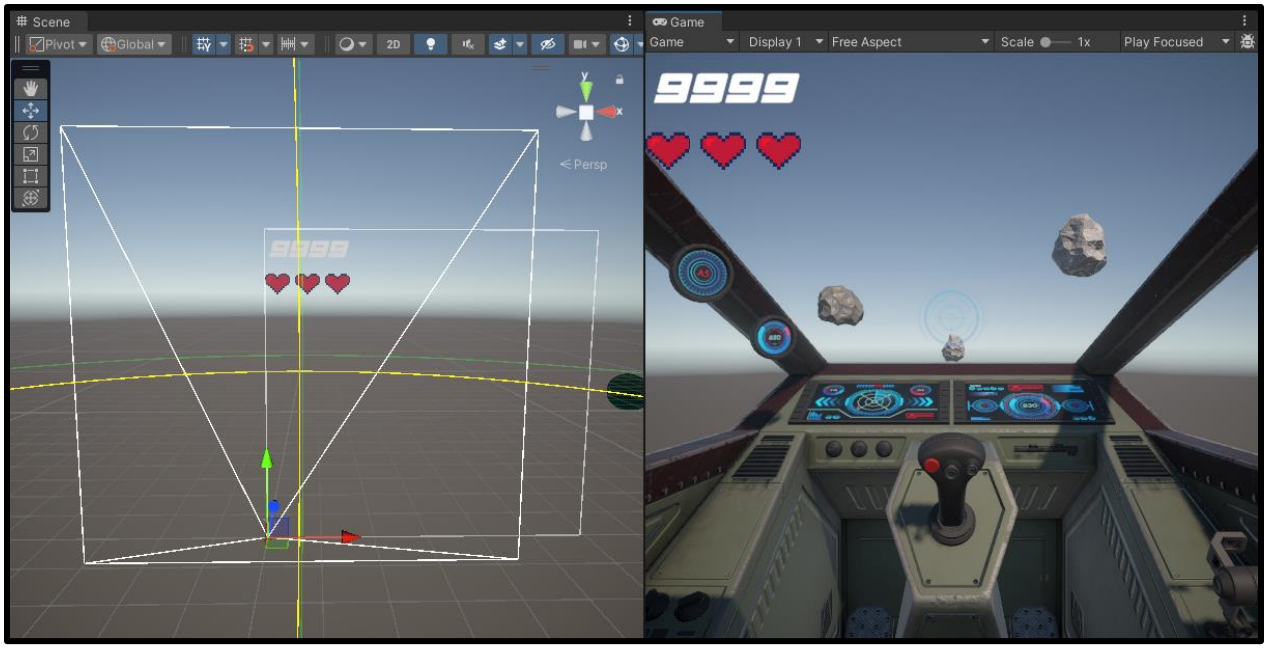

Figure 25. Scene and in-game view of the current screen overlay UI with the player's ship.

Note that the heart icons, courtesy from asset publisher MR3_2004 in Unity Asset Store, have been formatted using a "Horizontal Layout Group" component in the UI panel's Inspector. This component allows the icons to spread out evenly while still being perfectly aligned to each other.

Regarding startup/sketched UI prototypes, the plan is to still use *Figure 3*'s UI sketch-out for the pause menu UI layout, as seen in the start of this document under the "Visuals" section. Meanwhile, the UI of the main menu and controls screen will go off of the following new visuals, seen below.

Figure 26. Visual of main menu UI rough-up for what its design is to look like.

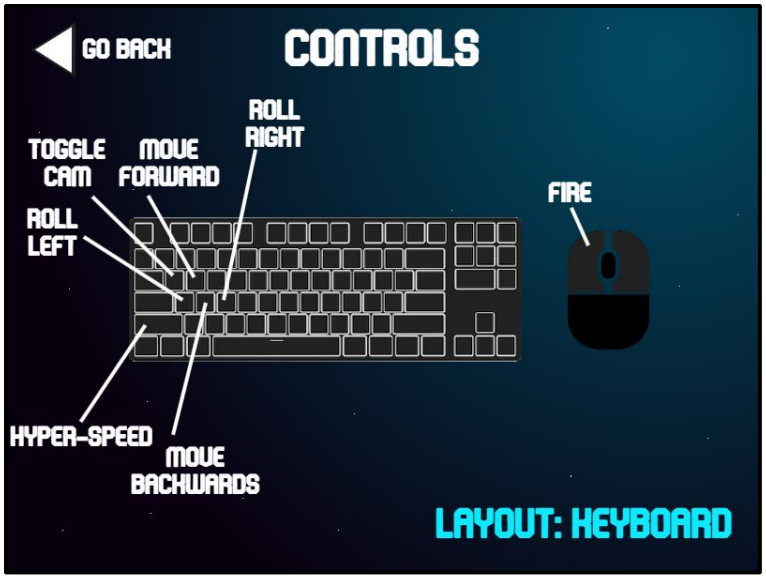

Figure 27. Visual of controls UI sketch-up to indicate its future design.

Bugs & Challenges

Since the majority of technical development was spent on the player and camera setup, the bugs were found mostly to be with the player's RigidBody components and effects. For example, one major bug found was the player's blaster cannons weren't aligned to the crosshair inside the model's cockpit and hence would fire too far below the cockpit's view. To fix this, a ray was cast from the center of the crosshair object and the blasters simply use Unity's LookAt function to point to a spot on the center ray. This, in turn, aligns the blaster's projectiles with the endpoint the crosshair is aiming at from inside the cockpit. Another bug with the player's ship asset was the conflict of RigidBodies between the spaceship and the blaster projectiles. To elaborate, the projectiles had their own RigidBody to enable both physical and ray-cast collision with asteroids to add on fallback collision detection since they are travelling at such high velocities. However, since these projectiles spawned inside the ship's muzzle, their RigidBody would collide with the ship's RigidBody first and thus cause a knockback and misfire. The fix to this bug was to edit the collision matrix to remove player projectiles from hitting player objects and add additional layer checks on what was hit before moving on with the collision routine.

As for challenges, the only real challenge faced so far is with the environment's playable space treadmill mechanism. The hurdle with the treadmill is that it has to be teleport based currently since the player inputs have been built out to actually move the player and not the terrain. The reason for this being called a challenge is that the teleport based version of the treadmill has a lot more abrupt effects that are more noticeable to the player and hence require more juice and layered effects to smooth out the mechanism.

Overall Time Dedicated

So far, at the time of writing this document, the average time dedicated per week to the "Asteroid Belt" game development project has been approximately 10 hours. Ignoring the 1st week where the project was still in its proposal stages, that would total out to 40 hours up until this report's revision date, as 4 weeks have already passed. A majority of this time invested in the project was spent on developing the player controller and ship prefab since that was the initial foundation or core of the game to build around. Giving an estimate, it would be around 24 out of the 40 hours spent on the player's input and controls. The remaining 16 hours out of that 40 was spent setting up camera mechanics, toggles, and special effects (like environmental shaders, post-processing, etc.).

Additionally, there were approximately 10 hours of crunch time to get some last minute polishing done before the demo video which equals to a grand total of 50 hours overall dedicated to "Asteroid Belt" thus far.

References

Midnite Oil Software LLC (2022, June 23). *How To Make a 3D Space Shooter Game in Unity – Tutorial[Video]*. YouTube. https://youtu.be/VW3PkEF1Fzk?si=nlgnGwqIz5dyxTtG.

Ebal Studios (2023, November 15). Hi-Rez Spaceships Creator Free Sample [Software Asset]. Retrieved from https://assetstore.unity.com/packages/3d/vehicles/space/hi-rez-spaceships-creator-free-sample-153363.

Daniel Ilett (2023, March 21). "Outline Post Process in Unity Shader Graph (URP)- Fun With Fullscreen Graphs". https://danielilett.com/2023-03-21-tut7-1-fullscreen-outlines/.

Chadderbox (2020, May 20). Breakable Asteroids [Software Asset]. Retrieved from https://assetstore.unity.com/packages/3d/props/breakable-asteroids-167825.

Ben Throop (2019, October 8). Detonator Explosion Framework [Software Asset]. Retrieved from https://github.com/gbradburn/3D-Space-Shooter-Tutorial/blob/Part_Four/Assets/Custom%20packages/Detonator%20Explosion%20Framework.unitypackage.

Stuck Muck Interactive (2017, January 18). Planet Texture Generator [Software Asset]. Retrieved from https://assetstore.unity.com/packages/tools/planet-texture-generator-51995.

MR3_2004 (2024, July 5). Heart in pixel [Software Asset]. Retrieved from https://assetstore.unity.com/packages/2d/gui/icons/heart-in-pixel-287862.

FlexUnit (2024, July 4). Animated Sci-Fi Satellite [Software Asset]. Retrieved from https://assetstore.unity.com/packages/3d/vehicles/space/animated-sci-fi-satellite-288027.

FlexUnit (2024, August 6). Sci-Fi Space Station [Software Asset]. Retrieved from https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-space-station-290875.

Pulsar Bytes (2017, February 6). SpaceSkies Free [Software Asset]. Retrieved from https://assetstore.unity.com/packages/2d/textures-materials/sky/spaceskies-free-80503.

OccaSoftware (2023, May 19). Crosshairs [Software Asset]. Retrieved from https://assetstore.unity.com/packages/2d/gui/icons/crosshairs-216732.