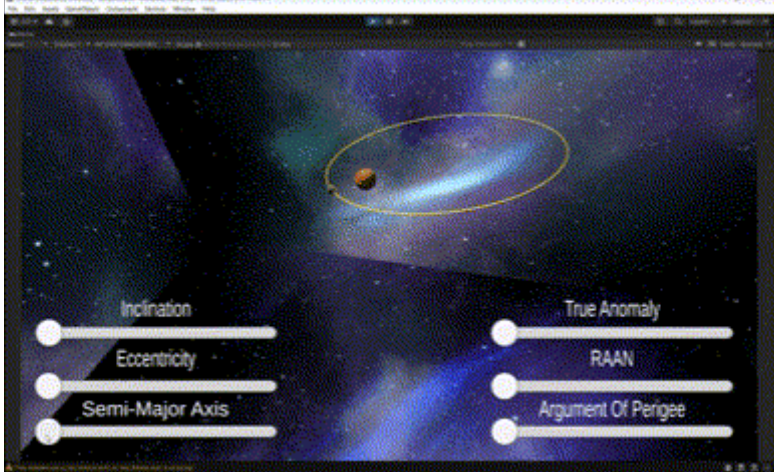
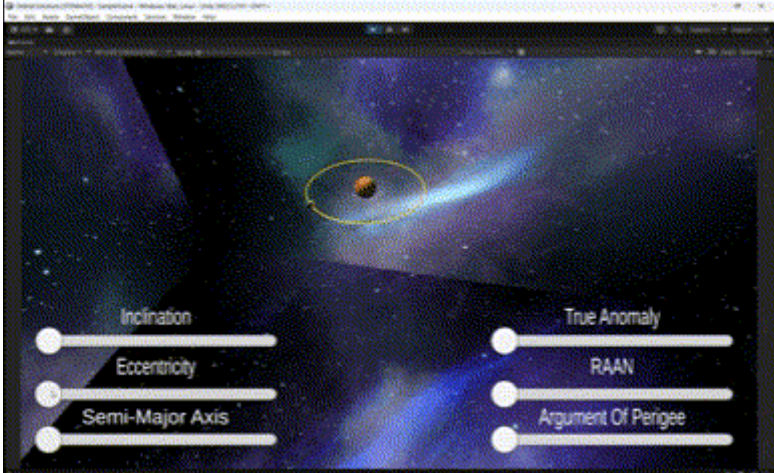
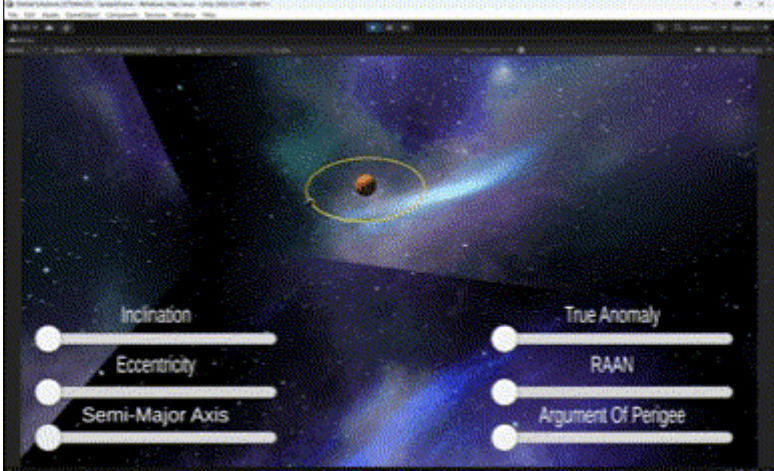
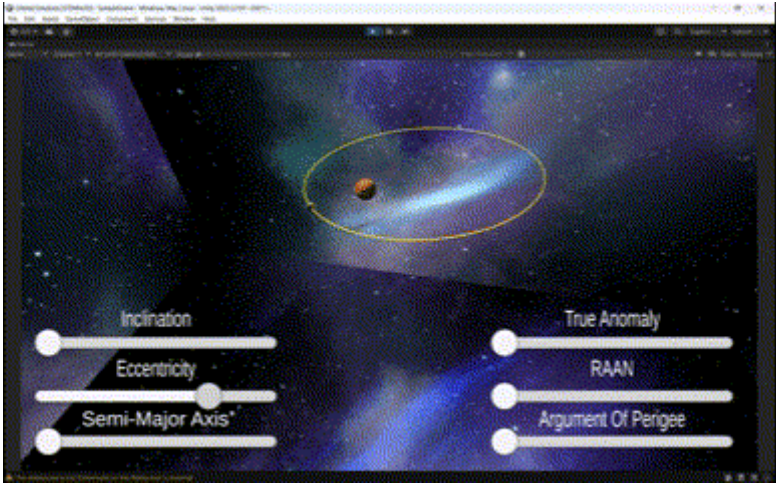
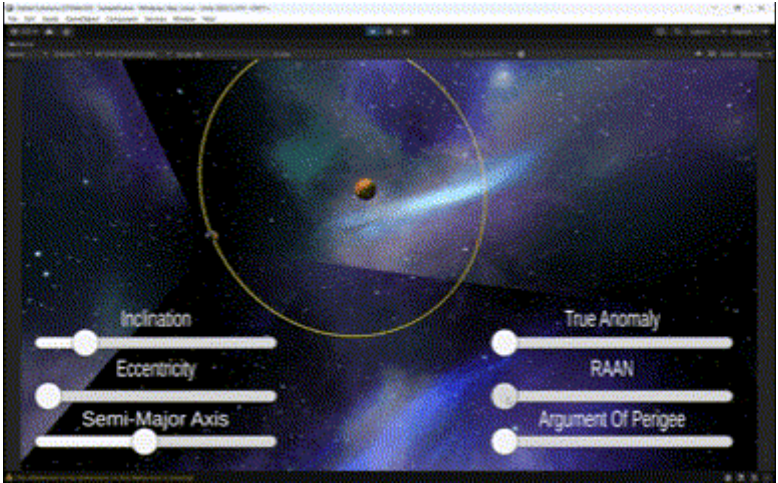
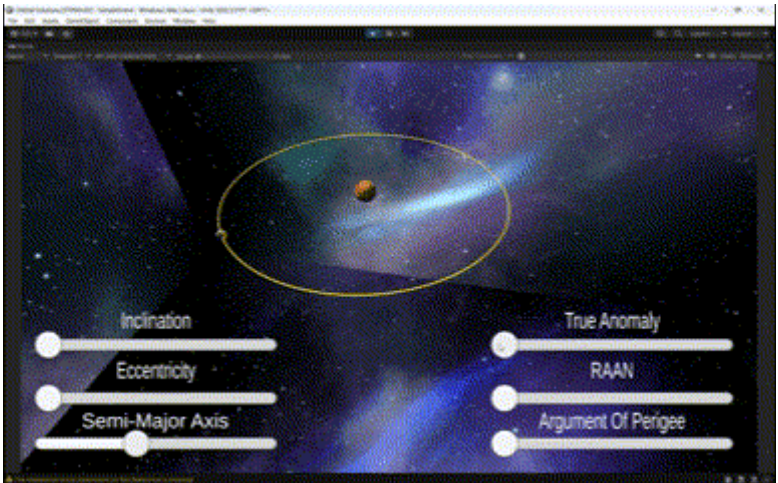


How To Use The Simulation

This simulation has sliders with their respective orbital elements above them. Those sliders will change the value when they are slid back and forth. Below this, each slider will be explained and there will be pictures of what it does.

Orbital Elements	Explanation	Demonstration of Slider
Semi-Major Axis	This orbital element changes the size of the orbit. It is how big or small the object is orbiting around another object.	
Eccentricity	This orbital element changes the shape of the orbit. An orbit's original shape is a circle, which changes the orbit to an elliptical shape.	
Inclination	This orbital element changes the tilt of the orbit. At first, the orbit will have no tilt but will be changed depending on the slider value.	

<p>Argument Of Perigee</p>	<p>This rotates the orbit as you change the slider and the periapsis's position.</p>	
<p>RAAN (Right Ascension Of the Ascending Node)</p>	<p>This is the orbital twist of the orbit and the slider changes that. You can't see if you don't tilt the orbit because it would just look like the argument of perigee.</p>	
<p>True Anomaly</p>	<p>This changes the position of the object in orbit. The slider value goes up to 1; when it reaches 1, it is a full orbit.</p>	

Explanation Of The Scripts

TrueAnomaly script

```
C/C++
using UnityEngine;
using UnityEngine.UI;
using SolarSystem;

public class YearTSliderController : MonoBehaviour
{
    public Slider yearTSlider;
    public SolarSystemManager solarSystemManager;

    void Start()
    {
        //This if statment checks if the yearTSlider is not assigned and if it isn't then it will
        get the Slider component from the current gameobject.
        if (yearTSlider == null)
        {
            yearTSlider = GetComponent<Slider>();
        }

        //This if statment checks if the solarSystemManager is assigned and if it isn't assigned
        it will find an instance in the scene.
        if (solarSystemManager == null)
        {
            solarSystemManager = FindObjectOfType<SolarSystemManager>();
        }

        yearTSlider.onValueChanged.AddListener(OnSliderValueChanged);
    }

    // This method updates the yearT value in the SolarSystemManager when the value of the
    slider changes.
    void OnSliderValueChanged(float value)
    {
        if (solarSystemManager != null)
        {
            solarSystemManager.yearT = value;
        }
    }
}
```


Sun Script

```
C/C++
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace SolarSystem
{
    // The Sun class manages the sun's position, color, and related animations in the solar
    system
    [ExecuteInEditMode] // Allows the script to be executed in edit mode in Unity
    public class Sun : MonoBehaviour
    {

        public bool animateSunColour;
        public Gradient sunColGradient;
        public float dayStartOffset;
        Light lightSource; // Reference to the Light component on the Sun
        Camera cam; // Reference to the main Camera

        [Header("Debug")]
        public Color sunColour;
        public float timeOfDayT;
        public int maxSize;

        void Start()
        {

            lightSource = GetComponent<Light>();

            cam = Camera.main;
        }

        // Update the sun's position and rotation based on the Earth's orbit
        // geocentric: if true, the sun orbits around the Earth; otherwise, the Earth orbits
        around the Sun
        public void UpdateOrbit(EarthOrbit earth, bool geocentric)
        {
            if (geocentric)
            {
                // Calculate the sun's position relative to a stationary Earth
                transform.position = Quaternion.Inverse(earth.earthRot) * -earth.earthPos;
                // Rotate the sun to look at the Earth's center
                transform.LookAt(Vector3.zero);

                // Update the sun's color based on its position
                UpdateColourApprox(Vector3.zero);
            }
        }
    }
}
```

```

    }
    else
    {
        // Set the sun's position to the origin
        transform.position = Vector3.zero;
        // Rotate the sun to look at the Earth's position
        transform.LookAt(earth.earthPos);

        // Update the sun's color based on its position
        UpdateColourApprox(earth.earthPos);
    }
}

// Estimate the sun's light color based on its angle relative to the viewer
// This avoids reading data from a GPU-based atmosphere system
void UpdateColourApprox(Vector3 earthPos)
{
    // Calculate the direction from the camera to the Earth
    Vector3 dirToCam = (cam.transform.position - earthPos).normalized;
    // Calculate the direction from the sun to the Earth
    Vector3 dirToSun = -transform.forward;
    // Calculate the time of day as a value between 0 and 1
    timeOfDayT = Mathf.Max(0, (Vector3.Dot(dirToCam, dirToSun) + dayStartOffset) / (1 +
dayStartOffset));
    // Evaluate the sun color based on the time of day
    sunColour = sunColGradient.Evaluate(timeOfDayT);

    // If sun color animation is enabled, update the Light component's color
    if (animateSunColour)
    {
        lightSource.color = sunColour;
    }
}
}
}

```

Solar System Manager Script

```
C/C++
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace SolarSystem
{
    // The SolarSystemManager class manages the animation and state of the solar system
    [ExecuteInEditMode] // Allows the script to be executed in edit mode in Unity
    public class SolarSystemManager : MonoBehaviour
    {
        // Public variables to be assigned in the Unity Editor
        public bool animate;

        [Header("Durations")]
        public float dayDurationMinutes; // Duration of a day in minutes
        public float monthDurationMinutes; // Duration of a month in minutes
        public float yearDurationMinutes; // Duration of a year in minutes

        [Header("References")]
        public Sun sun; // Reference to the Sun component
        public EarthOrbit earth; // Reference to the EarthOrbit component
        public Transform player; // Reference to the player transform

        [Header("Time state")]
        [Range(0, 1)]
        public float dayT; // Current time of the day (0 to 1)
        [Range(0, 1)]
        public float monthT; // Current time of the month (0 to 1)
        [Range(0, 1)]
        public float yearT; // Current time of the year (0 to 1)

        public float fastForwardDayDuration; // Duration for fast forwarding the day
        float oldPlayerT; // Previous player time
        float fastForwardTargetTime; // Target time for fast forwarding
        bool fastForwardApproachingTargetTime; // Whether fast forwarding is approaching the
        target time

        [Header("Debug")]
        public bool geocentric; // Whether the system is geocentric (Earth-centered)

        // Update is called once per frame
        void Update()
        {
            // Update the Earth's orbit based on the current time and geocentric setting
            earth?.UpdateOrbit(yearT, dayT, geocentric);
            // Update the Sun's orbit based on the Earth's orbit and geocentric setting
            sun?.UpdateOrbit(earth, geocentric);
        }
    }
}
```

```

    }

    // Set the current times for day, month, and year
    public void SetTimes(float dayT, float monthT, float yearT)
    {
        this.dayT = dayT;
        this.monthT = monthT;
        this.yearT = yearT;
    }

    // Calculate the player's current time of day based on their position relative to the
Sun
    float CalculatePlayerDayT()
    {
        return Vector3.Dot(player.position.normalized, -sun.transform.forward);
    }

    // Calculate the distance to the target time from a given time
    float DstToTargetTime(float fromT, float targetT)
    {
        return Mathf.Abs(targetT - fromT);
    }
}

```

Skybox Rotate Script

```

C/C++
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SkyboxRotate : MonoBehaviour
{
    // Public variable to set the rotation speed of the skybox
    public float speed;

    // Update is called once per frame
    void Update()
    {
        // Rotate the skybox by setting the "_Rotation" property based on the time elapsed and
the speed
        RenderSettings.skybox.SetFloat("_Rotation", Time.time * speed);
    }
}

```

Orbit Control UI Script

C/C++

```
using UnityEngine;
using UnityEngine.UI;
using SolarSystem;

public class OrbitControlUI : MonoBehaviour
{
    // Public variables to be assigned in the Unity Editor
    public Slider yearTSlider;
    public Slider inclinationSlider;
    public Slider argumentOfPerigeeSlider;
    public Slider rightAscensionSlider;

    public SolarSystemManager solarSystemManager;

    // Start is called before the first frame update
    void Start()
    {
        // Check if all sliders are assigned, if not, log an error and return
        if (yearTSlider == null || inclinationSlider == null || argumentOfPerigeeSlider == null
|| rightAscensionSlider == null)
        {
            Debug.LogError("Please assign all sliders in the Inspector.");
            return;
        }

        // If the SolarSystemManager is not assigned, find it in the scene
        if (solarSystemManager == null)
        {
            solarSystemManager = FindObjectOfType<SolarSystemManager>();
        }

        // Add listeners to the sliders to call the appropriate methods when the slider values
change
        yearTSlider.onValueChanged.AddListener(OnYearTSliderChanged);
        inclinationSlider.onValueChanged.AddListener(OnInclinationSliderChanged);
        argumentOfPerigeeSlider.onValueChanged.AddListener(OnArgumentOfPerigeeSliderChanged);
        rightAscensionSlider.onValueChanged.AddListener(OnRightAscensionSliderChanged);
    }

    // Method to update the yearT value in the SolarSystemManager
    void OnYearTSliderChanged(float value)
    {
        solarSystemManager.yearT = value;
    }
}
```



```
// Method to update the inclination value and refresh the orbit
void OnInclinationSliderChanged(float value)
{
    solarSystemManager.earth.inclination = value;
    solarSystemManager.earth.UpdateOrbit(solarSystemManager.yearT, solarSystemManager.dayT,
solarSystemManager.geocentric);
}

// Method to update the argument of perigee value and refresh the orbit
void OnArgumentOfPerigeeSliderChanged(float value)
{
    solarSystemManager.earth.argumentOfPerigee = value;
    solarSystemManager.earth.UpdateOrbit(solarSystemManager.yearT, solarSystemManager.dayT,
solarSystemManager.geocentric);
}

// Method to update the right ascension value and refresh the orbit
void OnRightAscensionSliderChanged(float value)
{
    solarSystemManager.earth.rightAscension = value;
    solarSystemManager.earth.UpdateOrbit(solarSystemManager.yearT, solarSystemManager.dayT,
solarSystemManager.geocentric);
}
}
```

Orbit Script

```
C/C++
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using static System.Math;

namespace SolarSystem
{
    public class Orbit : MonoBehaviour
    {
        // Calculate the position of a point on the orbit at a given t value (0 to 1)
        // The orbit is defined by the periapsis (closest distance) and apoapsis (farthest distance)
        public static Vector3 CalculatePointOnOrbit(double periapsis, double apoapsis, double t, double inclination, double argumentOfPerigee, double rightAscension)
        {
            // Calculate the semi-major axis length
            double semiMajorLength = (apoapsis + periapsis) / 2;
            // Calculate the linear eccentricity
            double linearEccentricity = semiMajorLength - periapsis;
            // Calculate the eccentricity
            double eccentricity = linearEccentricity / semiMajorLength;
            // Calculate the semi-minor axis length
            double semiMinorLength = Sqrt(Pow(semiMajorLength, 2) - Pow(linearEccentricity, 2));

            // Calculate the mean anomaly
            double meanAnomaly = t * PI * 2;
            // Solve for the eccentric anomaly using Kepler's equation
            double eccentricAnomaly = SolveKepler(meanAnomaly, eccentricity);

            // Calculate the position in the plane of the ellipse
            double ellipseCentreX = -linearEccentricity;
            double pointX = Cos(eccentricAnomaly) * semiMajorLength + ellipseCentreX;
            double pointY = Sin(eccentricAnomaly) * semiMinorLength;

            Vector3 position = new Vector3((float)pointX, 0, (float)pointY);

            // Apply rotations for argument of perigee, inclination, and right ascension of the ascending node (RAAN)
            Quaternion rotation = Quaternion.Euler(0, (float)rightAscension, 0) *
                Quaternion.Euler((float)inclination, 0, 0) *
                Quaternion.Euler(0, (float)argumentOfPerigee, 0);

            // Return the rotated position
            return rotation * position;
        }

        // Solve Kepler's equation for the eccentric anomaly using the Newton-Raphson method
    }
}
```

```

static double SolveKepler(double meanAnomaly, double eccentricity, int maxIterations =
100)
{
    const double h = 0.0001; // step size for approximating the gradient
    const double acceptableError = 0.00000001; // acceptable error margin
    double guess = meanAnomaly; // initial guess for the eccentric anomaly

    for (int i = 0; i < maxIterations; i++)
    {
        double y = KeplerEquation(guess, meanAnomaly, eccentricity);
        if (Abs(y) < acceptableError)
        {
            break; // exit if the error is within the acceptable range
        }
        double slope = (KeplerEquation(guess + h, meanAnomaly, eccentricity) - y) / h;
        // approximate the gradient
        double step = y / slope; // calculate the step size
        guess -= step; // update the guess
    }
    return guess;

    // Kepler's equation:  $M - E + e \cdot \sin(E) = 0$ 
    double KeplerEquation(double E, double M, double e)
    {
        return M - E + e * Sin(E);
    }
}

```

Earth Orbit Script

```
C/C++
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace SolarSystem
{
    public class EarthOrbit : MonoBehaviour
    {
        // Public properties to get the Earth's rotation, position, and current axis angle
        public Quaternion earthRot { get; private set; }
        public Vector3 earthPos { get; private set; }
        public float currentAxisAngle { get; private set; }

        // Orbital parameters
        public float periapis = 147.2f; // Periapsis distance
        public float apoapsis = 152.1f; // Apoapsis distance
        public float inclination = 23.4f; // Orbital inclination
        public float argumentOfPerigee = 0f; // Argument of perigee
        public float rightAscension = 0f; // Right ascension of the ascending node

        public float distanceScale = 1; // Scale for the distance

        [Header("Debug")]
        public float debug_dst; // Debug variable for distance

        public Vector3 orbitEllipse { get; private set; } // The calculated orbit ellipse

        public static SolarSystemManager time; // Static reference to the SolarSystemManager

        public Vector2 angle { get; private set; } // Angle of the orbit

        public LineRenderer lineRenderer; // Line renderer for drawing the orbit
        public int segments = 100; // Number of segments for the orbit line

        // Method to set the size of the orbit
        public void size(float newSize)
        {
            periapis = newSize;
            apoapsis = newSize;
        }

        // Method to set the periapsis of the orbit
        public void shapeP(float newPeriapis)
        {
            periapis = newPeriapis;
        }
    }
}
```

```

// Method to set the apoapsis of the orbit
public void shapeA(float newApoapsis)
{
    apoapsis = newApoapsis;
}

// Method to update the orbit based on time and geocentric flag
public void UpdateOrbit(float yearT, float dayT, bool geocentric)
{
    // Calculate the orbit ellipse
    orbitEllipse = Orbit.CalculatePointOnOrbit(periapis, apoapsis, yearT, inclination,
argumentOfPerigee, rightAscension);
    // Calculate the Earth's position in the orbit
    earthPos = orbitEllipse * distanceScale;
    debug_dst = new Vector2(orbitEllipse.x, orbitEllipse.z).magnitude;

    // Calculate the sidereal and solar day angles
    float siderealDayAngle = -dayT * 360;
    float solarDayAngle = siderealDayAngle - yearT * 360;
    currentAxisAngle = solarDayAngle;

    // Calculate the Earth's rotation
    earthRot = Quaternion.Euler(0, 0, -inclination) * Quaternion.Euler(0,
currentAxisAngle, 0);

    if (geocentric)
    {
        // If geocentric, set position and rotation to default
        transform.position = Vector3.zero;
        transform.rotation = Quaternion.identity;
    }
    else
    {
        // Otherwise, set position and rotation based on calculated values
        transform.position = earthPos;
        transform.rotation = earthRot;
    }

    // Draw the orbit
    DrawOrbit();
}

// Method to draw the orbit using the LineRenderer
private void DrawOrbit()
{
    Vector3[] points = new Vector3[segments + 1];
    for (int i = 0; i <= segments; i++)
    {
        float t = (float)i / (float)segments;

```



```

        Vector3 point = Orbit.CalculatePointOnOrbit(periapis, apoapsis, t, inclination,
argumentOfPerigee, rightAscension);
        Vector3 pos = point * distanceScale;
        points[i] = pos;
    }

    points[segments] = points[0];

    lineRenderer.positionCount = segments + 1;
    lineRenderer.SetPositions(points);
}

// Method called when the script is loaded or a value is changed in the Inspector
void OnValidate()
{
    if (Application.isPlaying)
    {
        DrawOrbit();
    }
}
}
}

```

Change Scenes Script

```

C/C++
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class ChangeScenes : MonoBehaviour
{
    // Method to be called when the start button is clicked
    public void OnStartButtonClick()
    {
        // Load the scene named "SimulationScene"
        SceneManager.LoadScene("SimulationScene");
    }

    // Method to be called when the quit button is clicked
    public void QuitGame()
    {
        // Quit the application
        Application.Quit();
    }
}

```

} }