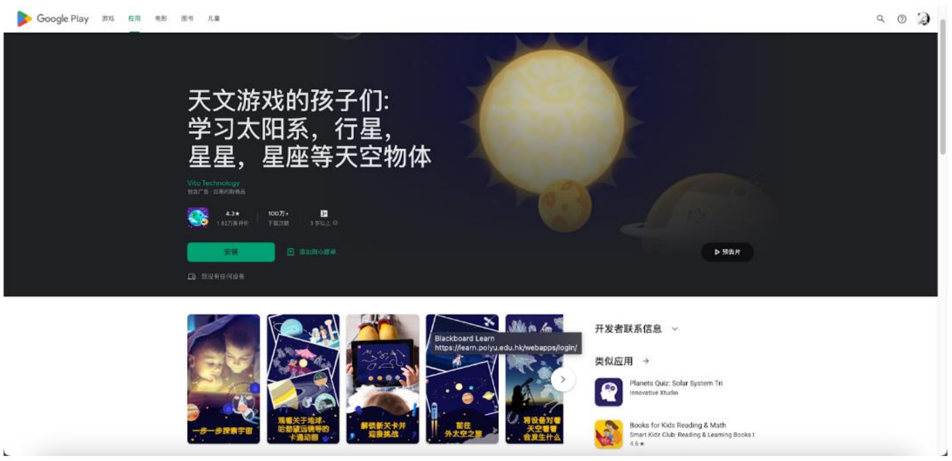
Directory

Abstract

AMA1D07 is the class which offers students a general understanding, at an elementary level, of cosmology from the observational and theoretical perspectives. However, because of the large number of students at the university, the university was unable to provide enough instruments, resulting in a course that had to focus more on the theoretical part at the expense of the observational part. Our group's project is a solar system simulator developed on the unity platform. It is used to support the teaching of the course AMA1D07, to visualize the knowledge from the book, and to supplement the course in guiding students' observation.This report will describe in detail the functionality and development of our software

Background and objectives

Background



“Star Walk” is a 2D interactive educational software for children, which developed by Vito Technology and available on mobile platform.In this app, kids can learn about the eight planets of our solar system, the 49 constellations and the 700 brightest stars, as well as the location of the International Space Station and the story of the Hubble Telescope. The software shows children the planets or constellations. They want to learn about 2D pictures and audio descriptions.

Our group will use this app as the reference application of our project. We are very happy to reserve the right for users to view various planets and galaxies, but we think that 2D pictures are not enough. We will use the 3D model to allow users to observe the orbit and position of the planet more intuitively. In addition, the user can move or zoom in and out to observe the movement of the planet from the angle the user wants. Users can even move certain planets to observe certain astronomical phenomena, such as a total solar eclipse. We made these changes to allow users to observe the solar system more intuitively.

Objectives

a. Design a 3D solar system simulator to help school which offer cosmology class.

b. Make our project can run on mobile(android) platform.

c. Good environment design and character design.

d. Simplify the operation, so that users do not need to spend a lot of time learning how to use the software, so that the software can be more easily accepted by teachers.

Functional Specifications

Introduction

The Solar System Simulator is a software developed using Unity that provides users with various features including realistic gravitational simulations, orbital displays, and the ability to add additional planets to observe their effects on the existing system. Users can also adjust the game speed and camera angle to enhance their overall experience.

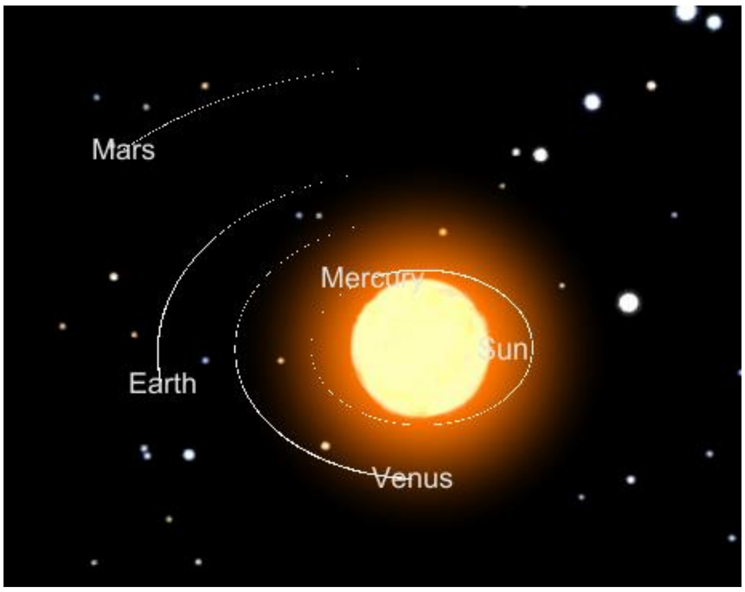
Functional Requirements

Gravitational Simulations

The Solar System Simulator will provide realistic gravitational simulations, accurately modeling the real-life effects of gravity on the planets in the system. This will include the ability to display the force of gravity between planets in real-time.

Orbital Displays

The Solar System Simulator will provide users with a detailed graphical representation of the planets and their respective orbits. This will include the ability to adjust the scale and size of the planets to better observe their movements.



Trails shown behind the planets

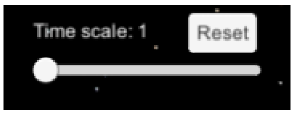
Add Planets

Users will be able to add additional planets to the existing system, observing the effects of different planets and their gravitational interactions with the other planets.

Different planets

Game Speed Adjustment

The Solar System Simulator will allow users to adjust the game speed to simulate different time scales, from real-time to faster or slower speeds.



User can use this slider to change the game speed.

Camera Angle Adjustment

Users will be able to adjust the camera angle to view the solar system from different perspectives, including zooming in on individual planets or viewing the system from a top-down perspective.

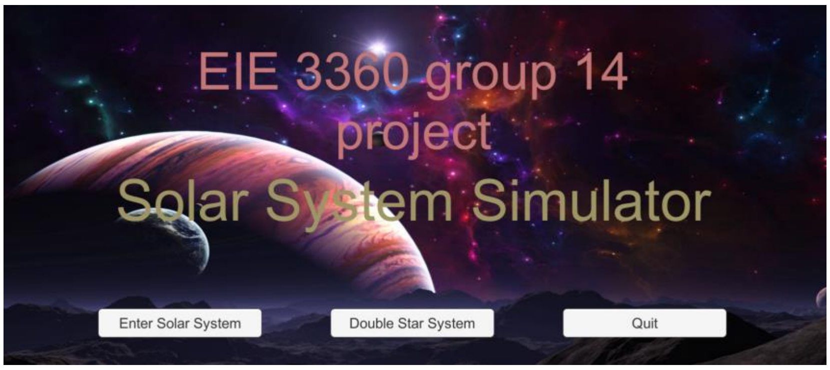
Non-Functional Requirements

Performance

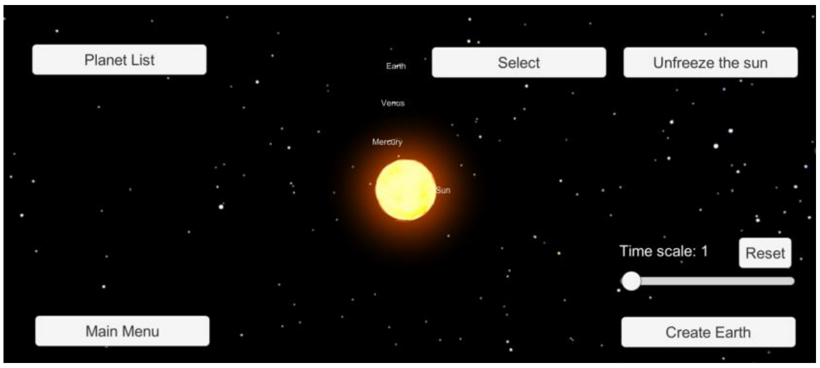
The Solar System Simulator should have fast and responsive performance, even with a large number of planets in the system.

User Interface

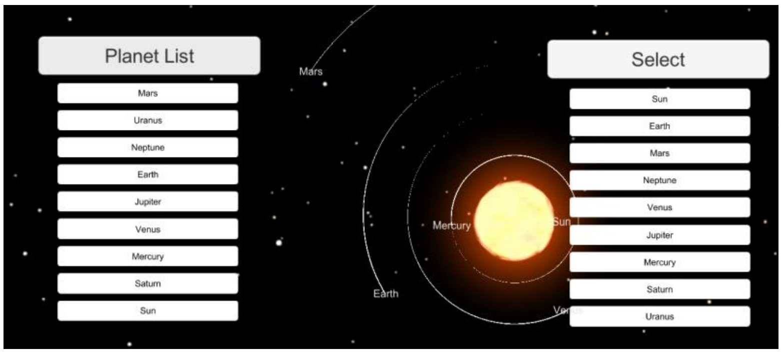
The user interface should be intuitive and easy to navigate, with clear labels and descriptions for each feature.



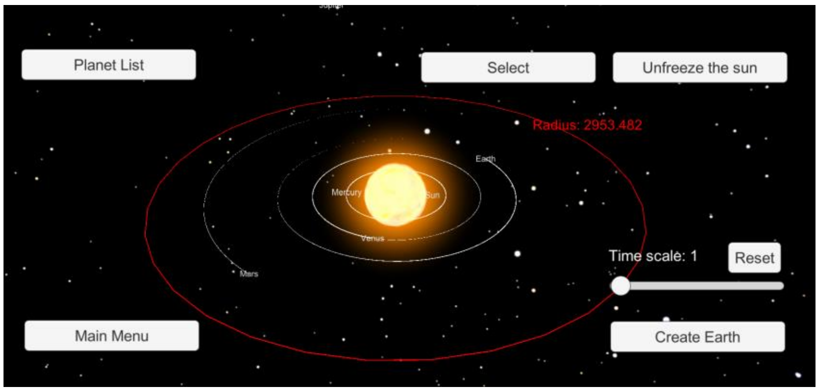
This is the menu UI of our project.



After entering a scene, the UI display is very clear.



After clicking select or planet list, this list will show up.



After clicking Create Earth, a circle will indicate the position of your planet and shows the radius.

Compatibility

The Solar System Simulator should be compatible with a wide range of hardware and operating systems.

Test Cases

Gravitational Simulations Test

To ensure the accuracy of the gravitational simulations, a test case will be created to compare the simulator's results with real-life observations.

Interface Usability Test

A usability test will be conducted to evaluate the ease of use and understandability of the user interface.

Performance Test

A performance test will be conducted to evaluate the simulator's performance under various conditions, including the number of planets in the system and the game speed.

Conclusion

The Solar System Simulator is a feature-rich software that provides users with a realistic and engaging simulation of the solar system. With accurate gravitational simulations, detailed orbital displays, and customizable options for adding planets and adjusting game speed and camera angle, the simulator offers a comprehensive and immersive experience for users.

Technical implementation

Core Mechanism

The core mechanism of our project consists of the following three parts:

1. Gravity system:

The most important aspect of simulating a universe system is to model the universal gravity. To achieve this, we use the RigidBody component in Unity. Firstly, to disable its own gravity, we uncheck the "use gravity" option so that our planets are not affected by gravity. Then, we write a new script to implement our gravity system. We first label all celestial bodies with the "Celestial" tag. During each update of the game, we enumerate all of the celestial objects and apply the gravitational force calculated based on the universal gravity formula. The RigidBody component automatically updates the objects' trajectories, velocities, and other information based on the force applied to each object.

Universal Gravitational Formula:

F = G * m1 * m2 / r^2

where F is the gravitational force between two objects, G is the universal gravitational constant, m1 and m2 are the masses of the two objects, and r is the distance between the two objects.

Our Code:

```
    void Gravity(){
        celestials = GameObject.FindGameObjectsWithTag("Celestial");
        foreach(GameObject a in celestials){
            foreach(GameObject b in celestials){
                if(!a.Equals(b)){
                    float m1 = a.GetComponent<Rigidbody>().mass;
                    float m2 = b.GetComponent<Rigidbody>().mass;
                    float r = Vector3.Distance(a.transform.position,
b.transform.position);
                    a.GetComponent<Rigidbody>().AddForce((b.transform.po
sition - a.transform.position).normalized *
                    (G * (m1 * m2) / (r * r)));
                }
            }
        }
    }
```

In the above code, we are finding all game objects tagged with "Celestial" and iterating through them with two nested loops to calculate gravitational force between each pair of objects. We are using the Vector3.Distance function to calculate the distance between the two objects and the AddForce function to apply the gravitational force to each object.

G represents the universal gravitational constant, and m1 and m2 represent the mass of the two objects. We are using the normalized difference vector between the two objects' positions to find the direction of the gravitational force, and multiplying it by the magnitude of the force calculated using the universal gravitational formula.

2. Initial Velocity Caculation:

   In the Solar System, all celestial bodies orbit the sun in approximately circular motion. This fact can be used to calculate the initial velocity of all celestial bodies. If a celestial body loses its initial velocity, it will only approach and collide with other stationary celestial bodies due to gravitational force. Therefore, calculating initial velocity is necessary to simulate the Solar System realistically.

Based on the formula for uniform circular motion (f = mv^2 / r) and the universal gravitational formula, we can derive the initial velocity (The derivation process is mentioned in the following content.) required for each celestial body to undergo uniform circular motion. In our program, we calculate the initial velocity information for each celestial body by iterating through each of the celestial bodies and applying the derived initial velocity to them. The following code shows how we implemented this:

```
void InitialVelocity(){
        foreach(GameObject a in celestials){
            foreach(GameObject b in celestials){
                if(!a.Equals(b)){
                    float m2 = b.GetComponent<Rigidbody>().mass;
                    float r = Vector3.Distance(a.transform.position,
b.transform.position);
                    a.transform.LookAt(b.transform);

                    a.GetComponent<Rigidbody>().velocity +=
a.transform.right * Mathf.Sqrt((G * m2) / r);
                }
            }
        }
    }
```

In the above code, we are iterating through each celestial body and calculating its initial velocity by applying the derived formula. We are using the Transform.LookAt() function to orient the celestial body towards the central mass (the sun), and using the Rigidbody.velocity property to apply the calculated initial velocity in the direction perpendicular to the celestial body's forward direction.

m2 represents the mass of the central mass (the sun), and r represents the distance between the celestial body and the central mass. We are using Mathf.Sqrt() to calculate the square root of the expression inside the formula for initial velocity. Overall, this code implements a system for calculating initial velocities for celestial bodies in the Solar System.

The derivation of the initial speed:

Starting with the formula for gravitational force between two objects:

F = G * m1 * m2 / r^2

We can use this formula to derive the velocity required for an object to maintain uniform circular motion around a central point.

In circular motion, the centripetal force required to keep an object in circular motion is given by:

$F = m * v^2 / r$

Equating the two equations for force, we get:

$G * m1 * m2 / r^2 = m * v^2 / r$

Simplifying and solving for v, we get:

$v^2 = G * m2 / r$

$v = sqrt(G * m2 / r)$

Therefore, the velocity required for an object to maintain uniform circular motion around a central point, where G is the universal gravitational constant, m2 is the mass of the central object, and r is the radius of the circular path, is given by the equation v=sqrt(Gm2/r).

3. Collision System

Planetary collisions frequently occur in celestial systems. When planets collide, the material between them is exchanged, mixed, and redistributed. If the masses of two planets are equal, their masses will merge after collision, so the total mass will be equal to the sum of the masses of the two planets. However, if the masses of two planets are different, the smaller planet will be shattered into pieces and absorbed by the larger planet, increasing its mass. Therefore, in order to simulate a planetary collision system, we simply add the masses of the colliding planets together. After a planetary collision, the resulting velocity can be calculated using the law of conservation of momentum. Therefore, we have written the following code:

```
    private void OnCollisionEnter(Collision collision){
        if (collision.gameObject.CompareTag("Celestial")){
            Rigidbody otherRb =
collision.gameObject.GetComponent<Rigidbody>();
            float newMass = rb.mass + otherRb.mass;
            if (rb.mass > otherRb.mass){
                rb.velocity = (rb.mass * rb.velocity + otherRb.mass *
otherRb.velocity) / newMass;
                Vector3 initialScale = transform.localScale;
                Vector3 newScale = initialScale * Mathf.Pow(newMass /
rb.mass, 1f / 3f);
                transform.localScale = newScale;
                rb.mass = newMass;
                Destroy(collision.gameObject);
            }
        }
    }
```

When a planet collides with another "Celestial" game object, the code calculates the total mass of the planets after the collision and compares the masses of the two planets. If the mass of the current planet is greater than the other planet, it absorbs the other planet, calculates the new velocity and scale, updates the mass, and destroys the other game object. If the mass of the current planet is less than or equal to that of the other planet, no collision occurs.

Features

1. Panel and Label Settings

We use panels to display various information about the star, and add labels on the edge of each celestials bodies. The following is our code:

```
    void OnGUI()
    {
        if (true)
        {
            RaycastHit hit;
            if (Physics.Linecast(transform.position,
camera.transform.position, out hit, ~(1 <<
LayerMask.NameToLayer("Celestial")))){
                return;
            }

            Vector3 worldPosition = new Vector3 (transform.position.x,
transform.position.y,transform.position.z);

            Vector2 position = camera.WorldToScreenPoint(worldPosition);

            var distance = Vector3.Distance(transform.position,
camera.transform.position);

            //get 2d position

            var frustumHeight = 2.0f * distance *
Mathf.Tan(camera.fieldOfView * 0.5f * Mathf.Deg2Rad);
            var frustumWidth = frustumHeight * camera.aspect;
            var true_width = 1.0f * Screen.width * radius /
frustumWidth;

            Vector2 labelSize = GUI.skin.label.CalcSize (new
GUIContent(this.name));

            position = new Vector2 (position.x + true_width,
Screen.height - position.y + (labelSize.y / 2.0f));
            GUI.color = Color.white;
            GUI.skin.label.fontSize = 18;//font size

            GUI.Label(new Rect(position.x - (labelSize.x/2),position.y -
labelSize.y ,labelSize.x,labelSize.y), this.name);

        }
        if (WindowShow){
            RaycastHit hit;
            if (Physics.Linecast(transform.position,
camera.transform.position, out hit, ~(1 <<
LayerMask.NameToLayer("Celestial")))){
                return;
            }
            Vector3 worldPosition = new Vector3 (transform.position.x,
transform.position.y,transform.position.z);

            Vector2 position = camera.WorldToScreenPoint(worldPosition);

            var distance = Vector3.Distance(transform.position,
camera.transform.position);

            //get 2d position

            var frustumHeight = 2.0f * distance *
Mathf.Tan(camera.fieldOfView * 0.5f * Mathf.Deg2Rad);
            var frustumWidth = frustumHeight * camera.aspect;
            var true_width = 1.0f * Screen.width * radius /
frustumWidth;
            position = new Vector2 (Mathf.Min(position.x + true_width,
Screen.width - windowSize.x), Mathf.Min(Screen.height - position.y +
(50 / 2.0f), Screen.height - windowSize.y));
            GUI.Window(0, new Rect(position.x, position.y, windowSize.x,
windowSize.y), MyWindow, this.name);
        }
    }

    void MyWindow(int WindowID){
        GUILayout.Label("Position: " + (transform.position -
GameObject.Find("Sun").transform.position));
        GUILayout.Label("Radius: " + transform.localScale.x * 2000f);
        GUILayout.Label("Mass(earth): " + rb.mass);
        GUILayout.Label("Velocity: " + rb.velocity.magnitude);

        if(GUILayout.Button("Close")){
            WindowShow = false;
        }
    }
    void OnMouseDown() // toggle the visibility of the panel when the
planet is clicked
    {
        if (WindowShow)
            WindowShow = false;
        else
            WindowShow = true;
    }
```

This code is responsible for displaying the name of the planet using GUI labels and panels. It first checks if the planet is being obstructed by other objects using Physics.Linecast, and then calculates the position of the planet on the screen using camera.WorldToScreenPoint. The size of the planet is then calculated based on the distance and field of view of the camera, and a label with the name of the planet is displayed at the calculated position.

If the WindowShow variable is true, a panel is displayed with information about the planet's position, radius, mass, and velocity. The panel is created using GUI.Window, and the information is displayed using GUILayout.Label. Additionally, a button with the label "Close" is displayed, which sets WindowShow to false when clicked, hiding the panel.

Finally, the OnMouseDown function is used to toggle the visibility of the panel when the planet is clicked. If WindowShow is true, it is set to false, and if it is false, it is set to true.

There is also another detail:

```
            position = new Vector2 (Mathf.Min(position.x + true_width,
Screen.width - windowSize.x), Mathf.Min(Screen.height - position.y +
(50 / 2.0f), Screen.height - windowSize.y));
```

The purpose of this line of code is to calculate the position of the panel. The variable "position" represents the position of the planet on the screen, while "true_width" represents the actual size of the planet on the UI. "Screen.width" and "Screen.height" represent the width and height of the screen, respectively, and "windowSize.x" and "windowSize.y" represent the width and height of the panel.

Therefore, this line of code restricts the position of the panel within the screen and places it near the edge of the planet. Specifically, the "Mathf.Min" function is used to compare two values and return the smaller value. "50 / 2.0f" represents the height of the label text.

2. List display

To put all planets in the scene into a list after clicking on the planet list, we used the following code to achieve this:

```
    void OnMenuButtonClick()
    {
        celestials = GameObject.FindGameObjectsWithTag("Celestial");
        foreach (Transform child in planetList.transform)
            Destroy(child.gameObject);

        foreach(GameObject a in celestials){

            // create the button
            GameObject buttonObject = new GameObject(a.name + "Button");
            RectTransform rectTransform =
buttonObject.AddComponent<RectTransform>();
            Button button = buttonObject.AddComponent<Button>();
            Image buttonImage = buttonObject.AddComponent<Image>();

            // set the position and size of the button
            rectTransform.SetParent(planetList.transform);
            rectTransform.anchoredPosition = new Vector2(0, 0);
            //rectTransform.sizeDelta = new
Vector2(planetList.GetComponent<RectTransform>().rect.width, 20);
            Vector2 size =
GameObject.Find("ListButton").GetComponent<RectTransform>().sizeDelta;
            // Debug.Log(size);
            size = Vector2.Scale(size, new Vector2(Screen.width / 800f,
Screen.height / 600f));
            // Debug.Log(size);
            rectTransform.sizeDelta = Vector2.Scale(size, new
Vector2(0.8f, 0.8f));

            // set the background color
            buttonImage.sprite = roundRectSprite;
            buttonImage.type = Image.Type.Sliced;

            // create button text
            GameObject buttonTextObject = new GameObject(a.name +
"Text");
            RectTransform buttonTextRectTransform =
buttonTextObject.AddComponent<RectTransform>();
            Text buttonText = buttonTextObject.AddComponent<Text>();

            // set the text of the buton
            buttonText.text = a.name;
            buttonText.font =
Resources.GetBuiltinResource<Font>("Arial.ttf");
            buttonText.fontSize = 15;
            buttonText.color = Color.black;
            buttonText.alignment = TextAnchor.MiddleCenter;

            // set the position and size of the text
            buttonTextRectTransform.SetParent(rectTransform);
            buttonTextRectTransform.anchoredPosition = Vector2.zero;
            buttonTextRectTransform.sizeDelta = rectTransform.sizeDelta;

            // attach the text to button
            buttonText.transform.SetParent(buttonObject.transform,
false);

            //buttonImage.color = Color.white;

            button.onClick.AddListener(delegate
{ OnPlanetButtonClick(a.name); });

            buttonObject.transform.SetParent(planetList.transform,
false);
        }


        planetList.SetActive(!planetList.activeSelf);
    }
    void OnPlanetButtonClick(string planetName)
    {
        GameObject planet = GameObject.Find(planetName);
        if (planet != null){
            camera.pivot = planet.transform;
            camera.target = planet.transform;
            camera.setTargetDistance(planet.transform.localScale.x *
6f);
        }
        planetList.SetActive(false);
    }
```

This code creates a list of buttons for all the planets in the scene when the planet list button is clicked. The code iterates through each object with the "Celestial" tag and creates a new game object with a button component for each planet. The position and size of the button are set by creating a new RectTransform component and setting the values accordingly. The button label is created by adding a new Text component to the button game object and setting the text and font properties. Finally, an event listener is added to the button to call the OnPlanetButtonClick function when clicked.

The OnPlanetButtonClick function finds the corresponding planet game object based on the clicked button's name and sets the camera's pivot and target to the planet. The setTargetDistance function is used to set the camera's distance from the planet based on the planet's scale. Finally, the planet list is hidden by setting its active state to false.

3. Camera movement

User can control the camera's movement and rotation in the scene. The camera can either be locked onto a target object or be freely moved around. The code allows the camera to zoom in and out, rotate around the target object, and tilt up and down. The camera distance, zoom speed, and rotation speed can all be adjusted in the code.

```
using UnityEngine;

public class CameraControl : MonoBehaviour
{
    public Transform pivot; // Manually added: Object to be tracked: 
pivot - what to use as pivot
    public Vector3 pivotOffset = Vector3.zero; // Offset from target
    public Transform target; // Object that is being focused on (used to 
check for objects between cam and target)
    public float distance = 10.0f; // Distance from target (use zoom)
    public float minDistance = 2f; // Minimum distance
    public float maxDistance = 200000; // Maximum distance
    public float zoomSpeed = 100f; // Speed multiplier
    public float xSpeed = 250.0f; // X speed
    public float ySpeed = 120.0f; // Y speed
    public bool allowYTilt = true; // Allow Y tilt
    public bool free = false; // Allow free movement of the camera
    public float speed = 5000f; // Free movement speed
    public float yMinLimit = -90f; // Maximum downward angle of the 
camera
    public float yMaxLimit = 90f; // Maximum upward angle of the camera
    private float x = 0.0f; // X variable
    private float y = 0.0f; // Y variable
    private float targetX = 0f; // Target X
    private float targetY = 0f; // Target Y
    private float targetDistance = 0f; // Target distance
    private float xVelocity = 1f; // X velocity
    private float yVelocity = 1f; // Y velocity
    private float zoomVelocity = 1f; // Zoom velocity

    void Start()
    {
        var angles = transform.eulerAngles;  // Current euler angles
        targetX = x = angles.x; // Set x and target x
        targetY = y = ClampAngle(angles.y, yMinLimit, yMaxLimit); // 
Clamp camera between up and down angle and set y and target y
        targetDistance = distance; // Starting distance data is 10
    }

    public void setTargetDistance(float f){
        targetDistance = f;
    }

    void LateUpdate()
    {
        if (pivot) // If there's a target
        {
            float scroll = Input.GetAxis("Mouse ScrollWheel");
            float tmp = distance -
target.GetComponent<Collider>().bounds.size.x;
            zoomSpeed = Mathf.Clamp(tmp * 0.1f,10f,10000f);

            // Zoom on touch devices
            if (Input.touchCount == 2){
                Touch touchZero = Input.GetTouch(0);
                Touch touchOne = Input.GetTouch(1);

                Vector2 touchZeroPrevPos = touchZero.position -
touchZero.deltaPosition;
                Vector2 touchOnePrevPos = touchOne.position -
touchOne.deltaPosition;

                float prevTouchDeltaMag = (touchZeroPrevPos -
touchOnePrevPos).magnitude;
                float touchDeltaMag = (touchZero.position -
touchOne.position).magnitude;

                float deltaMagnitudeDiff = prevTouchDeltaMag -
touchDeltaMag;

                targetDistance += deltaMagnitudeDiff * zoomSpeed;
                targetDistance = Mathf.Clamp(targetDistance,
minDistance, maxDistance);
            }

            // Zoom using mouse wheel
            if (scroll > 0.0f) targetDistance -= zoomSpeed;
            else if (scroll < 0.0f) targetDistance +=zoomSpeed;

            // Set minimum distance to twice the radius of the pivot
game object's sphere collider
            minDistance =
pivot.gameObject.GetComponent<SphereCollider>().radius * 2.0f;

            targetDistance = Mathf.Clamp(targetDistance, minDistance,
maxDistance);

            // Rotate camera on mouse right click or left click with
ctrl key pressed
            if (Input.GetMouseButton(1) || Input.GetMouseButton(0) &&
(Input.GetKey(KeyCode.LeftControl) ||
Input.GetKey(KeyCode.RightControl))){
```

```
                targetX += Input.GetAxis("Mouse X") * xSpeed * 0.02f;
                if (allowYTilt){
                    targetY -= Input.GetAxis("Mouse Y") * ySpeed *
0.02f;
                    targetY = ClampAngle(targetY, yMinLimit, yMaxLimit);
                }
            }

            // Use smoothing interpolation
            x = Mathf.SmoothDampAngle(x, targetX, ref xVelocity, 0.3f);
            if (allowYTilt) y = Mathf.SmoothDampAngle(y, targetY, ref
yVelocity, 0.3f);
            else y = targetY;
            Quaternion rotation = Quaternion.Euler(y, x, 0);
            distance = Mathf.SmoothDamp(distance, targetDistance, ref
zoomVelocity, 0.5f);
            Vector3 position = rotation * new Vector3(0.0f, 0.0f, -
distance) + pivot.position + pivotOffset;
            transform.rotation = rotation;
            transform.position = position;
        }
    }
    // Clamp the angle between min and max values
    private float ClampAngle(float angle, float min, float max)
    {
        if (angle < -360) angle += 360;
        if (angle > 360) angle -= 360;
        return Mathf.Clamp(angle, min, max);
    }
}
```

This code implements a camera controller that allows the camera to rotate around a tracked object and control the view through scaling, rotating, and moving the camera. In each frame, the camera's position and orientation are updated based on user inputs, and the camera smoothly moves to the target position using a smooth interpolation algorithm. The camera controller includes several adjustable parameters such as the distance between the camera and the target object, rotation speed, zoom speed, etc., which can be customized by the user as needed.

In summary, this code controls the camera movement and zoom level in the scene. It provides options to adjust the camera's speed and rotation. The code also includes touch support for zooming in and out when using a touch screen device. Overall, this code provides a flexible camera control system for navigating through a 3D scene.

4. Freezing the sun

This function is quite simple, after click the “freeze the sun” button, we just set the velocity of the sun to Vector3.zero in every update.

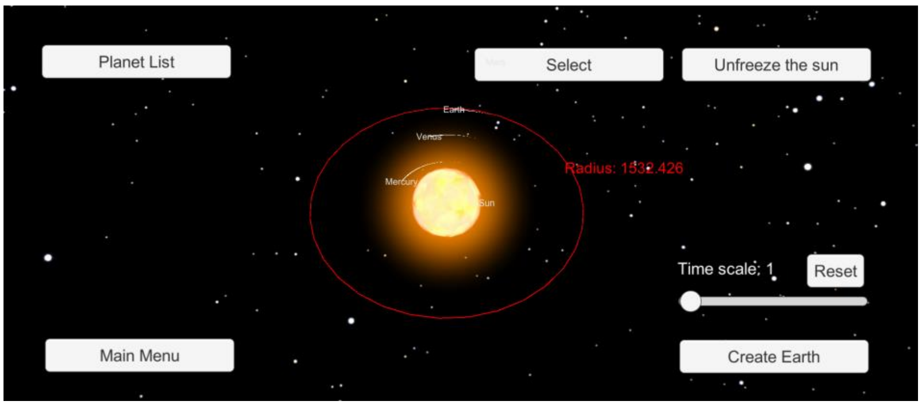
5. Game speed manager

The game speed manager is another simplest function, it uses the SetGameSpeed function provided by unity. When the value of the slider changed, this function will be used to change the game speed accordingly.

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

public class GameSpeedController : MonoBehaviour
{
    public Slider speedSlider;
    public Text text;
    private void Start() {
        speedSlider.onValueChanged.AddListener(delegate
{ValueChangeCheck ();});
        speedSlider.value = 1;

    }
    public void SetGameSpeed(float speed)
    {
        Time.timeScale = speed;
    }
    private void Update() {
        text.text = "Time scale: " + speedSlider.value;
    }

    public void ValueChangeCheck(){
        SetGameSpeed(speedSlider.value);
    }
}
```

6. Creating planet

The creating part is quite simple, just create a new object at target position, the difficult part is to calculating the position on the solar panel, and showing the preview circle on the canvas.



```
    void Update()
    {
        if(!addPlanet){
            lineRenderer.enabled = false;
            return;
        }
        lineRenderer.enabled = true;
        // Calculate the normal of the plane using the cross product of
two vectors on the plane
        normal = Vector3.Cross(pointC.position - pointA.position,
pointB.position - pointA.position).normalized;
```

```
        // Cast a ray from the camera to the mouse position on the
screen
        Vector3 mousePosition = Input.mousePosition;
        float rayDistance;
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

        // If the ray intersects with the plane, calculate the hit point
and the radius of the circle
        if (new Plane(normal, planePoint).Raycast(ray, out rayDistance))
        {
            Vector3 hitPoint = ray.GetPoint(rayDistance);
            radius = Vector3.Distance(hitPoint, pointA.position);
            Vector3 u, v;
            GetBasisVectors(normal, out u, out v);

            // Calculate the positions of the points on the circle
            for (int i = 0; i <= segments; i++){
                float angle = Mathf.PI * 2f * i / segments;
                Vector3 pointOnCircle = pointA.position + radius *
(Mathf.Cos(angle) * u + Mathf.Sin(angle) * v);
                positions[i] = pointOnCircle;
            }

            // Update the line renderer with the new positions of the
points
            lineRenderer.positionCount = positions.Length;
            lineRenderer.SetPositions(positions);

            // Set the material and color of the line renderer
            lineRenderer.material = new
Material(Shader.Find("Sprites/Default"));
            lineRenderer.material.color = color;
        }
    }

    private void OnGUI() {
        if(!addPlanet) return;
        // Otherwise, create a new GUIStyle for the label and display
the radius of the circle on the screen
        GUIStyle style1= new GUIStyle();
        style1.fontSize = 30;
        style1.normal.textColor = Color.red;
        GUI.Label(new Rect(Input.mousePosition.x, Screen.height -
Input.mousePosition.y, 400, 50),"Radius: " + radius, style1);
```

```
}
```

This code implements a script for drawing a circle on a plane in Unity. It contains several adjustable parameters, such as radius, line width, color, number of segments, etc.

To draw a circle on the plane, the script first calculates the radius by measuring the distance between the mouse click position and the center of the circle. The script then calculates the positions of the points on the circle using the radius and stores them in an array. Finally, the circle is drawn using the LineRenderer component in Unity.

To calculate the center of the circle, the script first casts a ray in the direction of the camera from the mouse click position and calculates the point of intersection with the plane. To calculate the positions of the points on the circle, the script first calculates the normal of the plane using the cross product of two vectors on the plane. Then, it calculates two perpendicular vectors using the normal vector and the up vector. Finally, the positions of the points on the circle are calculated using the radius and the two perpendicular vectors.

The script also includes additional logic, such as controlling whether to draw the circle based on user inputs and displaying the radius of the circle on the screen.

To draw the circle, the LineRenderer component in Unity is used to draw the lines between the points on the circle. The width and color of the lines can be adjusted using the component's parameters.

To display the radius of the circle on the screen, OnGUI function is used to draw a GUI element on the screen.

The code about creating a new planet:

```
    // Create a ray from the mouse position on the screen to the scene.
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

    // Cast a ray from the camera through the mouse position and check
for a collision with an object.
    RaycastHit hit;
    if (Physics.Raycast(ray, out hit))
    {
        Debug.Log("Clicked on " + hit.collider.gameObject.name + " in
scene.");
    }
    else
    {
        // If no object is hit, create a new object on the plane.
        Vector3 normal = new Vector3(0f, 1f, 0f);
        Vector3 mousePosition = Input.mousePosition;
        float rayDistance;
        // Create a new ray from the camera through the mouse position
and check for a collision with the plane.
        ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        if (new Plane(normal,
GameObject.Find("Sun").transform.position).Raycast(ray, out
rayDistance))
        {
            // Get the point where the ray intersects the plane and
instantiate a new prefab at that point.
            Vector3 hitPoint = ray.GetPoint(rayDistance);
            GameObject.Find("CircleOnPlane").GetComponent<CircleOnPlane>
().addPlanet = false;
            chk = false;
            GameObject spawnedPrefab = Instantiate(prefab, hitPoint,
Quaternion.identity);

            // Set the camera of the new object to the main camera.
            spawnedPrefab.GetComponent<Label>().camera =
GameObject.Find("Main Camera").GetComponent<Camera>();
        }
    }
```

Details

1. Trail

   Setting the size of the trail was initially very challenging because a trail that was too narrow would not be visible when the camera was too far away, while a trail that was too wide would obstruct the planet when the camera was too close. To solve this problem, we used dynamic trail adjustment techniques. The following is the relevant code.

```
    private void Update() {
        if(!camera) camera = GameObject.Find("Main
Camera").GetComponent<Camera>();
        if(this.name == "Sun" || this.name == "Sun(Clone)") return;
        var dis = Vector3.Distance(camera.transform.position,
transform.position) - radius;
        var width = Mathf.Min(dis / 100f, 1f);
        // Get the widthCurve property of the TrailRenderer.
        if(!trailRenderer) return;
        AnimationCurve widthCurve = trailRenderer.widthCurve;

        // Find the keyframe with a time value equal to widthCurveTime.
        int keyframeIndex = -1;
        for (int i = 0; i < widthCurve.length; i++){
            if (Mathf.Approximately(widthCurve.keys[i].time,
widthCurveTime)) {
                keyframeIndex = i;
                break;
            }
        }
        // If the keyframe is found, set its value to the maximum width.
        if (keyframeIndex >= 0) {
            Keyframe keyframe = widthCurve.keys[keyframeIndex];
            keyframe.value = width;
            widthCurve.MoveKey(keyframeIndex, keyframe);
        }
        // Otherwise, create a new keyframe with its value set to the
maximum width.
        else{
            float widthCurveValue = widthCurve.Evaluate(widthCurveTime);
            Keyframe keyframe = new Keyframe(widthCurveTime,
Mathf.Max(widthCurveValue, width));
            widthCurve.AddKey(keyframe);
        }
        // Assign the modified curve to the TrailRenderer's widthCurve
property.
        trailRenderer.widthCurve = widthCurve;
    }
```

This code implements a feature that dynamically adjusts the width of a TrailRenderer based on the distance between the camera and the object.

First, the code checks if the camera exists. If it does not exist, the code uses the GameObject.Find function to find the camera object. The code then calculates the distance between the camera and the object and uses this distance to calculate the width of the TrailRenderer.

The code uses an AnimationCurve to set the width of the TrailRenderer. It first gets the widthCurve property of the TrailRenderer. The code then searches for a keyframe with a time value equal to widthCurveTime in the widthCurve. If it finds the keyframe, it sets its value to the width of the TrailRenderer. If it does not find the keyframe, it creates a new keyframe with its value set to the width of the TrailRenderer. Finally, the modified curve is assigned to the TrailRenderer's widthCurve property to achieve dynamic TrailRenderer width adjustment.

2. Handling of label when occlusion occurs

```
                if (Physics.Linecast(transform.position,
camera.transform.position, out hit, ~(1 <<
LayerMask.NameToLayer("Celestial")))){
```

```
                // Debug.Log("blocked:"+hit.transform.name);
                return;
            }
```

This line of code uses Physics.Linecast to determine if a planet is occluded by other planets (in other words, if it can be seen by the camera). If it is occluded, its label will not be displayed.

The impact on individuals, organizations, and society locally and globally

This solar system simulator is a very useful multimedia software that can impact individuals, organizations, and society in many ways. It brings new possibilities for teaching and provides students with an interactive and visual learning experience. For individual students, it helps to make abstract knowledge of the universe concrete and vivid, allowing students to better understand the workings and structure of the solar system. It can bring changes and enhancements to school teaching and learning, providing students with a more interesting, richer, and more intuitive learning experience.

In addition to the personal and organizational aspects, this simulator also has social impacts. This solar system simulator can have a wide impact on the general public. Users can use this software to observe the planets in our solar system, as well as other phenomena in the universe. People can learn about some of the mechanisms of how the universe works and discover the beauty and mystery of space. This will not only stimulate people's curiosity and desire to explore the universe but also enhance public interest in the universe and astronomy, and promote awareness of the universe. This will help improve the level of public knowledge about the universe and increase people's support for cosmic exploration, thus opening up a broader prospect for cosmic exploration. Overall, this solar system simulator is a powerful tool for education and exploration, with the potential to impact individuals, organizations, and society locally and globally.

Difficulties encountered

Code understanding

We often have difficulty understanding each other's code, for several reasons:

Different coding styles: Different people have different coding styles, including indentation, commenting, variable naming, etc. This can make it difficult for us to understand and maintain other people’s code.

Lack of documentation: If there is a lack of documentation, comments, or the code is not easily readable, other members will find it difficult to understand the code's intent and functionality.

Different skill levels: Because we have different levels of programming skills.

To address these issues, we made these attempts:

A unified coding style: We developed a coding standard that includes indentation, commenting, variable naming, etc., so that everyone can follow the same coding standard, reducing issues with code readability.

Increased commenting and documentation: We wrote clear and easy-to-understand comments and documentation so that other members can more easily understand the code's intent and functionality, providing better readability for the entire team.

Code review: We use code review tools to review code written by other team members to identify potential issues and provide suggestions. This can help the us better understand and manage the code, improving its readability and maintainability.

Button size setting

The newly created buttons in the planet list cannot adjust their size according to the screen scaling, and their size remains the same across different resolutions, causing an unattractive appearance.

Solution: I used the size of the parent element (Vertical Layout Group) and the current screen ratio to calculate the appropriate size. Since I was debugging the program on an 800/600 canvas and scaling it proportionally, the calculated size should be:

```
           Vector2 size =
GameObject.Find("ListButton").GetComponent<RectTransform>().sizeDelta;
            // Debug.Log(size);
            size = Vector2.Scale(size, new Vector2(Screen.width / 800f,
Screen.height / 600f));
            // Debug.Log(size);
            rectTransform.sizeDelta = Vector2.Scale(size, new
Vector2(0.8f, 0.8f));
```

At the end I scale the result to 0.8x, to make the button look nicer.

Teamwork

Our team collaboration was very close and efficient. Before starting the project, we conducted requirement analysis and discussion to determine the project goals and functionality, and assigned tasks to each member of the team. Throughout the development process of the project, we adopted an agile development approach, holding regular meetings to discuss progress and any issues encountered. We also used online collaboration tools such as WeChat, QQ, Unity Collaborate and Git for better collaboration and communication.

During the project implementation, our team members worked closely together, each fulfilling their respective roles and helping each other out. Programmers were responsible for code writing and testing, designers created UI interfaces and conducted graphic design, and testers performed testing and quality control. We also maintained frequent iterations and feedback to ensure product quality and user experience.

After the entire project development was completed, we also reviewed and summarized the project to improve and optimize our team collaboration approach for better work on future projects. Through our team collaboration and cooperation, we successfully completed the project and created a high-quality solar system simulation program.

Testing result

1. Gravitational Simulations Test

The gravitational system performs well. In the solar system scene, planets can move stably according to the laws of physics. And in an unstable system, the orbit of the planets will also change according to the gravity.

2. Interface Usability Test

The UI performs normally and there are no interaction problems.

3. Performance Test

The program performs well on Android platform (Google Pixel 5) and PC platform. It can support over 50 planets on Android platform or allow the game to run at over 30 times speed.

References

[1] Universe Sandbox, universesandbox.com, accessed on Mar. 29, 2023. [Online]. Available: https://universesandbox.com/

[2] Coderious, " Unity Tutorial - Make a STAR in Shader Graph, " YouTube, Mar. 29, 2023. [Online]. Available: https://www.youtube.com/watch?v=ykwvCCqdcCs&t=22s&ab_channel=Coderious

[3] Coderious, " Unity Tutorial - Solar System with Unity Physics" YouTube, Mar. 29, 2023. [Online]. Available: https://www.youtube.com/watch?v=kUXskc76ud8&ab_channel=Coderious

[4] 梦幻 DUO, " Unity 用 GUI 在角色头上显示名字（C#脚本）" CSDN，Mar. 29, 2023. [Online]. Available: https://blog.csdn.net/sinat_24229853/article/details/42590175

[5] hanguangfei, "Unity 学习笔记——鼠标移动到物品上显示物品名字，点击后显示物品信息" CSDN， Mar. 29, 2023. [Online]. Available: https://blog.csdn.net/HanGuangFei/article/details/78094214

[6] Khanrad Coder, "Universe Simulator," khanradcoder.github.io, accessed on Sep. 29, 2023. [Online]. Available: https://khanradcoder.github.io/UniverseSimulator/