FPT Education
FPT UNIVERSITY

Unity

Understanding Unity's Component System

Learning Objectives

- Understand the concept of Unity's component system.
- Explore common components and their roles.
- Grasp the significance of component-based design in Unity.

What is Unity's Component System

The Entity Component System (ECS) is the core of the Unity Data-Oriented Tech Stack (DOTS). As the name indicates, ECS has three principal parts:

Entities — the entities, or things, that populate your game or program.

Components — the data associated with your entities, but organized by the data itself rather than by entity. (This difference in organization is one of the key differences between an object-oriented and a data-oriented design.)

Systems — the logic that transforms the component data from its current state to its next state— for example, a system might update the positions of all moving entities by their velocity times the time interval since the previous frame.

What is Unity's Component System

In a component system, components are mix-and-match packets of functionality, and objects are built up as a collection of components, rather than as a strict hierarchy of classes. A component system is a different (and usually more flexible) approach to object-oriented programming (OOP) that constructs game objects through composition rather than inheritance. [1]

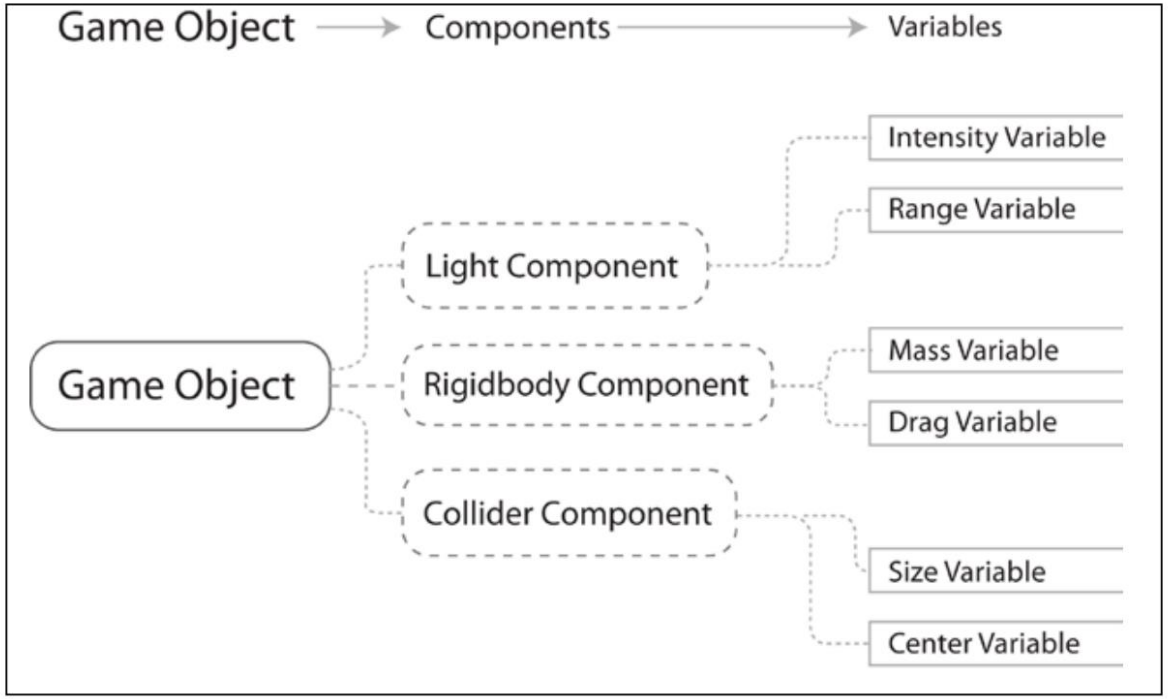
In a component system, objects exist on a flat hierarchy, and different objects have different collections of components. An inheritance structure, in contrast, has different objects on completely different branches of the tree. The component arrangement facilitates rapid prototyping, because you can quickly mix and match components rather than having to refactor the inheritance chain when objects change.

What is Unity's Component System

ESC
Entity
Component
System

Unity
Game Objects
Unity Components: Transform, MeshFilter, Renderer, Colider, Rigidbody, Script component
RenderSystem: SpriteRenderer, MeshRenderer, ParticleSystemRenderer MovingSystem: PhysicSystem: behaviour of RigidbodyComponent, ColliderComponent ...

Components in Unity



Components are one of the three principle elements of an Entity Component System architecture. They represent the data of your game or program.[8]

In Unity, components come in various forms. They can be for creating behavior, defining appearance, and influencing other aspects of an object's function in the game.

By attaching components to an object, you can immediately apply new parts of the game engine to your object. [10]

Components in Unity

Common components of game production come ***built-in*** with Unity, such as the Rigidbody component mentioned earlier, down to simpler elements such as lights, cameras, particle emitters, and more [10]

To build further interactive elements of the game, you'll write ***scripts***, which are also treated as components in Unity. Try to think of a script as something that extends or modifies the existing functionality available in Unity or creates *behavior* with the Unity scripting classes provided. [10]

Common Components in Unity

Components	Using / Datastore
Transform	Position, Rotation, scale
Renderer	Object appear on the screen
Collider	Define the shape of an object for the purposes of physical collisions
Rigidbody	Add motion to GameObject under the control of Unity's physics engine.
Script	All code execution in Unity starts from code files linked to an object in the scene Scripts in Unity are more akin to individual OOP classes, and scripts attached to objects in the scene are object instances [1]

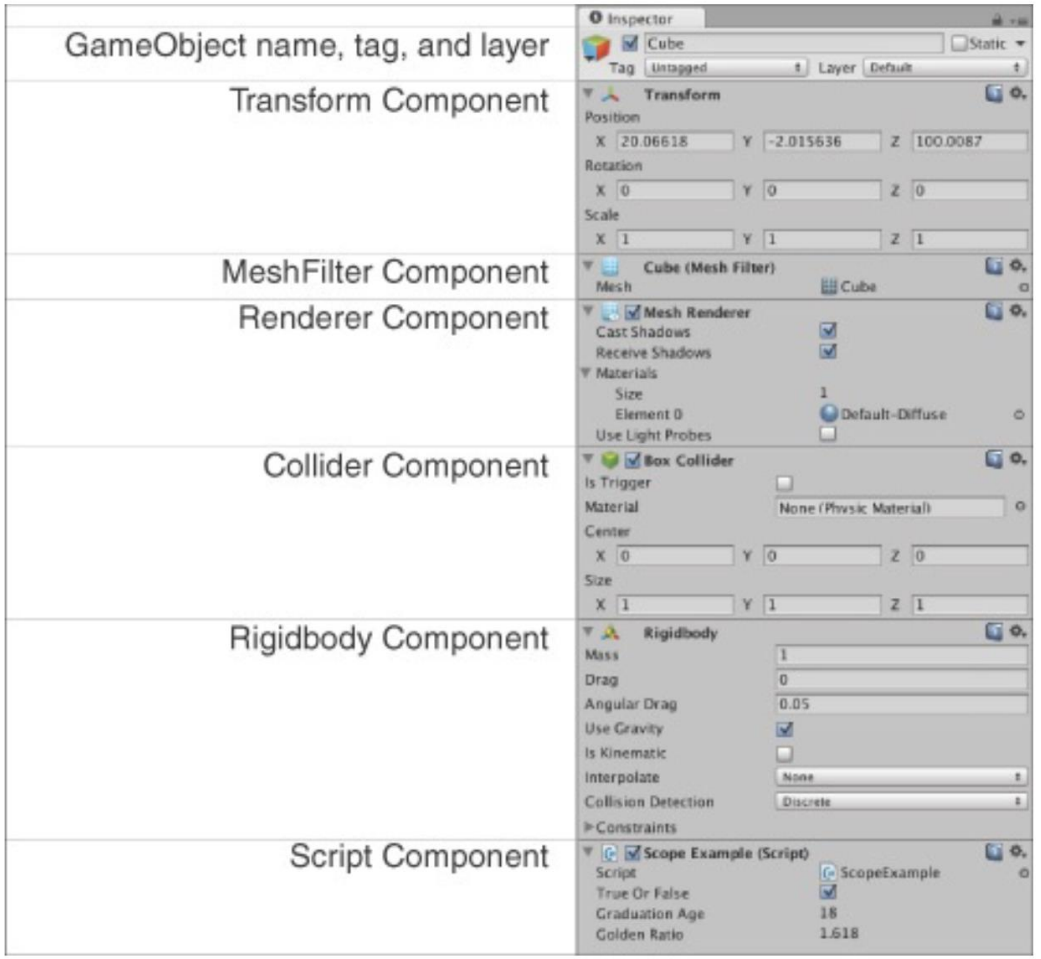


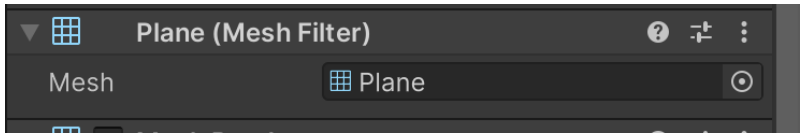
Figure 20.1 The Inspector pane showing various important components

The Transform Component

Transform: Position, Rotation, and Scale

Transform is a ***mandatory component*** that is present on all GameObjects. Transform handles critical GameObject information like ***position*** (the location of the GameObject), ***rotation*** (the orientation of the GameObject), and ***scale*** (the size of the GameObject). Though the information is displayed in the *Inspector pane*, Transform is also responsible for the parent/child relationships in the Hierarchy pane. When one object is the child of another, it moves with that parent object as if attached to it.

The MeshFilter Component

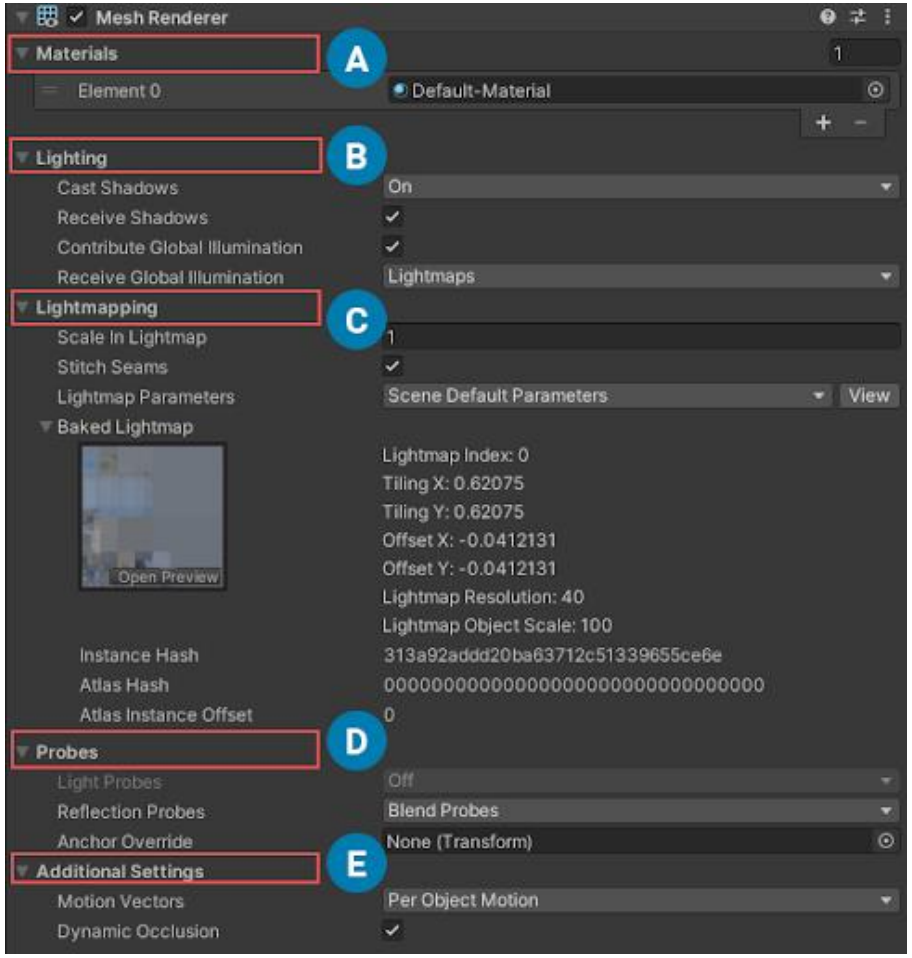


MeshFilter: The Model You See

A *MeshFilter*[19] component attaches a 3D mesh in your Project pane to a GameObject. To see a model on screen, the GameObject must have both a MeshFilter that handles the actual 3D mesh information and a MeshRenderer that combines that mesh with a shader or material and displays the image on screen. The MeshFilter creates a skin or surface for a GameObject, and the MeshRenderer determines the shape, color, and texture of that surface.

The Renderer Component

Renderer: Allows You to See the GameObject



A *Renderer* component—in most cases, a **MeshRenderer**—allows you to see the GameObject in the Scene and Game panes. The MeshRenderer requires a ***MeshFilter*** to provide ***3D mesh data*** as well as at least one ***Material*** if you want it to look like anything other than an ugly magenta blob (Materials apply textures to objects, and when no Material is present, Unity defaults to solid magenta to alert you to the problem). *Renderers bring the MeshFilter, the Material(s), and lighting together to show the GameObject on screen*.

The Rigdbody Component

Rigidbody: The Physics Simulation

The **Rigidbody** component controls the physics simulation of your GameObject. The Rigidbody component simulates acceleration and velocity every ***FixedUpdate*** (generally every 50th of a second) to update the position and rotation of the Transform component over time. It also uses the Collider component to handle collisions with other GameObjects. The Rigidbody component can also model things like ***gravity***, ***drag***, and various forces like *wind* and *explosions*. Set *isKinematic* to true if you want to directly set the position of your GameObject without using the physics provided by Rigidbody.

The Collider Component

Collider: The Physical Presence of the GameObject

A *Collider*[21] component enables a GameObject to have a physical presence in the game world and collide with other objects. Unity has four different kinds of Collider components, which I've arranged below in order of their speed. Calculating whether another object has collided with a Sphere Collider is extremely fast, but calculating whether an object has collided with a Mesh Collider is much slower:

- **Sphere Collider:**[22] The fastest collision shape to calculate. A ball or sphere.
- **Capsule Collider:**[23] A pipe with spheres at each end. The second fastest type.
- **Box Collider:**[24] A rectangular solid. Useful for crates and other boxy things.
- **Mesh Collider:**[25] A collider formed from a 3D mesh. Although useful and accurate, mesh colliders are much, much slower than any of the other three. Also, only Mesh Colliders with Convex set to `true` can collide with other Mesh Colliders.

Script Components

Running code in Unity: Script components

All code execution in Unity starts from code files linked to an object in the scene. Ultimately, this code execution is all part of the component system described earlier; game objects are built up as a collection of components, and that collection can include scripts to execute.

Listing 1.1 Code template for a basic script component

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

public class HelloWorld : MonoBehaviour {
    void Start() {
        // do something once
    }

    void Update() {
        // do something every frame
    }
}
```

Include namespaces for Unity and .NET/Mono classes.

The syntax for inheritance

Put code here that runs once.

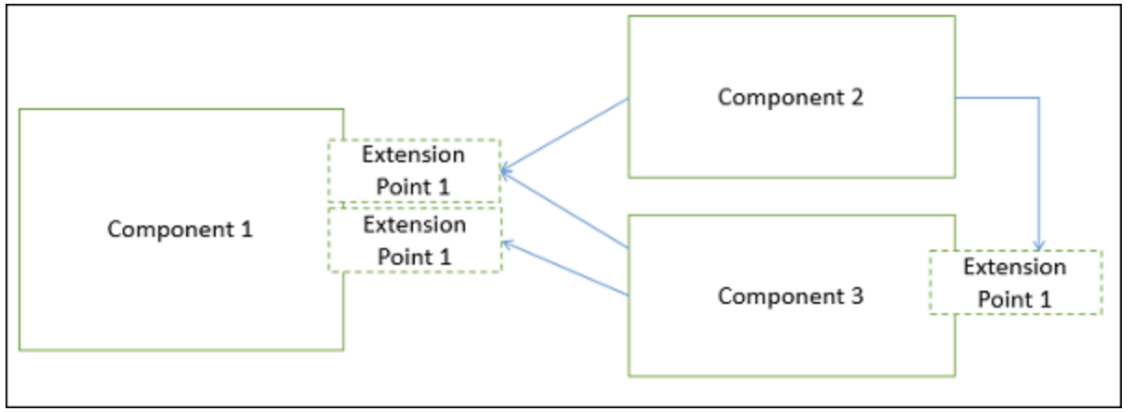
Put code here that runs every frame.

Script Components

As you've probably surmised from this description, in Unity, scripts *are* components—not all scripts, mind you, only scripts that inherit from `MonoBehaviour`, the base class for script components. `MonoBehaviour` defines the invisible groundwork for attaching components to game objects, and (as shown in listing 1.1) inheriting from it provides a couple of automatically run methods that you can implement. Those methods include `Start()`, called once when the object becomes active (which is generally as soon as the scene with that object has loaded), and `Update()`, which is called every frame. Your code is run when you put it inside these predefined methods.

Advantages of Component-Based Design (CBD)

There are many advantages of developing the applications using the CBD



1. **Ease of deployment** − As new compatible versions become available, it is easier to replace existing versions with no impact on the other components or the system as a whole.

1. **Reduced cost** − The use of third-party components allows you to spread the cost of development and maintenance.

3. **Ease of development** − Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.

3. **Reusable** − The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.

Advantages of Component-Based Design (CBD)

5. **Modification of technical complexity** − A component modifies the complexity through the use of a component container and its services.

1. **Reliability** − The overall system reliability increases since the reliability of each individual component enhances the reliability of the whole system via reuse.

1. **System maintenance and evolution** − Easy to change and update the implementation without affecting the rest of the system.

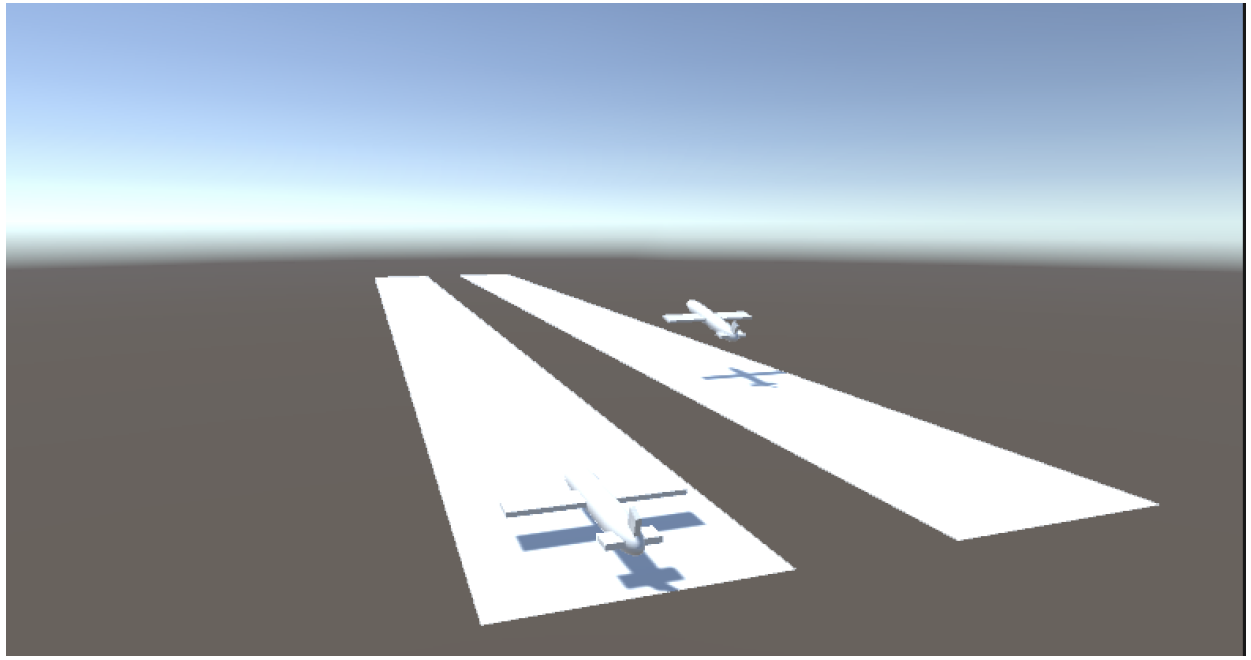
1. **Independent** − Independency and flexible connectivity of components. Independent development of components by different group in parallel. Productivity for the software development and future software development.

Hands-on

A. Create C# script component to control the Airplane

B. Follow camera

C. Add I/O control to Airplane



Practice Prepair

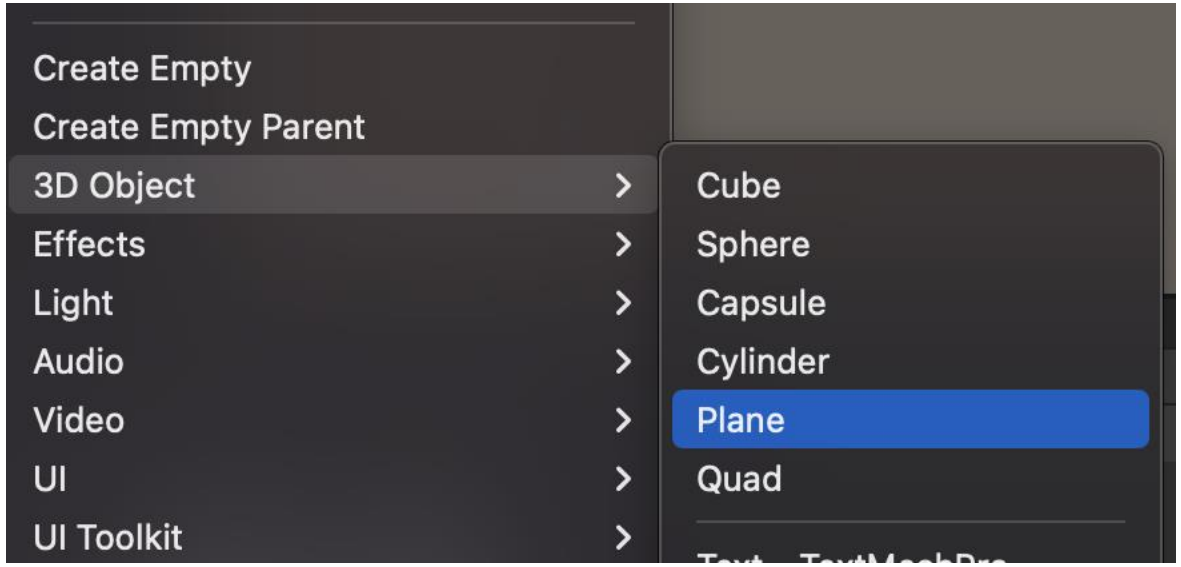
Hint: In order to organize the Assets folder, create the following folders:

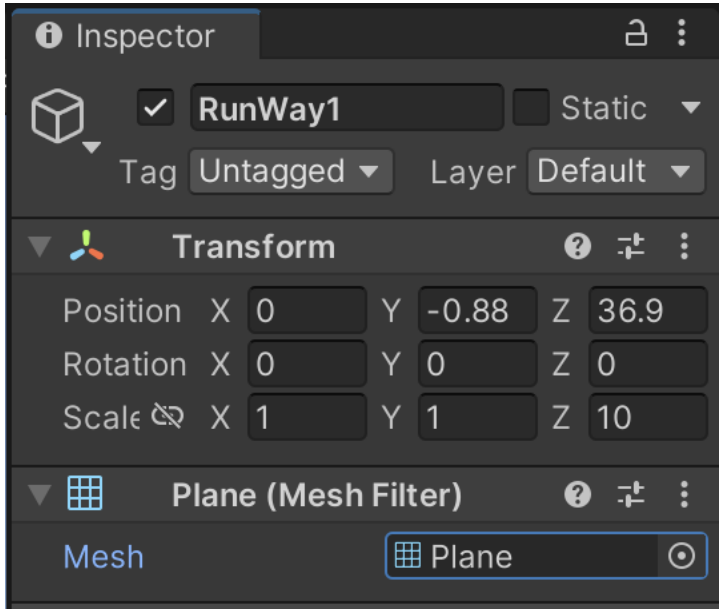
Example 1

```
Assets
+---Art
| +---Materials
| +---Models
| +---Textures
+---Audio
| +---Music
| \---Sound
+---Code
| +---Scripts  # C# scripts
| \---Shaders  # Shader files and shader graphs
+---Docs   # Wiki, concept art, marketing material
+---Level   # Anything related to game design in Unity
| +---Prefabs
| +---Scenes
| \---UI
```

Practice 1 – Airplane take off

1. Add Plane to scene. Rename it to Runway. And Scale it to make Airplane's Runway

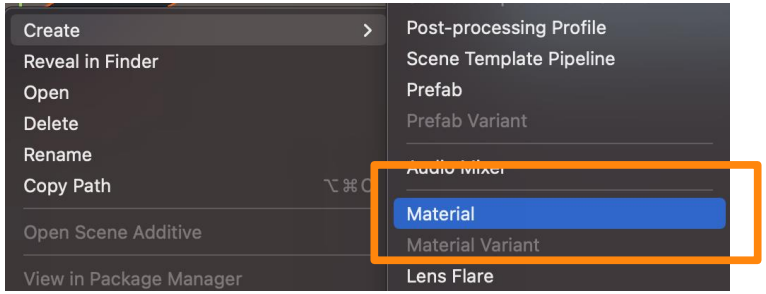


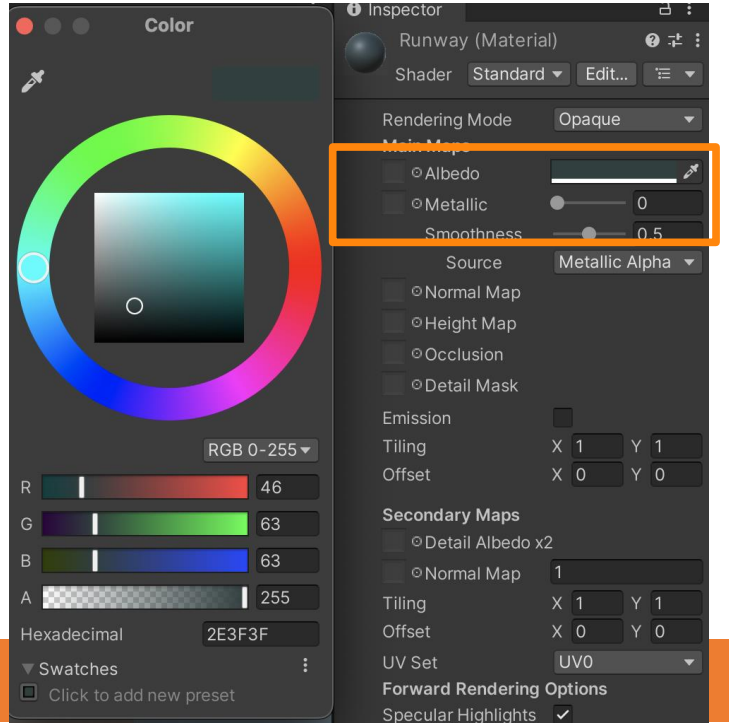


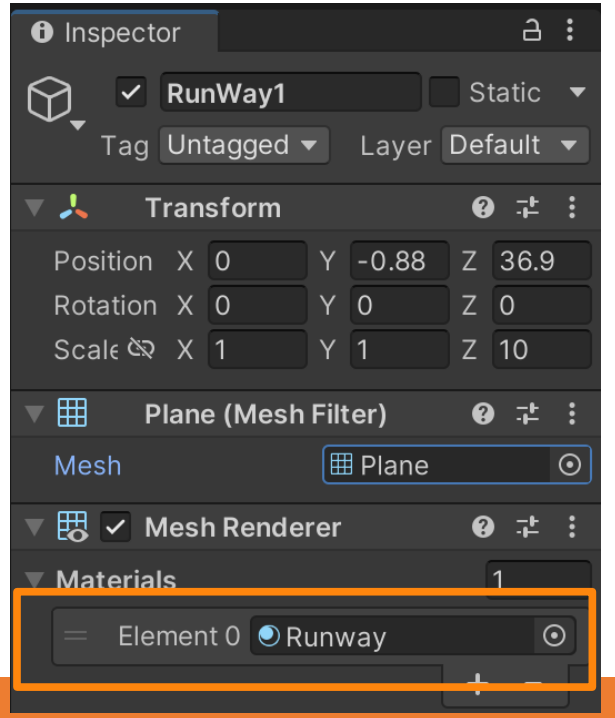
Practice 1 – Airplane take off

2. Create new material for runway (optional)

- On Project window, select Asset folder for Material
- Select Asset menu and choose Create > Material. Rename new material to Runway. On Inspector panel, choose Albedo color at you like.
- Select RunWay GameObject, assign new material to GameObject by drag and drop new material to Material - Element 0







Practice 1 – Airplane take off

3. Add script component to Airplane GameObject

- In Assets > Code > Scripts, right-click and choose Create > C# Script. Rename the script to ‘AirplaneTakeOff’.
- Double-click on Script to open a C# script file in Virtual Studio Code. Follow *this tutorial* if you can not open VSC.
- Paste Code in next slide to AirplaneTakeOff.cs file and Save
- Drag and Drop script from asset to Inspector panel of GameObject

Practice 1 – Airplane take off

```
// Update is called once per frame
0 references
void Update()
{
    if (isTakeOff)
    {
        if (takeOffSpeed < throttle)
        {
            takeOffSpeed = takeOffSpeed * takeOffAccelerate;
            transform.Translate(0, 0, -(takeOffSpeed * Time.deltaTime));
        }
        else
        {
            transform.Rotate(takeOffElevator, 0, 0);
            isTakeOff = false;
        }
    }
    else
    {
        transform.Rotate(elevatorUpWard * Time.deltaTime, 0, 0);
        transform.Translate(0f, elevatorUpWard * Time.deltaTime, -(throttle * Time.deltaTime));
    }

}
```

```
// Update is called once per frame
0 references
void Update()
{
    if (isTakeOff)
    {
        if (takeOffSpeed < throttle)
        {
            takeOffSpeed = takeOffSpeed * takeOffAccelerate;
            transform.Translate(0, 0, -(takeOffSpeed * Time.deltaTime));
        }
        else
        {
            transform.Rotate(takeOffElevator, 0, 0);
            isTakeOff = false;
        }
    }
    else
    {
        transform.Rotate(elevatorUpWard * Time.deltaTime, 0, 0);
        transform.Translate(0f, elevatorUpWard * Time.deltaTime, -(throttle * Time.deltaTime));
    }

}
```

Practice A – Airplane take off

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

public class NewBehaviourScript : MonoBehaviour
{

    // Elevator: Up / Down
    [SerializeField] private float elevatorUpWard = 0f;

    // Throttle (Speed): Up / Down
    [SerializeField] float throttle = 30f;
    // TODO Ailerons: Left / Right
    // TODOO Rudder

    // takeOffSpeed
    [SerializeField] float takeOffSpeed = 9f;

    // takeOffAccelerate
    [SerializeField] private float takeOffAccelerate = 1.0072f;

    [SerializeField] private bool isTakeOff = true;
    [SerializeField] private float takeOffElevator = 3.5f;
```

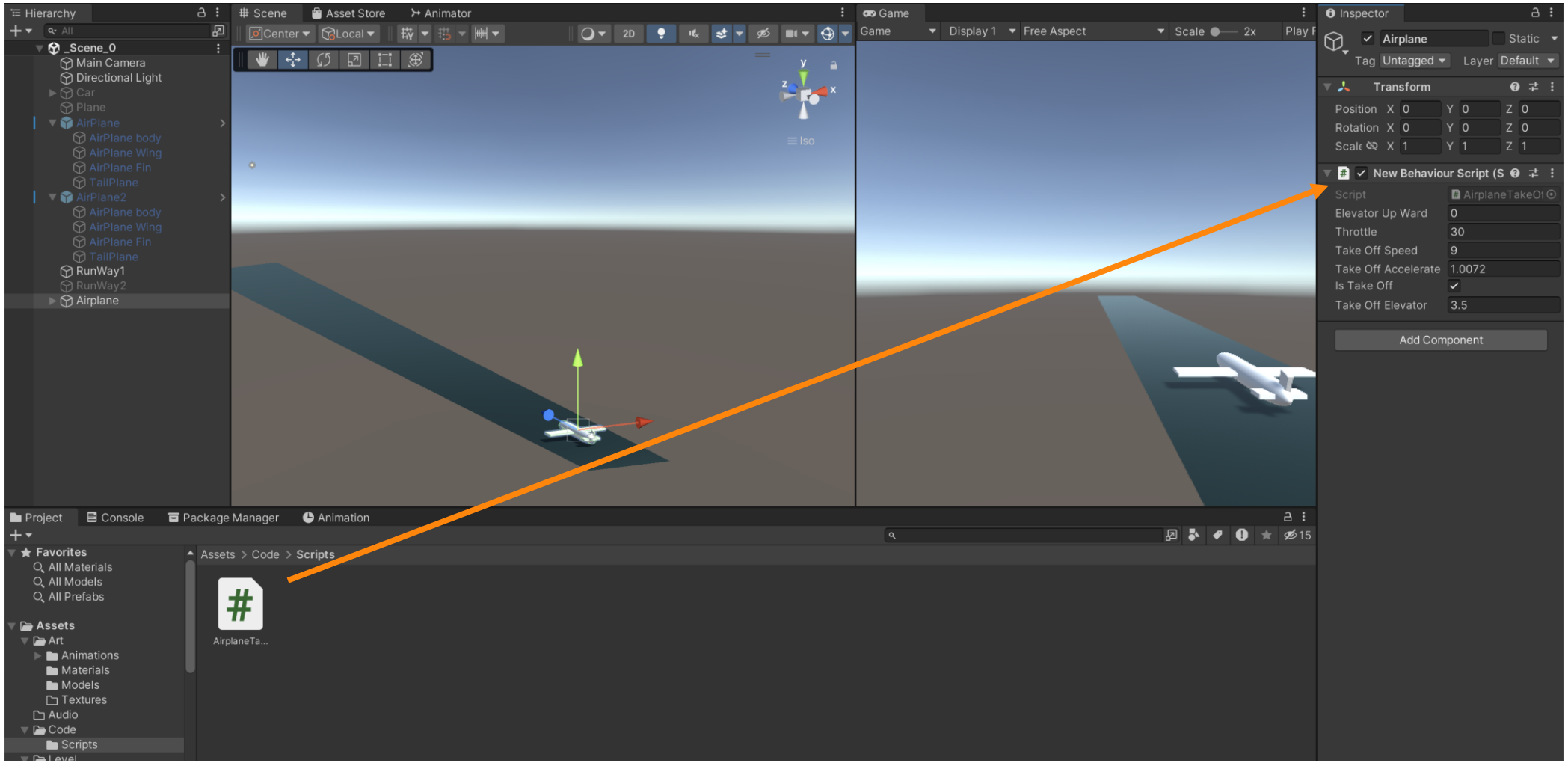
Practice A – Airplane take off

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

   // Update is called once per frame
   void Update()
   {
      if (isTakeOff)
      {
         if (takeOffSpeed < throttle)
         {
            takeOffSpeed = takeOffSpeed * takeOffAccelerate;
            transform.Translate(0, 0, -(takeOffSpeed * Time.deltaTime));
         }
         else
         {
            transform.Rotate(takeOffElevator, 0, 0);
            isTakeOff = false;
         }
      }
      else
      {
         transform.Rotate(elevatorUpWard * Time.deltaTime, 0, 0);
         transform.Translate(0f, elevatorUpWard * Time.deltaTime, -(throttle *
Time.deltaTime));
      }

   }
}
```

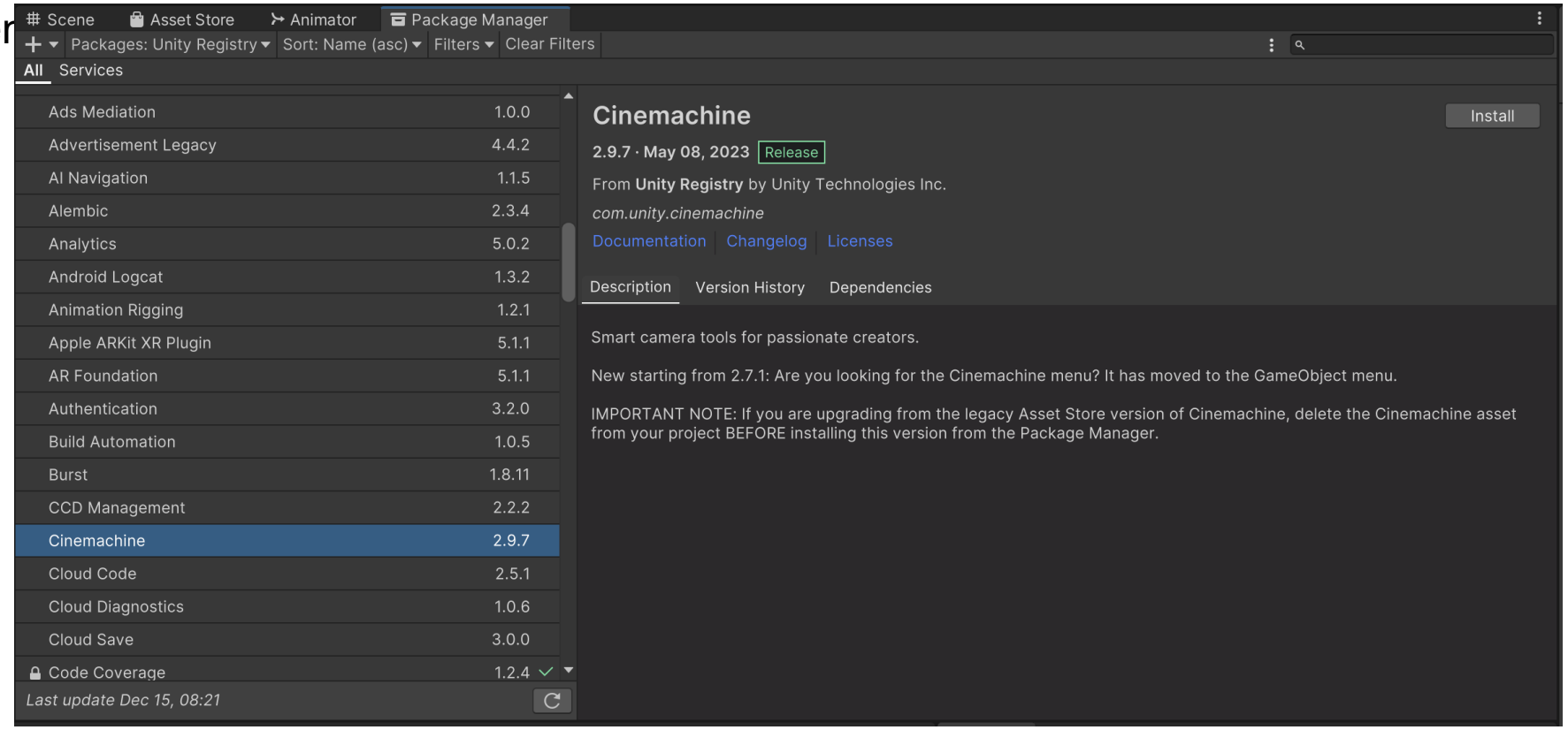
Practice A – Airplane take off



Practice B – Camera Follow Player

1. Install Cinemachine package

- From Package Nanager pane, select Unity Registry
- Find Ciner



Practice B – Camera Follow Player

1. Add Virtual Camera to scene

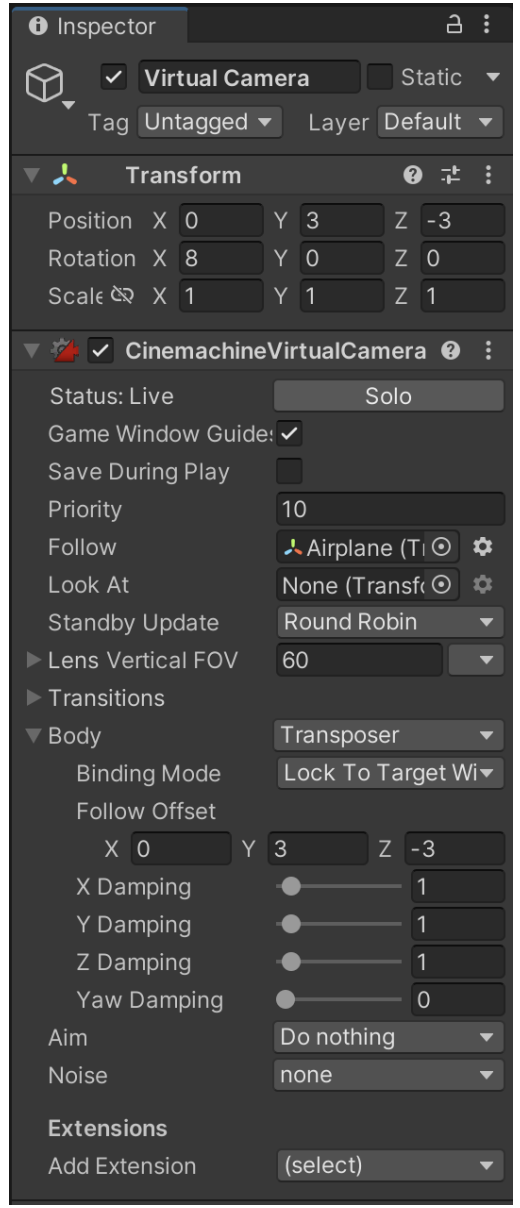
1. - From menu, choose GameObject > Cinemachine > Virtual Camera. Reset and change transform of Virtual Camera to set GameObject view from camera.
2. - Drag and Drop Virtual Camera to GameObject (optional)
3. - Drag and Drop Player GameObject (Airplane object) to Follow property of Virtual Camera.
4. - Set Virtual Camera property to set best view to Airplane:

   Follow: Airplane.

   Body: Transpoter

   Body – Follow Offset: X = 0, Y = 3, Z = -3

   Aim: Do nothing



Practice B – Camera Follow Player

1. Add Input key to controll the Airplane

1. Add following function to AirplaneTakeOff Script

```
55      void ProcessInput()
56      {
57          if (Input.GetKey(KeyCode.UpArrow))
58          {
59              transform.Rotate(elevatorUpWard * Time.deltaTime, 0, 0);
60              transform.Translate(0f, -(elevatorUpWard * Time.deltaTime), 0);
61              Debug.Log("Up");
62          }
63          if (Input.GetKey(KeyCode.DownArrow))
64          {
65              transform.Rotate(-(elevatorUpWard * Time.deltaTime), 0, 0);
66              transform.Translate(0f, -(elevatorUpWard * Time.deltaTime), 0);
67              Debug.Log("Down");
68          }
69          if (Input.GetKey(KeyCode.LeftArrow))
70          {
71              transform.Rotate(0, 0, -(elevatorUpWard * Time.deltaTime));
72              transform.Translate(-(elevatorUpWard * Time.deltaTime), 0f, 0);
73              Debug.Log("Right");
74          }
75          if (Input.GetKey(KeyCode.RightArrow))
76          {
77              transform.Rotate(0, 0, (elevatorUpWard * Time.deltaTime));
78              transform.Translate((elevatorUpWard * Time.deltaTime), 0f, 0);
79              Debug.Log("Right");
80          }
81      }
```

Conclusions

- Understanding Unity's **Component System**: Explored the modular approach where components encapsulate entity functionalities.
- Exploration of **Common Components**: Introduced vital components like **Transform, Renderer, Collider, Rigidbody,** and **Scripts**, elucidating their roles and importance.
- **Component-Based** Design Advantages: Discussed the benefits—**reusability**, **modularity**, **extensibility** — of this design approach.

References

1: Hocking, Joseph; Schell, Jesse, Unity in action: multiplatform game development in C#, 2022

3: Geig, Mike, Sams teach yourself Unity Game development in 24 hours, 2014

6: Gibson Bond, Jeremy, Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C,

8: Unity Technologies, Unity Manual, 2023

9: Tutorials Point, Learning Unity, 2023

10: Goldstone Will, Unity 3.x game development essentials, 2011