

Taller Scripting Principios Solid

Santiago Escandón

ID: 000422314

VIDEO 1: Aplicación interactiva

En este video se enseña acerca de como utilizar un Raycast para seleccionar objetos en un juego en Unity, mostrando como se debe implementar mediante una clase Update que utiliza selecciones, tranforms, GetComponent y claramente también el Raycast.

Mostrando también los ciclos que y condiciones que se cumplen para que el objeto en cuestión cambie de color cuando es seleccionado y también indicando cuales objetos son los que pueden ser seleccionados para evitar que todo en el juego cambie de color.

Una pieza importante para que esto funcione es esta parte del código:

```
var ray = Camera.main.ScreenPointToRay(Input.mousePosition);  
RaycastHit hit;
```

La cual nos dice que habrá un punto que indicara a donde esta mirando el jugador y este punto estará ubicado en donde se encuentra la posición del mouse, la cual por defecto es en el medio de la pantalla.

```
if (Physics.Raycast(ray, out hit))  
{  
    var selection = hit.transform;  
    if (selection.CompareTag(selectableTag))  
    {  
        var selectionRenderer = selection.GetComponent<Renderer>();  
        if (selectionRenderer != null)  
        {  
            selectionRenderer.material = highlightMaterial;  
        }  
    }  
}
```

Esta parte nos muestra que ocurre si un objeto es seleccionado indicando que debe renderizarse a un material diferente, si no está siendo seleccionado simplemente no pasara nada con el objeto.

La función anterior esta llamando a otra que es esta de acá:

```
if (_selection != null)  
{  
    var selectionRenderer = _selection.GetComponent<Renderer>();  
    selectionRenderer.material = defaultMaterial;  
    _selection = null;  
}
```

Mostrando que, si la selección es nula, es decir que no está seleccionando nada, el material del objeto será el que tiene por defecto en la escena.

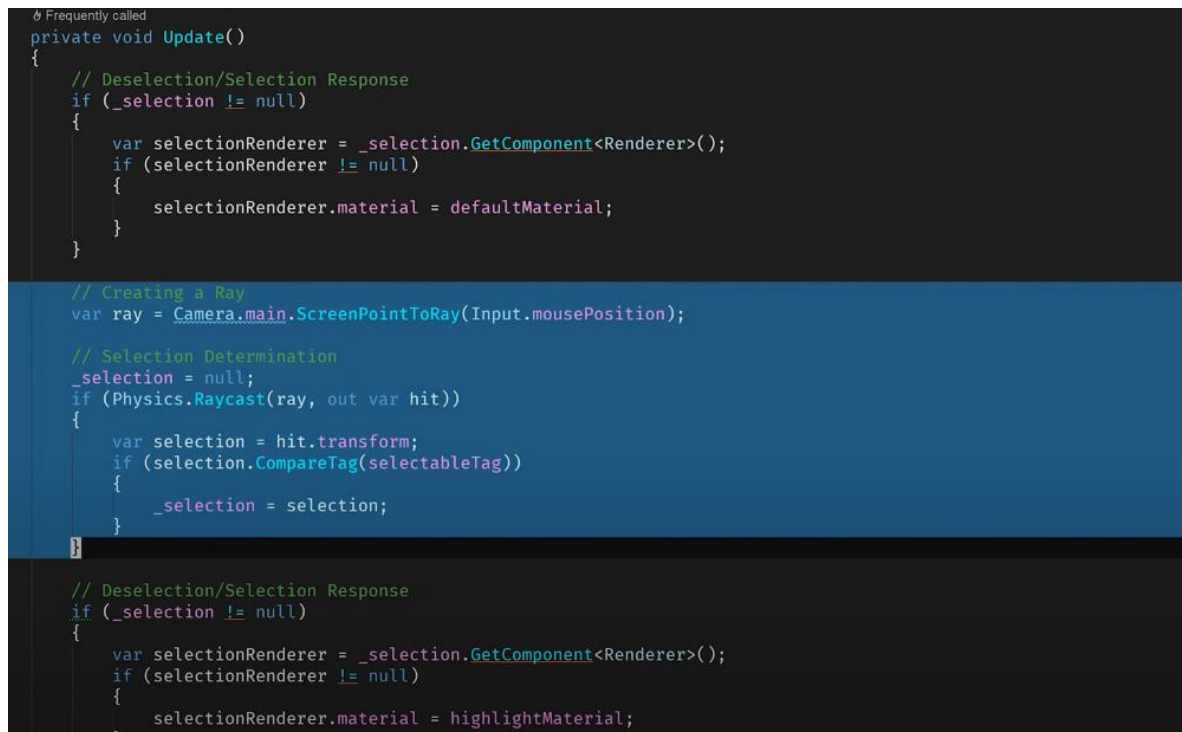
VIDEO 2: PRINCIPIOS SOLID

En este video se habla acerca de los 5 principios solid implementándolos en el mismo código que se usó en el anterior video.

1: Principio de responsabilidad única

Este principio dice que se debe separar todas las funciones que sirven para cosas diferentes.

En el video se implementa del siguiente modo:



```

// Frequently called
private void Update()
{
    // Deselection/Selection Response
    if (_selection != null)
    {
        var selectionRenderer = _selection.GetComponent<Renderer>();
        if (selectionRenderer != null)
        {
            selectionRenderer.material = defaultMaterial;
        }
    }

    // Creating a Ray
    var ray = Camera.main.ScreenPointToRay(Input.mousePosition);

    // Selection Determination
    _selection = null;
    if (Physics.Raycast(ray, out var hit))
    {
        var selection = hit.transform;
        if (selection.CompareTag(selectableTag))
        {
            _selection = selection;
        }
    }

    // Deselection/Selection Response
    if (_selection != null)
    {
        var selectionRenderer = _selection.GetComponent<Renderer>();
        if (selectionRenderer != null)
        {
            selectionRenderer.material = highlightMaterial;
        }
    }
}

```

Aquí podemos ver como por medio de comentarios dice para que sirve cada función y luego selecciona las partes a las cuales vamos a hacer especial énfasis en el video separándolas del resto.

2: Principio de abierto/cerrado

El cual nos indica que las clases deben estar abiertas a extensiones y cerradas para modificaciones

Este principio es utilizado en el video para que en vez de modificar la función de selección que hace que el objeto cambie de color se pueda extender de tal manera que siga usando esa lógica pero que en vez de cambiar de color simplemente envuelva al objeto.

De esta forma:

```
private void OnDeselect(Transform selection)
{
    var selectionRenderer = selection.GetComponent<Renderer>();
    if (selectionRenderer != null)
    {
        selectionRenderer.material = _selectionResponse.defaultMaterial;
    }
}

Scripting component 1 usage
internal class HighlightSelectionResponse : MonoBehaviour

[SerializeField] public Material highlightMaterial;  Set by Unity
[SerializeField] public Material defaultMaterial;  Set by Unity

public void OnSelect(Transform selection)
{
    var selectionRenderer = selection.GetComponent<Renderer>();
    if (selectionRenderer != null)
    {
        selectionRenderer.material = this.highlightMaterial;
    }
}
```

Sin embargo, para que esto funcione del todo aun así debería modificar el código por lo que para lograrlo usa el siguiente principio.

3: Principio de sustitución de Liskov

Este principio dice que las clases derivadas pueden ser sustituidas por sus clases bases.

Por lo cual lo que se va a hacer es tomar la clase Highlightselectionresponse y que en vez de que depende de la implementación indicada anteriormente dependa de una interfaz que podrá ser reemplazada por cualquier subtipo que quiera alterar su comportamiento.

```

Scripting component
internal class HighlightSelectionResponse : MonoBehaviour, ISelectionResponse
{
    [SerializeField] public Material highlightMaterial;  Set by Unity
    [SerializeField] public Material defaultMaterial;  Set by Unity

    0+1 usages
    public void OnSelect(Transform selection)
    {
        var selectionRenderer = selection.GetComponent<Renderer>();
        if (selectionRenderer != null)
        {
            selectionRenderer.material = this.highlightMaterial;
        }
    }
}

```

4: Principio de segregación e la interfaz

El cual dice que es preferible tener muchas interfaces que tengan pocos métodos a una interfaz que implemente muchos métodos de los cuales pueda que no use varios.

En el video se implementa de la siguiente manera:

```

Scripting component
public class OutlineSelectionResponse : MonoBehaviour, ISelectionResponse
{
    0+1 usages
    public void OnSelect(Transform selection)
    {
        var outline = selection.GetComponent<Outline>();
        if (outline != null) outline.OutlineWidth = 10;
    }

    0+1 usages
    public void OnDeselect(Transform selection)
    {
        var outline = selection.GetComponent<Outline>();
        if (outline != null) outline.OutlineWidth = 0;
    }
}

```

Mostrando una interfaz diferente para definir cómo será el contorno que va a envolver al objeto seleccionado.

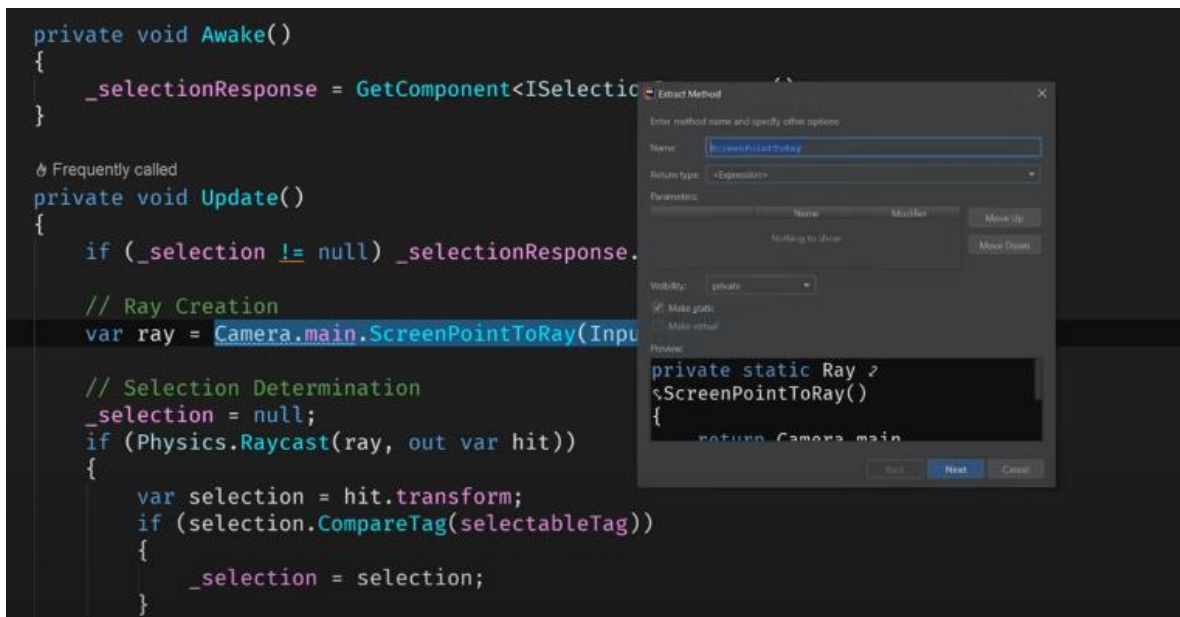
5: Principio de inversión de dependencias

Indica que las clases deben depender de abstracciones no de concreciones.

Esto es utilizado en que antes en la clase de respuesta de selección dependía de una implementación concreta pero ahora dependen de una abstracción.

VIDEO 3: PRINCIPIOS SOLID PARTE 2

Principio de responsabilidad única



Separas las funciones que sirvan para diferentes cosas.

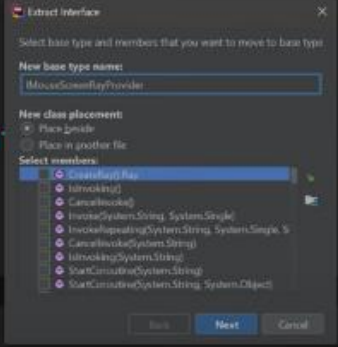
```
Scripting component 1 usage
public class MouseScreenRayProvider : MonoBehaviour
{
    Frequently called 1 usage
    public Ray CreateRay()
    {
        return Camera.main.ScreenPointToRay(Input.mousePosition);
    }
}
```

Principio de inversión de dependencias

```
if (selection.CompareTag(selectableTag))
{
    _selection = selection;
}

if (_selection != null) _selectionResponse.OnSelec...
```

```
Scripting component 1 usage
public class MouseScreenRayProvider : MonoBehaviour
{
    Frequently called 1 usage
    public Ray CreateRay()
    {
        return Camera.main.ScreenPointToRay(Input.mousePosition);
    }
}
```



No debe depender de una implementación concreto sino de una interfaz

```
Scripting component 1 usage
public class MouseScreenRayProvider : MonoBehaviour, IRayProvider
{
    Frequently called 1 usage
    public Ray CreateRay()
    {
        return Camera.main.ScreenPointToRay(Input.mousePosition);
    }
}
```

Principio de abierto/cerrado


```

private void Update()
{
    if (_selection != null) _selectionResponse.OnDeselect(_selection);

    // Ray Creation
    var ray = _rayProvider.CreateRay();

    // Selection Determination
    _selection = null;
    if (Physics.Raycast(ray, out var hit))
    {
        var selection = hit.transform;
        if (selection.CompareTag(selectableTag))
        {
            _selection = selection;
        }
    }

    if (_selection != null) _selectionResponse.OnSelect(_selection);
}

```

Esta implementación se podría modificar para que sea mejor, sin embargo, esto rompería el principio de abierto/cerrado por lo cual se creara una nueva clase que tendrá 2 métodos uno que verifique la selección y otro que la devuelva.

```

private void Update()
{
    if (_currentSelection != null) _selectionResponse.OnDeselect(_currentSelection);

    // Ray Creation
    var ray = _rayProvider.CreateRay();

    // Selection Determination
    _selector.Check(ray);

    _currentSelection = _selector.GetSelection();

    if (_currentSelection != null) _selectionResponse.OnSelect(_currentSelection);
}

```

Scripting component 1 usage

public class RayCastBasedTagSelector : MonoBehaviour

[SerializeField] public string selectableTag = "Selectable"; ⚡ Set by Unity

public Transform _selection; ⚡ Set by Unity

⚡ Frequently called 1 usage

```

public void Check(Ray ray)
{
    this._selection = null;
    if (Physics.Raycast(ray, out var hit))
    {
        var selection = hit.transform;
        if (selection.CompareTag(this.selectableTag))
        {
            this._selection = selection;
        }
    }
}

```


Principio de sustitución de Liskov

```
public interface ISelector
{
    // Frequently called 1 usage 1 implementation
    void Check(Ray ray);
    // Frequently called 1 usage 1 implementation
    Transform GetSelection();
}
```

Iselector se debería poder reemplazar por cualquier subtipo

```
Scripting component
public class SelectionManager : MonoBehaviour
{
    private IRayProvider _rayProvider;
    private ISelector _selector;
    private ISelectionResponse _selectionResponse;

    private Transform _currentSelection;
    private Transform _selection;

    private void Awake()
    {
        _rayProvider = GetComponent<IRayProvider>();
        _selector = GetComponent<ISelector>();
        _selectionResponse = GetComponent<ISelectionResponse>();
    }

    // Frequently called
    private void Update()
    {
        if (_currentSelection != null) _selectionResponse.OnDeselect(_currentSelection);

        _selector.Check(_rayProvider.CreateRay());
        _currentSelection = _selector.GetSelection();

        if (_currentSelection != null) _selectionResponse.OnSelect(_currentSelection);
    }
}
```

Principio de segregación e la interfaz

Esta última imagen también cumple con este principio debido a que se han tomado todas las responsabilidades y se han dividido en pequeños subconjuntos de funcionalidad y cada uno en su propia interfaz que puede ser reemplazada cuando sea necesario.

VIDEO 4: BENEFICIOS DE LOS PRINCIPIOS SOLID EN UN PROYECTO

Al ver la funcionalidad de los principios solid en la implementación de un proyecto puedo decir que los principio ayudan mucho a que los códigos sean muchas organizados y concretos, mostrando que evitan que estos se sumerjan en un caos de líneas donde pueden surgir errores inesperados.

A primera vista puede verse muy raro y confuso sin embargo este estará en realidad mucho más ordenado y entendible.

Debido a los anteriores videos el código ya estaba organizado y gracias a esto a la ora de modificarlo para agregar nuevas funciones fue más fácil la implementación de este cambio aun manten8iendo todos los principios intactos para seguir con un buen desempeño del código.

Abierto/Cerrado

```
using UnityEngine;

Scripting component
public class ResponsiveSelector : MonoBehaviour, ISelector
{
    private Transform _selection;

    0+1 usages
    public void Check(Ray ray)
    {
        _selection = null;
    }

    0+1 usages
    public Transform GetSelection()
    {
        return _selection;
    }
}
```

Segregación e interfaz, Responsabilidad única

```

using System.Collections.Generic;
using UnityEngine;

[Scripting component]
public class ResponsiveSelector : MonoBehaviour, ISelector
{
    [SerializeField] private List<Selectable> selectables; [Set by Unity]

    private Transform _selection;

    [0+1 usages]
    public void Check(Ray ray)
    {
        _selection = null;

        for (int i = 0; i < selectables.Count; i++)
        {
            var vector1 = ray.direction;
            var vector2 = selectables[i].transform.position - ray.origin;

            var lookPercentage = Vector3.Dot(lhs: vector1.normalized, rhs: vector2.normalized);

            selectables[i].LookPercentage = lookPercentage;
        }
    }

    [0+1 usages]
    public Transform GetSelection()
    {
        return _selection;
    }
}

```

Inversión de dependencia

```

using UnityEngine;

[Scripting component]
public class ResponsiveSelector : MonoBehaviour, ISelector
{
    [SerializeField] private List<Selectable> selectables; [Set by Unity]
    [SerializeField] private float threshold = 0.97f; [Set by Unity]

    private Transform _selection;

    [0+1 usages]
    public void Check(Ray ray)
    {
        _selection = null;

        var closest = 0f;

        for (int i = 0; i < selectables.Count; i++)
        {
            var vector1 = ray.direction;
            var vector2 = selectables[i].transform.position - ray.origin;

            var lookPercentage = Vector3.Dot(lhs: vector1.normalized, rhs: vector2.normalized);

            selectables[i].LookPercentage = lookPercentage;

            if (lookPercentage > threshold && lookPercentage > closest)
            {
                closest = lookPercentage;
                _selection = selectables[i].transform;
            }
        }
    }

    [0+1 usages]
    public Transform GetSelection()
    {
        return _selection;
    }
}

```

VIDEO 5: HERRAMIENTA PARA EDITOR DE UNITY

Principio de responsabilidad Única

```
Scripting component  ... More
public class CompositeSelectionResponse : MonoBehaviour, ISelectionResponse
{
    private List<ISelectionResponse> _selectionResponses;
    private int _currentIndex;

    [0+2 usages]
    public void OnSelect(Transform selection)
    {
        _selectionResponses[_currentIndex].OnSelect(selection);
    }

    [0+2 usages]
    public void OnDeselect(Transform selection)
    {
        _selectionResponses[_currentIndex].OnDeselect(selection);
    }
}
```

Abierto/Cerrado

```
Scripting component
public class CompositeSelectionResponse : MonoBehaviour, ISelectionResponse
{
    [SerializeField] private GameObject selectionResponseHolder;  Set by Unity

    private List<ISelectionResponse> _selectionResponses;
    private int _currentIndex;

    private void Start()
    {
        _selectionResponses = selectionResponseHolder.GetComponent<ISelectionResponse>().ToList();
    }
}
```

Segregación de interfaz, sustitución de Liskov

```

Scripting component
public class Switcher : MonoBehaviour
{
    public List<GameObject> ChangeableObjects; // Set by Unity

    private List<IChangeable> _changeableObjects = new List<IChangeable>();

    // Event function
    private void Start()
    {
        for (var i = 0; i < ChangeableObjects.Count; i++)
        {
            var changeableObject = ChangeableObjects[i].GetComponent<IChangeable>();
            _changeableObjects.Add(changeableObject);
        }
    }

    // Event function
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.N))
        {
            for (var i = 0; i < _changeableObjects.Count; i++)
            {
                _changeableObjects[i].Next();
            }
        }
    }
}

```

Inversión de dependencias

```

private void Start()
{
    _selectionResponses = selectionResponseHolder.GetComponents<ISelectionResponse>().ToList();
}

[ContextMenu("Next")]
// 0+1 usages
public void Next()
{
    _selectionResponses[_currentIndex].OnDeselect(_selection);
    _currentIndex = (_currentIndex + 1) % _selectionResponses.Count;
    _selectionResponses[_currentIndex].OnSelect(_selection);
}

// 0+3 usages
public void OnSelect(Transform selection)
{
    _selection = selection;
    if (HasSelection())
    {
        _selectionResponses[_currentIndex].OnSelect(selection);
    }
}

// 0+3 usages
public void OnDeselect(Transform selection)
{
    _selection = null;
    if (HasSelection())
    {
        _selectionResponses[_currentIndex].OnDeselect(selection);
    }
}
}

```

CONCLUSIÓN:

La implementación de Solid ayuda a comprender más fácil un código y hace que se sienta mas limpio por así decirlo dando mayor accesibilidad a agregar nuevas funciones en él.

