using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class Filler : MonoBehaviour

{

    //public bool StartEmpty;

    [Range(0, 1.0f)] public float StartFillPercent;

    public float TriggerTime = 0.2f;

    private float Hat = 0.0f;

    //private float Factor = 13.543f;//making real size

    private float a = 0.75f;

    private float FillOffset = 0.51f;

     private float CurrentVolume;

     private float S;

     private float VesselHeight;

     private float MaxVolume;

     private Renderer rend;

     private bool IsPouring;

     private float ActualHeight;

     private float VolPerDrop;

     private bool IsEmpty;

     private enum FunctionControlls { Start, Stop }

    public float GetFillPercent() => CurrentVolume / MaxVolume;

    public float GetVolume() => CurrentVolume;

    public void Fill(float volume)

     {

         if ((IsEmpty == true) && (volume != 0))

         {

             IsEmpty = false;

             transform.parent.GetChild(1).GetComponent<LiquidCore\_v1>().SetUpMaterial();

         }

         float h;

         CurrentVolume += volume;

        CurrentVolume = Mathf.Clamp(CurrentVolume, 0, MaxVolume);

        h = CurrentVolume / MaxVolume;

        ActualHeight = h;

    }

     private void Pouring(FunctionControlls Control)

     {

         ParticleSystem PS = transform.parent.GetChild(1).GetComponent<ParticleSystem>();

         var emission = PS.emission;

         //logic

         if (Control == FunctionControlls.Start)

         {

             emission.enabled = true;

             IsPouring = true;

         }

         else if (Control == FunctionControlls.Stop)

         {

             emission.enabled = false;

             IsPouring = false;

         }

         else

         {

             emission.enabled = false;

             IsPouring = false;

         }

     }

     private void EnableCollider()

     {

         GetComponent<CapsuleCollider>().enabled = true;

     }

    private void DisableCollider()

    {

        GetComponent<CapsuleCollider>().enabled = false;

    }

    void Start()

     {

         VolPerDrop = transform.parent.GetChild(1).GetComponent<LiquidCore\_v1>().VolumePerDrop;

         rend = GetComponent<Renderer>();

         VesselHeight = rend.bounds.size.y;

         S = Mathf.PI \* Mathf.Pow(rend.bounds.size.x / 2, 2);

        MaxVolume = 1f / 3f \* S \* VesselHeight;//\* Factor;

        //Debug.Log(MaxVolume + " of " + transform.parent.name);

         CurrentVolume = StartFillPercent \* MaxVolume;

        switch (transform.parent.tag)

        {

            case "Cylinder":

                a = 0.8f;

                FillOffset = 1.71f;

                Hat = 0.62f;

                break;

            case "Flask":

                a = 1.3f;

                FillOffset = 1f;

                Hat = 0;

                break;

            case "Beaker":

                a = 0.85f;

                FillOffset = 1.02f;

                Hat = 0.33f;

                break;

            case "Tube":

                a = 0.8f;

                FillOffset = 0.5f;

                Hat = 1.2f;

                break;

            case "Bottle":

                a = 2f;

                FillOffset = 1.1f;

                Hat = 0.01f;

                break;

            case "Funnel":

                a = 0.5f;

                FillOffset = 0.5f;

                Hat = 0.01f;

                break;

        }

         Fill(0);

    }

    private void Awake()

    {

       // pouring fix

        GetComponent<CapsuleCollider>().contactOffset = 0.3f;

        GetComponentInParent<MeshCollider>().contactOffset = 0.4f;

        var col = transform.parent.GetChild(1).GetComponent<ParticleSystem>().collision;

        col.radiusScale = 0.5f;

    }

    void Update()

     {

         float angle, cos, AmountToDisplay;

         //emptyness check

         if (CurrentVolume <= 0 && IsEmpty == false)

         {

             IsEmpty = true;

             transform.parent.GetChild(1).GetComponent<LiquidCore\_v1>().SetEmpty();

             ActualHeight = 0;

            transform.parent.GetComponent<contain>().inVessel.Clear();

            transform.parent.GetComponent<acidity>().Empty();

        }

         angle = Vector3.Angle(transform.up, Vector3.up);

         cos = Mathf.Cos(angle \* Mathf.PI / 180);

        //display fix

        AmountToDisplay = -VesselHeight \* a \* Mathf.Pow(ActualHeight, (2f - angle / 180) / 2.5f) +

            (90 - angle) / 90 \* FillOffset + (VesselHeight + Hat) \* a \* angle / 180;

        //   AmountToDisplay = (90 - angle) / 90 \* (FillOffset - ( VesselHeight \* a \* CurrentVolume/MaxVolume ) );

        rend.material.SetFloat("\_FillAmount", AmountToDisplay);

        if (IsEmpty == false)

         {

             if ((cos \* (VesselHeight)) <= -(ActualHeight - FillOffset))

             {

                DisableCollider();

                 if (IsPouring == true)

                 {

                    Fill(VolPerDrop \* transform.parent.GetChild(1).GetComponent<ParticleSystem>().emission.rateOverTime.constant \* 2 \* Time.deltaTime \* -1);

                 }

                 else

                 {

                    Pouring(FunctionControlls.Start);

                 }

             }

             else

             {

                 Invoke("EnableCollider", TriggerTime);

                 if (IsPouring == true)

                 {

                     Pouring(FunctionControlls.Stop);

                     Debug.Log("Stop");

                 }

             }

         }

         else

         {

                 Invoke("EnableCollider", TriggerTime);

                 Pouring(FunctionControlls.Stop);

         }

     }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class LiquidCore\_v1 : MonoBehaviour {

    private Renderer rend;

    [Header("Properties")]

    public string Name;

    public string Type;

    public Material DefaultMat;

    public Material DefaultParticleMat;

    public float VolumePerDrop;

    public Color StartColor;

    private Color TransparentColor = new Color(255, 255, 255, 0);

    private List<ParticleCollisionEvent> CollEvents;

    public static float addedC, addedV;

    public static string added, curVessel;

    public void SetUpMaterial()

    {

        Material matp , matl;

        matl = new Material(DefaultMat);

        matp = new Material(DefaultParticleMat);

        matp.color = StartColor;

        gameObject.GetComponent<ParticleSystemRenderer>().trailMaterial = matp;

        if (rend != null)

        {

            matl.SetColor("\_Tint", StartColor);

            rend.material = matl;

        }

    }

    public void SetEmpty()

    {

        Name = "NONE";

        Material matl;

        matl = new Material(DefaultMat);

        matl.SetColor("\_Tint", TransparentColor);

        rend.material = matl;

        Invoke("DisablePS", 1);

    }

    private void DisablePS()

    {

        gameObject.GetComponent<ParticleSystemRenderer>().trailMaterial.color = TransparentColor;

    }

     void Start () {

        CollEvents = new List<ParticleCollisionEvent>();

        if (transform.parent != null) rend = transform.parent.GetChild(0).GetComponent<Renderer>();

            SetUpMaterial();

    }

    private void OnParticleCollision(GameObject other)

    {

        int i = 0;

        Debug.Log("PrtCol");

        int numCollisionEvents = GetComponent<ParticleSystem>().GetCollisionEvents(other, CollEvents);

        while (i < numCollisionEvents)

        {

            if (CollEvents[i].colliderComponent.GetComponent<Filler>())

            {

                other.GetComponentInChildren<Filler>().Fill(VolumePerDrop);

                // Destroy(GetComponent<ParticleSystem>().GetParticles[i]);

                added = transform.GetComponent<LiquidCore\_v1>().Type;

                addedC = transform.GetComponentInParent<acidity>().startC;

                addedV = transform.GetComponent<LiquidCore\_v1>().VolumePerDrop;

                if (CollEvents[i].colliderComponent.gameObject.transform.parent.tag == "Funnel")

                {

                    CollEvents[i].colliderComponent.transform.parent.GetChild(1).GetComponent<LiquidCore\_v1>().Type = added;

                    CollEvents[i].colliderComponent.transform.GetComponentInParent<acidity>().startC = addedC;

                }

                else

                {

                    CollEvents[i].colliderComponent.gameObject.transform.parent.GetComponent<contain>().Add();

                    CollEvents[i].colliderComponent.gameObject.transform.parent.GetComponent<acidity>().Reaction();

                    if (CollEvents[i].colliderComponent.gameObject.transform.parent.GetComponent<indicators>() != null)

                        CollEvents[i].colliderComponent.gameObject.transform.parent.GetComponent<indicators>().done = false;

                }

            }

            i++;

        }

    }

}

пїњ

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class acidity : MonoBehaviour

{

    public float startV, startC, pH, addedV, addedC;

    public string thisSub, addedSub;

    private float curCH, curCOH;

    void Start()

    {

        startV = transform.GetChild(0).GetComponent<Filler>().GetFillPercent();

        thisSub = transform.GetChild(1).GetComponent<LiquidCore\_v1>().Type;

        if (thisSub == "HCl" || thisSub == "H2O") pH = Mathf.Log10(startC) \* (-1);

        if (thisSub == "NaOH") pH = Mathf.Log10(Mathf.Pow(10, -14) / startC) \* (-1);

    }

    public void Reaction()

    {

        addedSub = LiquidCore\_v1.added;

        addedV = LiquidCore\_v1.addedV;

        addedC = LiquidCore\_v1.addedC;

        if (startV == 0)

        {

            thisSub = addedSub;

            startC = addedC;

            startV = addedV;

            if (thisSub == "HCl" || thisSub == "H2O")

                pH = Mathf.Log10(startC) \* (-1);

            if(thisSub == "NaOH")

                pH = Mathf.Log10(Mathf.Pow(10, -14) / startC) \* (-1);

        }

        else

        {

            #region HCl;

            if (thisSub == "HCl" || thisSub == "H2O")

            {

                if (addedSub == "NaOH")

                {

                    curCH = startC \* (startV / startV + addedV);

                    curCOH = addedC \* (addedV / startV + addedV);

                    float b = curCH + curCOH;

                    float D = Mathf.Pow(b, 2) - 4 \* (curCH \* curCOH - Mathf.Pow(10, -14));

                    float x = (b - Mathf.Pow(D, 0.5f)) / 2;

                    curCH = curCH - x;

                    curCOH = curCOH - x;

                    pH = Mathf.Log10(curCH) \* (-1);

                    startV = startV + addedV;

                    if (pH <= 7)

                    {

                        startC = curCH;

                    }

                    else

                    {

                        startC = Mathf.Pow(10, -14) / curCH;

                        thisSub = "NaOH";

                    }

                }

                else

                {

                    curCH = (startC \* startV + addedC \* addedV) / (startV + addedV);

                    pH = Mathf.Log10(curCH) \* (-1);

                    startC = curCH;

                    startV = startV + addedV;

                }

            }

            #endregion;

            #region NaOH;

            if (thisSub == "NaOH" || thisSub == "H2O")

            {

                if (addedSub == "HCl")

                {

                    curCOH = startC \* (startV / startV + addedV);

                    curCH = addedC \* (addedV / startV + addedV);

                    float b = curCH + curCOH;

                    float D = Mathf.Pow(b, 2) - 4 \* (curCH \* curCOH - Mathf.Pow(10, -14));

                    float x = (b - Mathf.Pow(D, 0.5f)) / 2;

                    curCH = curCH - x;

                    curCOH = curCOH - x;

                    pH = Mathf.Log10(Mathf.Pow(10, -14) / curCOH) \* (-1);

                    startV = startV + addedV;

                    if (pH >= 7)

                    {

                        startC = curCOH;

                    }

                    else

                    {

                        startC = Mathf.Pow(10, -14) / curCOH;

                        thisSub = "HCl";

                    }

                }

                //dilution

                else

                {

                    curCOH = (startC \* startV + addedC \* addedV) / (startV + addedV);

                    pH = Mathf.Log10(Mathf.Pow(10, -14) / curCOH) \* (-1);

                    startC = curCOH;

                    startV = startV + addedV;

                }

            }

            #endregion;

        }

    }

    public void Empty()

    {

        startV = 0;

        thisSub = "-";

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class camera360 : MonoBehaviour

{

    private GameObject thisCamera;

    public float speed = 10f;

    public float angleY, angleX;

    private float time, rot;

    void Start()

    {

        thisCamera = this.gameObject;

    }

    void Update()

    {

        time += Time.deltaTime \* speed;

        rot = angleY + time;

        if (angleY >= 180) angleY = 180;

        this.transform.localRotation = Quaternion.Euler(angleX, rot, 0);

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class cameraMovement : MonoBehaviour

{

    public float speed = 1f;

    private float posX, posY, posZ;

    private float time, mov;

    public string movAxis, movDir;

    void Start()

    {

        posX = transform.position.x;

        posY = transform.position.y;

        posZ = transform.position.z;

    }

    void Update()

    {

        time += Time.deltaTime \* speed;

        if (movDir == "+")

            mov = time;

        if(movDir == "-")

            mov = -time;

        if (movAxis == "x" || movAxis == "X")

        transform.localPosition = new Vector3 (posX + mov, posY, posZ);

        if (movAxis == "y" || movAxis == "Y")

            transform.localPosition = new Vector3(posX, posY + mov, posZ);

        if (movAxis == "z" || movAxis == "Z")

            transform.localPosition = new Vector3(posX, posY, posZ + mov);

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class connect : MonoBehaviour

{

    private GameObject lHand, rHand;

    public string connectTo;

    public Vector3 pos;

    public float rotX, rotY, rotZ;

    private bool connected, KM = false;

    void Start()

    {

        if (GameObject.Find("PlayerVR") != null)

        {

            lHand = GameObject.Find("PlayerVR").transform.Find("Controller (left)").gameObject;

            rHand = GameObject.Find("PlayerVR").transform.Find("Controller (right)").gameObject;

        }

        else

        {

            rHand = GameObject.Find("PlayerKM").transform.GetChild(0).GetChild(0).gameObject;

            Debug.Log(rHand.name);

            lHand = null;

            KM = true;

        }

    }

    private void OnTriggerEnter(Collider col)

    {

        if(col.tag == connectTo && connected == false)

        {

            if (KM == false)

            {

                lHand.GetComponent<hand>().Drop();

                lHand.GetComponent<hand>().m\_ContactInteractables.Remove(transform.GetComponent<interactable>());

                rHand.GetComponent<hand>().Drop();

                rHand.GetComponent<hand>().m\_ContactInteractables.Remove(transform.GetComponent<interactable>());

            }

            else

            {

                rHand.GetComponent<handKM>().Drop();

                rHand.GetComponent<handKM>().m\_ContactInteractables.Remove(transform.GetComponent<interactable>());

            }

            Destroy(transform.GetComponent<interactable>());

            Destroy(transform.GetComponent<Rigidbody>());

            Destroy(col.GetComponent<SphereCollider>());

            transform.SetParent(col.transform);

            transform.localPosition = pos;

            transform.localRotation = Quaternion.Euler(rotX, rotY, rotZ);

            connected = true;

        }

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class contain : MonoBehaviour

{

    public List<string> inVessel;

    private float time;

    private bool zero, done = false;

    void Start()

    {

        inVessel.Add(transform.GetChild(1).GetComponent<LiquidCore\_v1>().Type);

    }

    void Update()

    {

        time += Time.deltaTime;

        if (!done && inVessel.Contains("AgNO3") && inVessel.Contains("NaCl"))

        {

            if (!zero)

            {

                time = 0;

                zero = true;

            }

            float ratio = time / 2;

            Color curCol = transform.Find("inner").GetComponent<Renderer>().material.GetColor("\_Tint");

            if (time <= 2)

            {

                Color newCol = new Color(curCol.r, curCol.g, curCol.b, ratio);

                transform.Find("inner").GetComponent<Renderer>().material.SetColor("\_Tint", newCol);

            }

            else

            {

                done = true;

            }

        }

    }

        public void Add()

    {

        if (!inVessel.Contains(LiquidCore\_v1.added))

            inVessel.Add(LiquidCore\_v1.added);

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class fire : MonoBehaviour

{

    void Update()

    {

        float rotX = -transform.parent.rotation.eulerAngles.x;

        transform.localRotation = Quaternion.Euler(rotX, 0, 0);

        //if (transform.localRotation.eulerAngles.x >= 80 || transform.localRotation.eulerAngles.x <= -80)

            //Destroy(transform.gameObject);

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class fluorescence : MonoBehaviour

{

    private List<string> inVessel;

    private bool done, check, zero;

    private float time;

    private Material mat;

    private Color def;

    public GameObject light1, light2, light3;

    void Start()

    {

        inVessel = transform.GetComponent<contain>().inVessel;

        done = false;

        check = false;

        zero = false;

    }

    void Update()

    {

        time += Time.deltaTime;

        if (inVessel.Contains("Luciferase") && done == false)

        {

            if (!check)

            {

                mat = transform.GetChild(0).GetComponent<Renderer>().material;

                def = transform.GetChild(0).GetComponent<Renderer>().material.GetColor("\_Tint");

                time = 0;

                check = true;

            }

            float ratio = time / 2;

            Color fluor = new Color(0.2503115f, 0.6851759f, 0.7075471f, 1);

            Color change = Color.Lerp(def, fluor, ratio);

            transform.GetChild(1).GetComponent<LiquidCore\_v1>().StartColor = change;

            transform.GetChild(1).GetComponent<LiquidCore\_v1>().SetUpMaterial();

            if (time >= 2)

            {

                done = true;

            }

        }

        if (done == true)

        {

            if (!zero)

            {

                transform.GetComponent<AudioSource>().Play();

                time = 0;

                zero = true;

            }

            if (time <= 1.2)

            {

                light1.GetComponent<Light>().intensity = 1.2f - time;

                light2.GetComponent<Light>().intensity = 1.2f - time;

                light3.GetComponent<Light>().intensity = 1.2f - time;

            }

            if(time >= 20 && time <= 20 + 1.2)

            {

                if (zero)

                {

                    time = 0;

                    zero = false;

                }

                light1.GetComponent<Light>().intensity += time;

                light2.GetComponent<Light>().intensity += time;

                light3.GetComponent<Light>().intensity += time;

            }

        }

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class indicators : MonoBehaviour

{

    private List<string> inVessel;

    private float pH, time;

    public bool done;

    private bool zero = false;

    void Start()

    {

        inVessel = transform.GetComponent<contain>().inVessel;

    }

    void Update()

    {

        pH = transform.GetComponent<acidity>().pH;

        time += Time.deltaTime;

        if (inVessel.Contains("methylOrange") && !done)

        {

            Color curCol = transform.Find("inner").GetComponent<Renderer>().material.GetColor("\_Tint");

                if (pH <= 6)

                {

                    Color newCol = new Color(0.9f, 0.3f, 0.37f, 0.3f);

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().StartColor = newCol;

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().SetUpMaterial();

                done = true;

            }

                if (pH >= 8)

                {

                    Color newCol = new Color(0.89f, 0.84f, 0.29f, 0.3f);

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().StartColor = newCol;

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().SetUpMaterial();

                done = true;

            }

                if (pH >= 6 && pH <= 8)

                {

                    Color newCol = new Color(0.95f, 0.54f, 0.1f, 0.3f);

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().StartColor = newCol;

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().SetUpMaterial();

                done = true;

                }

        }

        if (inVessel.Contains("PP") && !done)

        {

            Color curCol = transform.Find("inner").GetComponent<Renderer>().material.GetColor("\_Tint");

                if (pH >= 8)

                {

                    Color newCol = new Color(0.97f, 0.52f, 0.8f, 0.3f);

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().StartColor = newCol;

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().SetUpMaterial();

                done = true;

            }

                if (pH <= 8)

                {

                    Color newCol = new Color(1f, 1f, 1f, 0.3f);

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().StartColor = newCol;

                    transform.GetChild(1).GetComponent<LiquidCore\_v1>().SetUpMaterial();

                done = true;

            }

        }

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.SceneManagement;

public class buttonScript : MonoBehaviour

{

    public Button but;

    void Start()

    {

        but = GetComponent<Button>();

    }

    public void Pressed()

    {

        but.onClick.Invoke();

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

public class buttons\_Fluor : MonoBehaviour

{

    public void Menu()

    {

        SceneManager.LoadScene("StartScene");

    }

    public void Again()

    {

        SceneManager.LoadScene("FluorRoom");

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

using UnityEngine.UI;

public class buttons\_Ind : MonoBehaviour

{

    public GameObject intro, testPan;

    void Start()

    {

        intro.SetActive(true);

        testPan.SetActive(false);

    }

    public void Menu()

    {

        SceneManager.LoadScene("StartScene");

    }

    public void Again()

    {

        SceneManager.LoadScene("IndicatorRoom");

    }

    public void Test()

    {

        testPan.SetActive(true);

        intro.SetActive(false);

    }

    public void Right()

    {

        transform.GetComponent<Image>().material = Resources.Load("Green") as Material;

    }

    public void Wrong()

    {

        transform.GetComponent<Image>().material = Resources.Load("Red") as Material;

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

public class buttons\_Sol : MonoBehaviour

{

    public void Menu()

    {

        SceneManager.LoadScene("StartScene");

    }

    public void Again()

    {

        SceneManager.LoadScene("SolutionsRoom");

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

public class buttons\_Start : MonoBehaviour

{

    public void FluorRoom()

    {

        SceneManager.LoadScene("FluorRoom");

    }

    public void LatticeRoom()

    {

        SceneManager.LoadScene("LatticeRoom");

    }

    public void IndRoom()

    {

        SceneManager.LoadScene("IndicatorRoom");

    }

    public void HRoom()

    {

        SceneManager.LoadScene("HRoom");

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

public class button\_H : MonoBehaviour

{

    public void Menu()

    {

        SceneManager.LoadScene("StartScene");

    }

    public void Again()

    {

        SceneManager.LoadScene("HRoom");

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class CuO : MonoBehaviour

{

    private Material oxideCol;

    public Material copperCol;

    private bool inArea = false;

    private float time;

    void Start()

    {

        oxideCol = transform.GetComponent<MeshRenderer>().material;

    }

    void Update()

    {

        if (inArea)

            time += Time.deltaTime;

        float ratio = time / 10;

        Color newColor = Color.Lerp(oxideCol.color, copperCol.color, ratio);

        transform.GetComponent<MeshRenderer>().material.color = newColor;

    }

    void OnTriggerEnter(Collider col)

    {

        if (col.tag == "Fire")

            inArea = true;

    }

    void OnTriggerExit(Collider col)

    {

        if (col.tag == "Fire")

            inArea = false;

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class HExit : MonoBehaviour

{

    void OnTriggerEnter(Collider col)

    {

        if(col.tag == "Tube" && !col.GetComponent<contain>().inVessel.Contains("H2"))

        {

            col.GetComponent<contain>().inVessel.Add("H2");

        }

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class HTrig : MonoBehaviour

{

    public GameObject CuO;

    public GameObject HExit;

    void Start()

    {

        var ps = transform.Find("Gas").GetComponent<ParticleSystem>().emission;

        ps.enabled = false;

        CuO.GetComponent<SphereCollider>().enabled = false;

        HExit.GetComponent<SphereCollider>().enabled = false;

    }

    void Update()

    {

        if (transform.GetComponent<contain>().inVessel.Contains("HCl"))

        {

            var ps = transform.Find("Gas").GetComponent<ParticleSystem>().emission;

            ps.enabled = true;

            if (transform.Find("Bung") != null && transform.parent.parent.tag == "Holder")

            {

                Debug.Log("ok");

                CuO.GetComponent<SphereCollider>().enabled = true;

                HExit.GetComponent<SphereCollider>().enabled = true;

            }

        }

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class filler : MonoBehaviour

{

    public GameObject playerCamera;

    public static float globalVolume;

    public Transform grabbedObj;

    public string liquidType;

    private bool planeVis, filling = false;

    public bool empty = true;

    private string volumeStr;

    private int volumeInt;

    public float volume;

    private float mouseScroll, scrollInput, volumeAdd, addAmount;

    private GameObject liquid;

    private Material mat;

    public GameObject text, plane;

    void Start()

    {

    }

    void Update()

    {

        plane.SetActive(planeVis);

        grabbedObj = playerCamera.GetComponent<interHead>().grabbedObj;

        mouseScroll = Input.GetAxis("Mouse ScrollWheel");

        scrollInput += mouseScroll \* 100;

        if (filling == true)

        {

            if (volume == 0) empty = true;

            else empty = false;

            volume = grabbedObj.GetComponent<volume>().curVolume;

            volumeInt = Mathf.FloorToInt(scrollInput);

            if (scrollInput >= 100) scrollInput = 100;

            if (scrollInput <= 0) scrollInput = 0;

            volumeStr = scrollInput.ToString();

            text.GetComponent<TextMesh>().text = "–Э–∞–±—А–∞—В—М: " + volumeStr + " –Љ–ї";

            if (Input.GetKeyDown(KeyCode.E))

            {

                if (empty == true)

                {

                    liquid = Instantiate(Resources.Load<GameObject>("Flask"));

                    mat = Instantiate(Resources.Load<Material>(liquidType));

                    liquid.name = "Liquid";

                    liquid.transform.SetParent(grabbedObj);

                    liquid.transform.localScale = new Vector3(0.9f, 0.9f, 0.9f);

                    liquid.transform.localPosition = new Vector3(0f, 0f, 0f);

                    liquid.transform.localRotation = Quaternion.Euler(0, 0, 0);

                    liquid.GetComponent<MeshRenderer>().material = mat;

                }

                else

                {

                    liquid = grabbedObj.Find("Liquid").gameObject;

                    mat = liquid.GetComponent<Renderer>().material;

                }

                    addAmount = 0;

                    addAmount = scrollInput + volume;

                    if (addAmount >= 100) addAmount = 100;

                    volumeAdd = 1f - (((addAmount) / 100f) - 0.1f);

                    mat.SetFloat("\_FillAmount", volumeAdd);

                    globalVolume = scrollInput;

                    grabbedObj.GetComponent<volume>().addVolume();

                    planeVis = false;

                    plane.SetActive(planeVis);

                    filling = false;

                    grabbedObj = null;

                    liquid = null;

            }

        }

    }

    public void Fill()

    {

        scrollInput = 0;

        planeVis = true;

        plane.SetActive(planeVis);

        filling = true;

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class handKM : MonoBehaviour

{

    private FixedJoint m\_Joint = null;

    private interactable m\_CurrentInteractable = null;

    public List<interactable> m\_ContactInteractables = new List<interactable>();

    private Transform laser, glove, glove\_A;

    private GameObject obj;

    public List<string> draggableItems;

    private bool laserVis, grabbed = false;

    private float rotate;

    private void Awake()

    {

        m\_Joint = GetComponent<FixedJoint>();

    }

    void Start()

    {

        laser = transform.GetChild(0);

        glove = transform.GetChild(1);

        glove\_A = transform.GetChild(2);

        glove\_A.gameObject.SetActive(false);

        laser.gameObject.SetActive(false);

    }

    void Update()

    {

        if (Input.GetMouseButtonDown(2))

        {

            laserVis = !laserVis;

            laser.gameObject.SetActive(laserVis);

            glove.gameObject.SetActive(!laserVis);

            glove\_A.gameObject.SetActive(laserVis);

            grabbed = true;

        }

        if (!grabbed && Input.GetMouseButtonDown(0))

        {

            PickUp();

            glove.gameObject.SetActive(false);

            glove\_A.gameObject.SetActive(true);

        }

        if (grabbed)

        {

            if (Input.GetKeyDown(KeyCode.E))

            {

                Drop();

                glove.gameObject.SetActive(true);

                glove\_A.gameObject.SetActive(false);

            }

            if (Input.GetKeyDown(KeyCode.R))

            {

                Rigidbody targetBody = m\_CurrentInteractable.GetComponent<Rigidbody>();

                targetBody.AddForce(transform.forward \* 800);

                Drop();

                glove.gameObject.SetActive(true);

                glove\_A.gameObject.SetActive(false);

            }

            if (Input.GetMouseButton(1))

            {

                rotate += Input.GetAxisRaw("Mouse Y") \* 20;

                transform.localRotation = Quaternion.Euler(0, 0, rotate);

                transform.parent.GetComponent<mouseView>().enabled = false;

            }

            else transform.parent.GetComponent<mouseView>().enabled = true;

        }

    }

    public void OnTriggerEnter(Collider col)

    {

        obj = col.gameObject;

        bool draggable = draggableItems.Contains(obj.tag);

        if (!draggable) return;

        m\_ContactInteractables.Add(obj.GetComponent<interactable>());

    }

    public void OnTriggerExit(Collider col)

    {

        obj = col.gameObject;

        m\_ContactInteractables.Remove(obj.GetComponent<interactable>());

    }

    private void PickUp()

    {

        m\_CurrentInteractable = GetNearestInteractable();

        grabbed = true;

        if (!m\_CurrentInteractable) return;

        if (m\_CurrentInteractable.m\_ActiveHand\_KM)

        {

            m\_CurrentInteractable.m\_ActiveHand\_KM.Drop();

        }

            m\_CurrentInteractable.transform.position = transform.position;

        m\_CurrentInteractable.transform.localRotation = Quaternion.Euler(0,0,0);

        Rigidbody targetBody = m\_CurrentInteractable.GetComponent<Rigidbody>();

            m\_Joint.connectedBody = targetBody;

            m\_CurrentInteractable.m\_ActiveHand\_KM = this;

    }

    public void Drop ()

    {

        grabbed = false;

        if (!m\_CurrentInteractable)

            return;

        Rigidbody targetBody = m\_CurrentInteractable.GetComponent<Rigidbody>();

        targetBody.AddForce(targetBody.velocity \* 1000);

        m\_Joint.connectedBody = null;

        m\_CurrentInteractable.m\_ActiveHand\_KM = null;

        m\_CurrentInteractable = null;

        grabbed = false;

        transform.localRotation = Quaternion.Euler(0, 0, 0);

        rotate = 0;

    }

    private interactable GetNearestInteractable()

    {

        interactable nearest = null;

        float minDistance = float.MaxValue;

        float distance = 0.0f;

        foreach(interactable interactable in m\_ContactInteractables)

        {

            if (interactable == null)

            {

                Debug.Log("break");

            }

            else

            {

                distance = (interactable.transform.position - transform.position).sqrMagnitude;

                if (distance < minDistance)

                {

                    minDistance = distance;

                    nearest = interactable;

                }

            }

        }

        return nearest;

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class interHead : MonoBehaviour

{

    public GameObject playerCamera;

    public string rayTag;

    public int rayLenght = 30;

    public List<string> DraggableItems;

    public List<string> FillerItems;

    #region dragObj

    public bool grabbed = false;

    public Transform grabbedObj;

    public float throwForce = 1000;

    private Vector3 grabPos, curGrabPos;

    private Quaternion grabRot, curGrabRot;

    public float objX = 1f;

    public float objY = -1f;

    public float objZ = 4f;

    public float smoothTime = 2;

    private float maxDeltaPos = 0f;

    private float maxDeltaRot = 0f;

    private float ratio, smooth;

    #endregion

    void Start()

    {

        grabPos = new Vector3(objX, objY, objZ);

        curGrabPos = new Vector3(objX, objY, objZ);

        grabRot = Quaternion.Euler(-90, 180, 0);

        curGrabRot = Quaternion.Euler(-90, 180, 0);

    }

    void Update()

    {

        RaycastHit hit;

        Vector3 dir = transform.TransformDirection(Vector3.forward);

        if (Physics.Raycast(transform.position, dir, out hit, rayLenght))

        {

            rayTag = hit.collider.gameObject.tag;

            bool draggable = DraggableItems.Contains(rayTag);

            bool filler = FillerItems.Contains(rayTag);

            if (draggable == true && grabbed == false && Input.GetMouseButtonDown(0) == true)

            {

                grabbedObj = hit.collider.transform;

                grabbedObj.parent = playerCamera.transform;

                grabbedObj.GetComponent<Rigidbody>().useGravity = false;

                grabbedObj.localPosition = grabPos;

                grabbedObj.localRotation = grabRot;

                grabbed = true;

            }

            if (filler == true && grabbed == true && Input.GetMouseButtonDown(0) == true)

            {

                hit.collider.GetComponent<filler>().Fill();

            }

        }

        if (grabbed == true)

        {

            #region dragObj

            // –Є–Ј–Љ–µ—А–µ–љ–Є–µ –Њ—В–Ї–ї–Њ–љ–µ–љ–Є—П –њ—А–µ–і–Љ–µ—В–∞ –Њ—В –љ–∞—З–∞–ї—М–љ–Њ–є –њ–Њ–Ј–Є—Ж–Є–Є

            curGrabPos = grabbedObj.localPosition;

            curGrabRot = grabbedObj.localRotation;

            float deltaPos = Vector3.Distance(curGrabPos, grabPos);

            float deltaRot = Vector3.Angle(curGrabPos, grabPos);

            smooth += Time.deltaTime;

            ratio = smooth / smoothTime;

            if (smooth >= smoothTime)

            {

                smooth = 0;

            }

            if (deltaPos >= maxDeltaPos || deltaRot >= maxDeltaRot)

            {

                grabbedObj.localPosition = Vector3.Lerp(curGrabPos, grabPos, ratio);

                grabbedObj.localRotation = Quaternion.Lerp(curGrabRot, grabRot, ratio);

                // —Г–±—А–∞—В—М –і–µ–є—Б—В–≤—Г—О—Й–Є–µ —Б–Є–ї—Л

                grabbedObj.GetComponent<Rigidbody>().isKinematic = true;

                grabbedObj.GetComponent<Rigidbody>().isKinematic = false;

                smooth = 0;

            }

            // –Њ—В–њ—Г—Б—В–Є—В—М –њ—А–µ–і–Љ–µ—В

            if (Input.GetKeyDown(KeyCode.Q))

            {

                grabbedObj.GetComponent<Rigidbody>().useGravity = true;

                grabbedObj.transform.parent = null;

                grabbedObj = null;

                grabbed = false;

            }

            // –Ї–Є–љ—Г—В—М –њ—А–µ–і–Љ–µ—В

            if (Input.GetKeyDown(KeyCode.R))

            {

                grabbedObj.GetComponent<Rigidbody>().AddForce(dir \* throwForce);

                grabbedObj.GetComponent<Rigidbody>().useGravity = true;

                grabbedObj.transform.parent = null;

                grabbedObj = null;

                grabbed = false;

            }

            #endregion

        }

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class laserPointer : MonoBehaviour

{

    void Update()

    {

        RaycastHit hit;

        if (Physics.Raycast(transform.position, transform.forward, out hit))

        {

            Transform colObj = hit.collider.transform;

            float distance = Vector3.Distance(colObj.position, transform.position);

            Vector3 pos = new Vector3(0, 0, distance);

            GetComponent<LineRenderer>().SetPosition(1, pos);

            if (hit.collider.tag == "Button" && Input.GetMouseButtonDown(0))

            {

                hit.collider.GetComponent<buttonScript>().Pressed();

            }

        }

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class lightSwitcher : MonoBehaviour

{

    public GameObject fumeLight;

    public bool switched = false;

    void Start()

    {

        fumeLight.SetActive(switched);

    }

    public void SwitchLight ()

    {

        switched = !switched;

        fumeLight.SetActive(switched);

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class mouseView : MonoBehaviour

{

   public float mouseSensitivity = 5f;

   public float mouseSmoothing = 2f;

   public float maxAngle = 40;

    public float minAngle = -60;

   GameObject player;

   Vector2 mouseLook;

   Vector2 smooth;

    void Start()

    {

        player = this.transform.parent.gameObject;

    }

    void Update()

    {

        var mouseDelta = new Vector2(Input.GetAxisRaw("Mouse X"), Input.GetAxisRaw("Mouse Y"));

        mouseDelta = Vector2.Scale(mouseDelta, new Vector2(mouseSensitivity \* mouseSmoothing, mouseSensitivity \* mouseSmoothing));

        smooth.x = Mathf.Lerp(smooth.x, mouseDelta.x, 1f / mouseSmoothing);

        smooth.y = Mathf.Lerp(smooth.y, mouseDelta.y, 1f / mouseSmoothing);

        mouseLook += smooth;

        if (mouseLook.y >= maxAngle)

        {

            mouseLook.y = maxAngle;

        }

        if (mouseLook.y <= minAngle)

        {

            mouseLook.y = minAngle;

        }

        transform.localRotation = Quaternion.AngleAxis(-mouseLook.y, Vector3.right);

        player.transform.localRotation = Quaternion.AngleAxis(mouseLook.x, player.transform.up);

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class openObject : MonoBehaviour

{

    public Animator anim;

    public bool open = false;

    void Start()

    {

        anim = GetComponent<Animator>();

        anim.SetBool("open", open);

    }

    public void Opening() {

        open = !open;

        anim.SetBool("open", open);

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class playerController : MonoBehaviour

{

    public float playerSpeed = 10f;

    public float vertical;

    public float horizontal;

    void Start()

    {

        Cursor.lockState = CursorLockMode.Locked;

    }

    void Update()

    {

        vertical = Input.GetAxis("Vertical") \* playerSpeed \* Time.deltaTime;

        horizontal = Input.GetAxis("Horizontal") \* playerSpeed \* Time.deltaTime;

        transform.Translate(horizontal, 0, vertical);

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class tutorial : MonoBehaviour

{

    public GameObject text1, text2, text3;

    void Start()

    {

    }

    void Update()

    {

    }

    public void Begin1()

    {

        text1.SetActive(false);

        text2.SetActive(true);

    }

    public void Begin2()

    {

        text2.SetActive(false);

        text3.SetActive(true);

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class volume : MonoBehaviour

{

    public float curVolume;

    void Start()

    {

        curVolume = 0;

    }

    public void addVolume()

    {

        curVolume += filler.globalVolume;

    }

}

пїњ

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

[RequireComponent(typeof(Rigidbody))]

public class interactable : MonoBehaviour

{

    [HideInInspector]

    public hand m\_ActiveHand = null;

    public handKM m\_ActiveHand\_KM = null;

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

public class laserPointerVR : hand

{

    private GameObject rHand;

    public Material mat;

    void Start()

    {

        rHand = GameObject.Find("PlayerVR").transform.Find("Controller (right)").gameObject;

    }

    void Update()

    {

        m\_GrabAction = rHand.GetComponent<hand>().m\_GrabAction;

        m\_Pose = rHand.GetComponent<hand>().m\_Pose;

        RaycastHit hit;

        if (Physics.Raycast(transform.position, transform.forward, out hit))

        {

            Transform colObj = hit.collider.transform;

            float distance = Vector3.Distance(colObj.position, transform.position);

            Vector3 pos = new Vector3(0, 0, distance);

            GetComponent<LineRenderer>().SetPosition(1, pos);

            if (hit.collider.tag == "Button" && m\_GrabAction.GetStateDown(m\_Pose.inputSource))

            {

                string butName = hit.collider.name;

                hit.collider.GetComponent<buttonScript>().Pressed();

            }

        }

    }

}

пїњusing System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class lattices : MonoBehaviour

{

    public GameObject NaCl, CsCl;

    private GameObject obj;

    void Start()

    {

        NaCl.SetActive(false);

        CsCl.SetActive(false);

    }

    void Update()

    {

        obj = transform.GetComponent<hand>().obj;

        if (obj.tag == "Lattice")

        {

            string lattice = obj.name;

            if (lattice == "NaCl")

                NaCl.SetActive(true);

            else

                NaCl.SetActive(false);

            if (lattice == "CsCl")

                CsCl.SetActive(true);

            else

                CsCl.SetActive(false);

        }

    }

}

пїњusing UnityEngine;

using System.Collections;

namespace Valve.VR

{

    public class SteamVR\_Windows\_Editor\_Helper

    {

        public enum BrowserApplication

        {

            Unknown,

            InternetExplorer,

            Firefox,

            Chrome,

            Opera,

            Safari,

            Edge,

        }

        public static BrowserApplication GetDefaultBrowser()

        {

#if UNITY\_EDITOR

    #if UNITY\_STANDALONE\_WIN

            const string userChoice = @"Software\Microsoft\Windows\Shell\Associations\UrlAssociations\http\UserChoice";

            using (Microsoft.Win32.RegistryKey userChoiceKey = Microsoft.Win32.Registry.CurrentUser.OpenSubKey(userChoice))

            {

                if (userChoiceKey == null)

                {

                    return BrowserApplication.Unknown;

                }

                object progIdValue = userChoiceKey.GetValue("Progid");

                if (progIdValue == null)

                {

                    return BrowserApplication.Unknown;

                }

                string browserId = progIdValue.ToString().ToLower();

                if (browserId.Contains("ie.http"))

                    return BrowserApplication.InternetExplorer;

                else if (browserId.Contains("firefox"))

                    return BrowserApplication.Firefox;

                else if (browserId.Contains("chrome"))

                    return BrowserApplication.Chrome;

                else if (browserId.Contains("opera"))

                    return BrowserApplication.Opera;

                else if (browserId.Contains("safari"))

                    return BrowserApplication.Safari;

                else if (browserId.Contains("appcq0fevzme2pys62n3e0fbqa7peapykr8v")) //AppXq0fevzme2pys62n3e0fbqa7peapykr8v

                    return BrowserApplication.Edge;

                else

                    return BrowserApplication.Unknown;

            }

    #else

            return BrowserApplication.Firefox;

    #endif

#else

            return BrowserApplication.Firefox;

#endif

        }

    }

}

пїњusing UnityEngine;

using UnityEngine.UI;

using System.Collections;

using TMPro;

public class ChatController : MonoBehaviour {

    public TMP\_InputField TMP\_ChatInput;

    public TMP\_Text TMP\_ChatOutput;

    public Scrollbar ChatScrollbar;

    void OnEnable()

    {

        TMP\_ChatInput.onSubmit.AddListener(AddToChatOutput);

    }

    void OnDisable()

    {

        TMP\_ChatInput.onSubmit.RemoveListener(AddToChatOutput);

    }

    void AddToChatOutput(string newText)

    {

        // Clear Input Field

        TMP\_ChatInput.text = string.Empty;

        var timeNow = System.DateTime.Now;

        TMP\_ChatOutput.text += "[<#FFFF80>" + timeNow.Hour.ToString("d2") + ":" + timeNow.Minute.ToString("d2") + ":" + timeNow.Second.ToString("d2") + "</color>] " + newText + "\n";

        TMP\_ChatInput.ActivateInputField();

        // Set the scrollbar to the bottom when next text is submitted.

        ChatScrollbar.value = 0;

    }

}

пїњ

пїњusing UnityEngine;

namespace TMPro.Examples

{

    public class TMP\_TextEventCheck : MonoBehaviour

    {

        public TMP\_TextEventHandler TextEventHandler;

        void OnEnable()

        {

            if (TextEventHandler != null)

            {

                TextEventHandler.onCharacterSelection.AddListener(OnCharacterSelection);

                TextEventHandler.onSpriteSelection.AddListener(OnSpriteSelection);

                TextEventHandler.onWordSelection.AddListener(OnWordSelection);

                TextEventHandler.onLineSelection.AddListener(OnLineSelection);

                TextEventHandler.onLinkSelection.AddListener(OnLinkSelection);

            }

        }

        void OnDisable()

        {

            if (TextEventHandler != null)

            {

                TextEventHandler.onCharacterSelection.RemoveListener(OnCharacterSelection);

                TextEventHandler.onSpriteSelection.RemoveListener(OnSpriteSelection);

                TextEventHandler.onWordSelection.RemoveListener(OnWordSelection);

                TextEventHandler.onLineSelection.RemoveListener(OnLineSelection);

                TextEventHandler.onLinkSelection.RemoveListener(OnLinkSelection);

            }

        }

        void OnCharacterSelection(char c, int index)

        {

            Debug.Log("Character [" + c + "] at Index: " + index + " has been selected.");

        }

        void OnSpriteSelection(char c, int index)

        {

            Debug.Log("Sprite [" + c + "] at Index: " + index + " has been selected.");

        }

        void OnWordSelection(string word, int firstCharacterIndex, int length)

        {

            Debug.Log("Word [" + word + "] with first character index of " + firstCharacterIndex + " and length of " + length + " has been selected.");

        }

        void OnLineSelection(string lineText, int firstCharacterIndex, int length)

        {

            Debug.Log("Line [" + lineText + "] with first character index of " + firstCharacterIndex + " and length of " + length + " has been selected.");

        }

        void OnLinkSelection(string linkID, string linkText, int linkIndex)

        {

            Debug.Log("Link Index: " + linkIndex + " with ID [" + linkID + "] and Text \"" + linkText + "\" has been selected.");

        }

    }

}

пїњ

пїњusing UnityEngine;

using System.Collections;

namespace TMPro.Examples

{

    public class TMP\_UiFrameRateCounter : MonoBehaviour

    {

        public float UpdateInterval = 5.0f;

        private float m\_LastInterval = 0;

        private int m\_Frames = 0;

        public enum FpsCounterAnchorPositions { TopLeft, BottomLeft, TopRight, BottomRight };

        public FpsCounterAnchorPositions AnchorPosition = FpsCounterAnchorPositions.TopRight;

        private string htmlColorTag;

        private const string fpsLabel = "{0:2}</color> <#8080ff>FPS \n<#FF8000>{1:2} <#8080ff>MS";

        private TextMeshProUGUI m\_TextMeshPro;

        private RectTransform m\_frameCounter\_transform;

        private FpsCounterAnchorPositions last\_AnchorPosition;

        void Awake()

        {

            if (!enabled)

                return;

            Application.targetFrameRate = -1;

            GameObject frameCounter = new GameObject("Frame Counter");

            m\_frameCounter\_transform = frameCounter.AddComponent<RectTransform>();

            m\_frameCounter\_transform.SetParent(this.transform, false);

            m\_TextMeshPro = frameCounter.AddComponent<TextMeshProUGUI>();

            m\_TextMeshPro.font = Resources.Load<TMP\_FontAsset>("Fonts & Materials/LiberationSans SDF");

            m\_TextMeshPro.fontSharedMaterial = Resources.Load<Material>("Fonts & Materials/LiberationSans SDF - Overlay");

            m\_TextMeshPro.enableWordWrapping = false;

            m\_TextMeshPro.fontSize = 36;

            m\_TextMeshPro.isOverlay = true;

            Set\_FrameCounter\_Position(AnchorPosition);

            last\_AnchorPosition = AnchorPosition;

        }

        void Start()

        {

            m\_LastInterval = Time.realtimeSinceStartup;

            m\_Frames = 0;

        }

        void Update()

        {

            if (AnchorPosition != last\_AnchorPosition)

                Set\_FrameCounter\_Position(AnchorPosition);

            last\_AnchorPosition = AnchorPosition;

            m\_Frames += 1;

            float timeNow = Time.realtimeSinceStartup;

            if (timeNow > m\_LastInterval + UpdateInterval)

            {

                // display two fractional digits (f2 format)

                float fps = m\_Frames / (timeNow - m\_LastInterval);

                float ms = 1000.0f / Mathf.Max(fps, 0.00001f);

                if (fps < 30)

                    htmlColorTag = "<color=yellow>";

                else if (fps < 10)

                    htmlColorTag = "<color=red>";

                else

                    htmlColorTag = "<color=green>";

                m\_TextMeshPro.SetText(htmlColorTag + fpsLabel, fps, ms);

                m\_Frames = 0;

                m\_LastInterval = timeNow;

            }

        }

        void Set\_FrameCounter\_Position(FpsCounterAnchorPositions anchor\_position)

        {

            switch (anchor\_position)

            {

                case FpsCounterAnchorPositions.TopLeft:

                    m\_TextMeshPro.alignment = TextAlignmentOptions.TopLeft;

                    m\_frameCounter\_transform.pivot = new Vector2(0, 1);

                    m\_frameCounter\_transform.anchorMin = new Vector2(0.01f, 0.99f);

                    m\_frameCounter\_transform.anchorMax = new Vector2(0.01f, 0.99f);

                    m\_frameCounter\_transform.anchoredPosition = new Vector2(0, 1);

                    break;

                case FpsCounterAnchorPositions.BottomLeft:

                    m\_TextMeshPro.alignment = TextAlignmentOptions.BottomLeft;

                    m\_frameCounter\_transform.pivot = new Vector2(0, 0);

                    m\_frameCounter\_transform.anchorMin = new Vector2(0.01f, 0.01f);

                    m\_frameCounter\_transform.anchorMax = new Vector2(0.01f, 0.01f);

                    m\_frameCounter\_transform.anchoredPosition = new Vector2(0, 0);

                    break;

                case FpsCounterAnchorPositions.TopRight:

                    m\_TextMeshPro.alignment = TextAlignmentOptions.TopRight;

                    m\_frameCounter\_transform.pivot = new Vector2(1, 1);

                    m\_frameCounter\_transform.anchorMin = new Vector2(0.99f, 0.99f);

                    m\_frameCounter\_transform.anchorMax = new Vector2(0.99f, 0.99f);

                    m\_frameCounter\_transform.anchoredPosition = new Vector2(1, 1);

                    break;

                case FpsCounterAnchorPositions.BottomRight:

                    m\_TextMeshPro.alignment = TextAlignmentOptions.BottomRight;

                    m\_frameCounter\_transform.pivot = new Vector2(1, 0);

                    m\_frameCounter\_transform.anchorMin = new Vector2(0.99f, 0.01f);

                    m\_frameCounter\_transform.anchorMax = new Vector2(0.99f, 0.01f);

                    m\_frameCounter\_transform.anchoredPosition = new Vector2(1, 0);

                    break;

            }

        }

    }

}

пїњ

пїњusing System;

using System.Collections.Generic;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [Serializable]

    class AspectRatio

    {

        [SerializeField] float m\_CustomAspectX = 1.0f;

        [SerializeField] float m\_CustomAspectY = 1.0f;

        [SerializeField] ImageAspect m\_ImageAspect = ImageAspect.x16\_9;

        internal static readonly Dictionary<ImageAspect, float> s\_AspectToValue = new Dictionary<ImageAspect, float>

        {

            { ImageAspect.x16\_9, 16.0f / 9.0f },

            { ImageAspect.x16\_10, 16.0f / 10.0f },

            { ImageAspect.x19\_10, 19.0f / 10.0f },

            { ImageAspect.x5\_4, 5.0f / 4.0f },

            { ImageAspect.x4\_3, 4.0f / 3.0f },

            { ImageAspect.x3\_2, 3.0f / 2.0f },

            { ImageAspect.x1\_1, 1.0f }

        };

        public float GetAspect()

        {

            return m\_ImageAspect == ImageAspect.Custom ? m\_CustomAspectX / m\_CustomAspectY : s\_AspectToValue[m\_ImageAspect];

        }

    }

}

пїњusing System;

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [CustomPropertyDrawer(typeof(AspectRatio))]

    class AspectRatioPropertyDrawer : PropertyDrawer

    {

        SerializedProperty m\_CustomAspectX;

        SerializedProperty m\_CustomAspectY;

        SerializedProperty m\_ImageAspect;

        bool m\_Initialized;

        const string k\_Format = "0.0###";

        static GUIContent[] s\_DisplayNames;

        static readonly Dictionary<ImageAspect, string> s\_AspectToName = new Dictionary<ImageAspect, string>

        {

            { ImageAspect.x19\_10, "19:10 (" + AspectRatio.s\_AspectToValue[ImageAspect.x19\_10].ToString(k\_Format) + ")"},

            { ImageAspect.x16\_9, "16:9 (" + AspectRatio.s\_AspectToValue[ImageAspect.x16\_9].ToString(k\_Format) + ")"},

            { ImageAspect.x16\_10, "16:10 (" + AspectRatio.s\_AspectToValue[ImageAspect.x16\_10].ToString(k\_Format) + ")"},

            { ImageAspect.x5\_4, "5:4 (" + AspectRatio.s\_AspectToValue[ImageAspect.x5\_4].ToString(k\_Format) + ")"},

            { ImageAspect.x4\_3, "4:3 (" + AspectRatio.s\_AspectToValue[ImageAspect.x4\_3].ToString(k\_Format) + ")"},

            { ImageAspect.x3\_2, "3:2 (" + AspectRatio.s\_AspectToValue[ImageAspect.x3\_2].ToString(k\_Format) + ")"},

            { ImageAspect.x1\_1, "1:1 (" + AspectRatio.s\_AspectToValue[ImageAspect.x1\_1].ToString(k\_Format) + ")"},

            { ImageAspect.Custom, "Custom"}

        };

        void Initialize(SerializedProperty property)

        {

            if (m\_Initialized )

                return;

            m\_Initialized = true;

            m\_CustomAspectX = property.FindPropertyRelative("m\_CustomAspectX");

            m\_CustomAspectY = property.FindPropertyRelative("m\_CustomAspectY");

            m\_ImageAspect = property.FindPropertyRelative("m\_ImageAspect");

            if (s\_DisplayNames == null)

            {

                s\_DisplayNames = ((ImageAspect[]) Enum.GetValues(typeof(ImageAspect))).Select(e => new GUIContent(s\_AspectToName[e])).ToArray();

            }

        }

        public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)

        {

            Initialize(property);

            EditorGUI.BeginProperty(position, label, property);

            m\_ImageAspect.intValue = EditorGUI.Popup(position, label, m\_ImageAspect.intValue, s\_DisplayNames);

            EditorGUI.EndProperty();

            var aspect = (ImageAspect) m\_ImageAspect.intValue;

            if (aspect == ImageAspect.Custom)

            {

                CustomAspectField();

            }

        }

        void CustomAspectField()

        {

            var r = EditorGUILayout.GetControlRect();

            var rContent = r;

            rContent.xMin = r.x + EditorGUIUtility.labelWidth;

            EditorGUI.BeginChangeCheck();

            int indentLevel = EditorGUI.indentLevel;

            var labelWidth = EditorGUIUtility.labelWidth;

            EditorGUI.indentLevel = 0;

            EditorGUIUtility.labelWidth = 0;

            const float columnWidth = 12.0f;

            var w = Mathf.Max(30.0f, (rContent.width - columnWidth) / 3.0f);

            var rCurrent = rContent;

            rCurrent.width = w;

            var x = EditorGUI.FloatField(rCurrent, m\_CustomAspectX.floatValue);

            rCurrent.x += w;

            rCurrent.width = columnWidth;

            EditorGUI.LabelField(rCurrent, ":");

            rCurrent.x += columnWidth;

            rCurrent.width = w;

            var y  = EditorGUI.FloatField(rCurrent, m\_CustomAspectY.floatValue );

            if (EditorGUI.EndChangeCheck())

            {

                m\_CustomAspectX.floatValue = Mathf.Max(x, 1.0f);

                m\_CustomAspectY.floatValue = Mathf.Max(y, 1.0f);

            }

            var aspect = m\_CustomAspectX.floatValue / m\_CustomAspectY.floatValue;

            rCurrent.xMin = rCurrent.xMax + 3.0f;

            rCurrent.xMax = r.xMax;

            EditorGUI.LabelField(rCurrent, "(" + aspect.ToString(k\_Format) + ")");

            EditorGUI.indentLevel = indentLevel;

            EditorGUIUtility.labelWidth = labelWidth;

        }

    }

}

using UnityEngine;

namespace UnityEditor.Recorder

{

    abstract class BaseRenderTextureInput : RecorderInput

    {

        public RenderTexture outputRT { get; set; }

        public int outputWidth { get; protected set; }

        public int outputHeight { get; protected set; }

        protected void ReleaseBuffer()

        {

            if (outputRT != null)

            {

                if (outputRT == RenderTexture.active)

                    RenderTexture.active = null;

                outputRT.Release();

                outputRT = null;

            }

        }

        protected override void Dispose(bool disposing)

        {

            base.Dispose(disposing);

            if (disposing)

                ReleaseBuffer();

        }

    }

}

using System;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Experimental.UIElements;

namespace UnityEditor.Recorder

{

    class EditableLabel : VisualElement

    {

        readonly Label m\_Label;

        readonly TextField m\_TextField;

        bool m\_IsEditing;

        Action<string> m\_OnValueChangedCallback;

        Focusable m\_PreviouslyFocused;

        internal string text

        {

            get { return m\_Label.text; }

            set { m\_Label.text = value; }

        }

        internal void SetLabelEnabled(bool value)

        {

            m\_Label.SetEnabled(value);

        }

        internal EditableLabel()

        {

            m\_IsEditing = false;

            m\_Label = new Label();

            m\_TextField = new TextField();

            UIElementHelper.SetFlex(this, 1.0f);

            UIElementHelper.SetFlex(m\_TextField, 1.0f);

            Add(m\_Label);

            RegisterCallback<KeyUpEvent>(OnKeyUpCallback, Capture.Capture);

            m\_TextField.RegisterCallback<FocusOutEvent>(OnTextFieldLostFocus);

        }

        void SetValueAndNotify(string newValue)

        {

            if (EqualityComparer<string>.Default.Equals(m\_Label.text, newValue))

                return;

            if (string.IsNullOrEmpty(newValue))

                return;

            m\_Label.text = newValue;

            if (m\_OnValueChangedCallback != null)

                m\_OnValueChangedCallback.Invoke(newValue);

        }

        internal void OnValueChanged(Action<string> callback)

        {

            m\_OnValueChangedCallback = callback;

        }

        internal void StartEditing()

        {

            if (m\_IsEditing)

                return;

            m\_IsEditing = true;

            m\_TextField.value = m\_Label.text;

            Remove(m\_Label);

            Add(m\_TextField);

            m\_TextField.focusIndex = 0;

            m\_PreviouslyFocused = focusController.focusedElement;

            m\_TextField.Focus();

        }

        void ApplyEditing()

        {

            if (!m\_IsEditing)

                return;

            SetValueAndNotify(m\_TextField.text);

            m\_IsEditing = false;

            Remove(m\_TextField);

            Add(m\_Label);

        }

        void CancelEditing()

        {

            if (!m\_IsEditing)

                return;

            m\_IsEditing = false;

            Remove(m\_TextField);

            Add(m\_Label);

        }

        void OnTextFieldLostFocus(FocusOutEvent evt)

        {

            ApplyEditing();

        }

        void OnKeyUpCallback(KeyUpEvent evt)

        {

            if (!m\_IsEditing)

                return;

            if (evt.keyCode == KeyCode.Return || evt.keyCode == KeyCode.KeypadEnter)

            {

                ApplyEditing();

                RestorePreviousFocus();

                evt.StopImmediatePropagation();

            }

            else if (evt.keyCode == KeyCode.Escape)

            {

                CancelEditing();

                RestorePreviousFocus();

                evt.StopImmediatePropagation();

            }

        }

        void RestorePreviousFocus()

        {

            if (m\_PreviouslyFocused != null)

                m\_PreviouslyFocused.Focus();

        }

    }

}

using System;

using System.Collections.Generic;

using UnityEngine;

namespace UnityEditor.Recorder

{

    abstract class EnumProperyDrawer<T> : PropertyDrawer

    {

        GUIContent[] m\_DisplayNames;

        public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)

        {

            if (m\_DisplayNames == null)

            {

                var displayNames = new List<GUIContent>();

                foreach (T frameRate in Enum.GetValues(typeof(T)))

                {

                    displayNames.Add(new GUIContent(ToLabel(frameRate)));

                }

                m\_DisplayNames = displayNames.ToArray();

            }

            EditorGUI.BeginProperty(position, label, property);

            property.intValue = EditorGUI.Popup(position, label, property.intValue, m\_DisplayNames);

            EditorGUI.EndProperty();

        }

        protected abstract string ToLabel(T value);

    }

}

пїњusing System;

using System.Collections.Generic;

using System.Globalization;

using System.IO;

using System.Linq;

using System.Text.RegularExpressions;

using UnityEditor.Recorder.Input;

using UnityEngine;

using UnityEngine.SceneManagement;

namespace UnityEditor.Recorder

{

    class Wildcard

    {

        readonly string m\_Pattern;

        readonly string m\_Label;

        readonly Func<RecordingSession, string> m\_Resolver;

        public string pattern { get { return m\_Pattern; } }

        public string label { get { return m\_Label; } }

        internal Wildcard(string pattern, Func<RecordingSession, string> resolver, string info = null)

        {

            m\_Pattern = pattern;

            m\_Label = m\_Pattern;

            if (info != null)

                m\_Label += " " + info;

            m\_Resolver = resolver;

        }

        internal string Resolve(RecordingSession session)

        {

            return m\_Resolver == null ? string.Empty : m\_Resolver(session);

        }

    }

    /// <summary>

    /// Helper class for default wildcards that can be used when constructing the output file of a recorder.

    /// <see cref="RecorderSettings.outputFile"/>

    /// </summary>

    public static class DefaultWildcard

    {

        public static readonly string Recorder = GeneratePattern("Recorder");

        public static readonly string Time = GeneratePattern("Time");

        public static readonly string Take = GeneratePattern("Take");

        public static readonly string Date = GeneratePattern("Date");

        public static readonly string Project = GeneratePattern("Project");

        public static readonly string Product = GeneratePattern("Product");

        public static readonly string Scene = GeneratePattern("Scene");

        public static readonly string Resolution = GeneratePattern("Resolution");

        public static readonly string Frame = GeneratePattern("Frame");

        public static readonly string Extension = GeneratePattern("Extension");

        public static string GeneratePattern(string tag)

        {

            return "<" + tag + ">";

        }

    }

    [Serializable]

    class FileNameGenerator

    {

        static string s\_ProjectName;

        [SerializeField] OutputPath m\_Path = new OutputPath();

        [SerializeField] string m\_FileName = DefaultWildcard.Recorder;

        readonly List<Wildcard> m\_Wildcards;

        public IEnumerable<Wildcard> wildcards

        {

            get { return m\_Wildcards; }

        }

        internal void FromPath(string str)

        {

            str = SanitizePath(str);

            var i = str.LastIndexOf('/');

            if (i != -1 && i < str.Length - 1)

            {

                m\_FileName = str.Substring(i + 1);

                if (i == 0)

                {

                    m\_Path.root = OutputPath.Root.Absolute;

                    m\_Path.leaf = "/";

                }

                else

                {

                    str = str.Substring(0, i);

                    m\_Path = OutputPath.FromPath(str);

                }

            }

            else

            {

                m\_FileName = str;

                m\_Path.root = OutputPath.Root.Absolute;

                m\_Path.leaf = string.Empty;

            }

        }

        internal string ToPath()

        {

            var path = m\_Path.GetFullPath();

            if (!string.IsNullOrEmpty(path))

                path += "/";

            return SanitizePath(path + SanitizeFilename(m\_FileName));

        }

        internal string fileName {

            get { return m\_FileName; }

            set { m\_FileName = value; }

        }

        internal OutputPath.Root root

        {

            get { return m\_Path.root; }

            set { m\_Path.root = value; }

        }

        internal string leaf

        {

            get { return m\_Path.leaf; }

            set { m\_Path.leaf = value; }

        }

        internal bool forceAssetsFolder

        {

            get { return m\_Path.forceAssetsFolder; }

            set { m\_Path.forceAssetsFolder = value; }

        }

        readonly RecorderSettings m\_RecorderSettings;

        internal FileNameGenerator(RecorderSettings recorderSettings)

        {

            m\_RecorderSettings = recorderSettings;

            m\_Wildcards = new List<Wildcard>

            {

                new Wildcard(DefaultWildcard.Recorder, RecorderResolver),

                new Wildcard(DefaultWildcard.Time, TimeResolver),

                new Wildcard(DefaultWildcard.Take, TakeResolver),

                new Wildcard(DefaultWildcard.Date, DateResolver),

                new Wildcard(DefaultWildcard.Project, ProjectNameResolver),

                new Wildcard(DefaultWildcard.Product, ProductNameResolver),

                new Wildcard(DefaultWildcard.Scene, SceneResolver),

                new Wildcard(DefaultWildcard.Resolution, ResolutionResolver),

                new Wildcard(DefaultWildcard.Frame, FrameResolver),

                new Wildcard(DefaultWildcard.Extension, ExtensionResolver)

            };

        }

        internal void AddWildcard(string tag, Func<RecordingSession, string> resolver)

        {

            m\_Wildcards.Add(new Wildcard(tag, resolver));

        }

        string RecorderResolver(RecordingSession session)

        {

            return m\_RecorderSettings.name;

        }

        static string TimeResolver(RecordingSession session)

        {

            var date = session != null ? session.sessionStartTS : DateTime.Now;

            return string.Format("{0:HH}h{1:mm}m", date, date);

        }

        string TakeResolver(RecordingSession session)

        {

            return m\_RecorderSettings.take.ToString("000");

        }

        static string DateResolver(RecordingSession session)

        {

            var date = session != null ? session.sessionStartTS : DateTime.Now;

            return date.ToString(CultureInfo.InvariantCulture).Replace('/', '-');

        }

        string ExtensionResolver(RecordingSession session)

        {

            return m\_RecorderSettings.extension;

        }

        string ResolutionResolver(RecordingSession session)

        {

            var input = m\_RecorderSettings.inputsSettings.FirstOrDefault() as ImageInputSettings;

            if (input == null)

                return "NA";

            return input.outputWidth + "x" + input.outputHeight;

        }

        static string SceneResolver(RecordingSession session)

        {

            return SceneManager.GetActiveScene().name;

        }

        static string FrameResolver(RecordingSession session)

        {

            var i = session != null ? session.frameIndex : 0;

            return i.ToString("0000");

        }

        static string ProjectNameResolver(RecordingSession session)

        {

            if (string.IsNullOrEmpty(s\_ProjectName))

            {

                var parts = Application.dataPath.Split('/');

                s\_ProjectName = parts[parts.Length - 2];

            }

            return s\_ProjectName;

        }

        static string ProductNameResolver(RecordingSession session)

        {

            return PlayerSettings.productName;

        }

        internal string BuildAbsolutePath(RecordingSession session)

        {

            var fullPath = ApplyWildcards(ToPath(), session) + "." + ExtensionResolver(session);

            string drive = null;

            if (Application.platform == RuntimePlatform.WindowsEditor)

            {

                if (fullPath.Length > 2 && char.IsLetter(fullPath[0]) && fullPath[1] == ':' && fullPath[2] == '/')

                {

                    drive = fullPath.Substring(0, 2);

                    fullPath = fullPath.Substring(3);

                }

            }

            fullPath = string.Join(Path.DirectorySeparatorChar.ToString(), fullPath.Split('/').Select(s =>

                Path.GetInvalidFileNameChars().Aggregate(s, (current, c) => current.Replace(c.ToString(), string.Empty))).ToArray());

            if (!string.IsNullOrEmpty(drive))

                fullPath = drive.ToUpper() + Path.DirectorySeparatorChar + fullPath;

            return fullPath;

        }

        internal void CreateDirectory(RecordingSession session)

        {

            var path = ApplyWildcards(m\_Path.GetFullPath(), session);

            if(!string.IsNullOrEmpty(path) && !Directory.Exists(path))

                Directory.CreateDirectory(path);

        }

        internal static string SanitizeFilename(string filename)

        {

            filename = filename.Replace("\\", "");

            filename = Regex.Replace(filename, "/", "");

            return filename;

        }

        internal static string SanitizePath(string fullPath)

        {

            fullPath = fullPath.Replace("\\", "/");

            fullPath = Regex.Replace(fullPath, "/+", "/");

            return fullPath;

        }

        string ApplyWildcards(string str, RecordingSession session)

        {

            if (string.IsNullOrEmpty(str))

                return string.Empty;

            foreach (var w in wildcards)

                str = str.Replace(w.pattern, w.Resolve(session));

            return str;

        }

    }

}

пїњusing System;

using System.IO;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [CustomPropertyDrawer(typeof(FileNameGenerator))]

    class FileNameGeneratorDrawer : TargetedPropertyDrawer<FileNameGenerator>

    {

        SerializedProperty m\_FileName;

        SerializedProperty m\_Path;

        static bool s\_Dirty = false;

        static readonly GUIStyle s\_PathPreviewStyle = new GUIStyle(GUI.skin.label) { wordWrap = true };

         static readonly GUIStyle s\_OpenPathButtonStyle = new GUIStyle("minibutton") { fixedWidth = 30 };

         static Texture2D s\_OpenPathIcon;

        protected override void Initialize(SerializedProperty property)

        {

             if (s\_OpenPathIcon == null)

             {

                  var iconName = "popout\_icon";

                  if (EditorGUIUtility.isProSkin)

                       iconName = "d\_" + iconName;

                  s\_OpenPathIcon = Resources.Load<Texture2D>(iconName);

             }

            if (target != null)

                return;

            base.Initialize(property);

            m\_FileName = property.FindPropertyRelative("m\_FileName");

            m\_Path = property.FindPropertyRelative("m\_Path");

        }

        public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)

        {

            Initialize(property);

            EditorGUI.BeginProperty(position, label, property);

            position = EditorGUI.PrefixLabel(position, GUIUtility.GetControlID(FocusType.Passive), label);

            const float tagWidth = 77;

            var txtWidth = position.width - tagWidth - 5;

            var txtRect = new Rect(position.x, position.y, txtWidth, position.height);

            var tagRect = new Rect(position.x + txtWidth + 5, position.y, tagWidth, position.height);

            GUI.SetNextControlName("FileNameField");

            m\_FileName.stringValue = GUI.TextField(txtRect, m\_FileName.stringValue);

            var editor = (TextEditor)GUIUtility.GetStateObject(typeof(TextEditor), GUIUtility.keyboardControl);

            if (GUI.GetNameOfFocusedControl().Equals("FileNameField") &&

                Event.current.type == EventType.KeyUp && (Event.current.modifiers == EventModifiers.Control || Event.current.modifiers == EventModifiers.Command))

            {

                if (Event.current.keyCode == KeyCode.C)

                {

                    Event.current.Use();

                    editor.Copy();

                }

                else if (Event.current.keyCode == KeyCode.V)

                {

                    Event.current.Use();

                    editor.Paste();

                    m\_FileName.stringValue = editor.text;

                }

            }

            if (EditorGUI.DropdownButton(tagRect, new GUIContent("+ Wildcards"), FocusType.Passive))

            {

                var menu = new GenericMenu();

                foreach (var w in target.wildcards)

                {

                    var pattern = w.pattern;

                    menu.AddItem(new GUIContent(w.label), false, () =>

                    {

                        m\_FileName.stringValue = InsertTag(pattern, m\_FileName.stringValue, editor);

                        m\_FileName.serializedObject.ApplyModifiedProperties();

                        s\_Dirty = true;

                    });

                }

                menu.DropDown(tagRect);

            }

            if (s\_Dirty)

            {

                s\_Dirty = false;

                GUI.changed = true;

            }

            EditorGUILayout.PropertyField(m\_Path);

            EditorGUILayout.BeginHorizontal();

            EditorGUILayout.PrefixLabel(" ");

             var path = target.BuildAbsolutePath(null);

             var r = GUILayoutUtility.GetRect(new GUIContent(path), s\_PathPreviewStyle, null);

             EditorGUI.SelectableLabel(r, path, s\_PathPreviewStyle);

             if (GUILayout.Button(s\_OpenPathIcon, s\_OpenPathButtonStyle))

                    OpenInFileBrowser.Open(path);

            EditorGUILayout.EndHorizontal();

            EditorGUI.EndProperty();

        }

        static string InsertTag(string pattern, string text, TextEditor editor)

        {

            if (!string.IsNullOrEmpty(editor.text)) // HACK If editor is not focused on

            {

                try

                {

                    editor.ReplaceSelection(pattern);

                    return editor.text;

                }

                catch (Exception)

                {

                    // ignored

                }

            }

            return text + pattern;

        }

    }

using UnityEngine;

namespace UnityEditor.Recorder

{

    abstract class GenericRecorder<T> : Recorder where T : RecorderSettings

    {

        [SerializeField]

        protected T m\_Settings;

        public override RecorderSettings settings

        {

            get { return m\_Settings; }

            set { m\_Settings = (T)value; }

        }

    }

}

пїњusing System;

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

namespace UnityEditor.Recorder

{

    class ImageHeightSelector

    {

        string[] m\_ToName;

        int[] m\_ToEnumValue;

        readonly Dictionary<ImageHeight, int> m\_ToIndex = new Dictionary<ImageHeight, int>();

        int m\_Max;

        readonly bool m\_AllowCustom;

        readonly bool m\_AllowWindow;

        static readonly Dictionary<ImageHeight, string> s\_HeightToName = new Dictionary<ImageHeight, string>

        {

            { ImageHeight.x4320p\_8K, "8K - 4320p" },

            { ImageHeight.x2880p\_5K, "5K - 2880p" },

            { ImageHeight.x2160p\_4K, "4K - 2160p" },

            { ImageHeight.x1440p\_QHD, "QHD - 1440p" },

            { ImageHeight.x1080p\_FHD, "FHD - 1080p" },

            { ImageHeight.x720p\_HD, "HD - 720p" },

            { ImageHeight.x480p, "SD - 480p" },

            { ImageHeight.x240p, "240p" },

            { ImageHeight.Window, "Match Window Size" },

            { ImageHeight.Custom, "Custom" }

        };

        public ImageHeightSelector(int max, bool allowCustom = true, bool allowWindow = true)

        {

            m\_AllowCustom = allowCustom;

            m\_AllowWindow = allowWindow;

            BuildPopup(max);

        }

        void BuildPopup(int max)

        {

            var values = (ImageHeight[]) Enum.GetValues(typeof(ImageHeight));

            var ordered = new List<ImageHeight>();

            if (m\_AllowWindow)

                ordered.Add(ImageHeight.Window);

            ordered.AddRange(values.Where(value => value != ImageHeight.Window && value != ImageHeight.Custom && (int)value <= max));

            if (m\_AllowCustom)

                ordered.Add(ImageHeight.Custom);

            var count = ordered.Count;

            m\_ToName = new string[count];

            m\_ToEnumValue = new int[count];

            for (int i = 0; i < count; ++i)

            {

                var e = ordered[i];

                m\_ToName[i] = s\_HeightToName[e];

                m\_ToEnumValue[i] = (int)e;

                m\_ToIndex[e] = i;

            }

            m\_Max = max;

        }

        public int Popup(GUIContent label, int value, int max)

        {

            if (m\_Max != max)

                BuildPopup(max);

            int index;

            if (!m\_ToIndex.TryGetValue((ImageHeight) value, out index))

                index = 0;

            index = EditorGUILayout.Popup(label, index, m\_ToName);

            return m\_ToEnumValue[index];

        }

    }

}

using System;

using UnityEditor.Recorder.Input;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [Serializable]

    class ImageInputSelector : InputSettingsSelector

    {

        [SerializeField] public GameViewInputSettings gameViewInputSettings = new GameViewInputSettings();

        [SerializeField] public CameraInputSettings cameraInputSettings = new CameraInputSettings();

        [SerializeField] public Camera360InputSettings camera360InputSettings = new Camera360InputSettings();

        [SerializeField] public RenderTextureInputSettings renderTextureInputSettings = new RenderTextureInputSettings();

        [SerializeField] public RenderTextureSamplerSettings renderTextureSamplerSettings = new RenderTextureSamplerSettings();

        public ImageInputSettings imageInputSettings

        {

            get { return (ImageInputSettings)selected; }

            set

            {

                if (value == null)

                    throw new ArgumentNullException("value");

                if (value is CameraInputSettings ||

                    value is GameViewInputSettings ||

                    value is Camera360InputSettings ||

                    value is RenderTextureInputSettings ||

                    value is RenderTextureSamplerSettings)

                {

                    selected = value;

                }

                else

                {

                    throw new ArgumentException("Video input type not supported: '" + value.GetType() + "'");

                }

            }

        }

        public void ForceEvenResolution(bool value)

        {

            gameViewInputSettings.forceEvenSize = value;

            cameraInputSettings.forceEvenSize = value;

        }

    }

    [Serializable]

    class UTJImageInputSelector : InputSettingsSelector

    {

        [SerializeField] public CameraInputSettings cameraInputSettings = new CameraInputSettings();

        [SerializeField] public RenderTextureInputSettings renderTextureInputSettings = new RenderTextureInputSettings();

        [SerializeField] public RenderTextureSamplerSettings renderTextureSamplerSettings = new RenderTextureSamplerSettings();

        public ImageInputSettings imageInputSettings

        {

            get { return (ImageInputSettings)selected; }

            set

            {

                if (value == null)

                    throw new ArgumentNullException("value");

                if (value is CameraInputSettings ||

                    value is RenderTextureInputSettings ||

                    value is RenderTextureSamplerSettings)

                {

                    selected = value;

                }

                else

                {

                    throw new ArgumentException("Video input type not supported: '" + value.GetType() + "'");

                }

            }

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Reflection;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [Serializable]

    abstract class InputSettingsSelector

    {

        [SerializeField] string m\_Selected;

        readonly Dictionary<string, RecorderInputSettings> m\_RecorderInputSettings = new Dictionary<string, RecorderInputSettings>();

        public RecorderInputSettings selected

        {

            get

            {

                if (string.IsNullOrEmpty(m\_Selected) || !m\_RecorderInputSettings.ContainsKey(m\_Selected))

                    m\_Selected = m\_RecorderInputSettings.Keys.First();

                return m\_RecorderInputSettings[m\_Selected];

            }

            protected set

            {

                foreach (var field in InputSettingFields())

                {

                    var input = (RecorderInputSettings)field.GetValue(this);

                    if (input.GetType() == value.GetType())

                    {

                        field.SetValue(this, value);

                        m\_Selected = field.Name;

                        m\_RecorderInputSettings[m\_Selected] = value;

                        break;

                    }

                }

            }

        }

        public IEnumerable<FieldInfo> InputSettingFields()

        {

            return GetInputFields(GetType()).Where(f => typeof(RecorderInputSettings).IsAssignableFrom(f.FieldType));

        }

        public static IEnumerable<FieldInfo> GetInputFields(Type type)

        {

            return type.GetFields(BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic);

        }

        protected InputSettingsSelector()

        {

            foreach (var field in InputSettingFields())

            {

                var input = (RecorderInputSettings)field.GetValue(this);

                m\_RecorderInputSettings.Add(field.Name, input);

            }

        }

    }

}

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Linq;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [CustomPropertyDrawer(typeof(InputSettingsSelector), true)]

    class InputSettingsSelectorDrawer : TargetedPropertyDrawer<InputSettingsSelector>

    {

        bool m\_Initialized;

        GUIContent[] m\_DisplayNames;

        Dictionary<string, int> m\_NameToIndex;

        Dictionary<int, SerializedProperty> m\_IndexToProperty;

        public override float GetPropertyHeight(SerializedProperty property, GUIContent label)

        {

            return 0.0f;

        }

        public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)

        {

            if (!m\_Initialized)

            {

                Initialize(property);

                m\_Initialized = true;

            }

            if (m\_DisplayNames.Length == 0)

            {

                EditorGUILayout.LabelField("No input to select.");

                return;

            }

            var newIndex = 0;

            var selected = property.FindPropertyRelative("m\_Selected");

            if (m\_DisplayNames.Length > 1)

            {

                int index;

                m\_NameToIndex.TryGetValue(selected.stringValue, out index);

                newIndex = EditorGUILayout.Popup(label, index, m\_DisplayNames);

            }

            var sp = m\_IndexToProperty[newIndex];

            selected.stringValue = sp.name;

            ++EditorGUI.indentLevel;

            EditorGUILayout.PropertyField(sp, true);

            --EditorGUI.indentLevel;

        }

        protected override void Initialize(SerializedProperty property)

        {

            base.Initialize(property);

            m\_NameToIndex = new Dictionary<string, int>();

            m\_IndexToProperty = new Dictionary<int, SerializedProperty>();

            var displayNames = new List<GUIContent>();

            int i = 0;

            foreach (var field in target.InputSettingFields())

            {

                var sp = property.FindPropertyRelative(field.Name);

                m\_NameToIndex.Add(sp.name, i);

                m\_IndexToProperty.Add(i, sp);

                displayNames.Add(new GUIContent(GetTypeDisplayName(field.FieldType)));

                ++i;

            }

            m\_DisplayNames = displayNames.ToArray();

        }

        static string GetTypeDisplayName(Type type)

        {

            var displayNameAttribute = type.GetCustomAttributes(typeof(DisplayNameAttribute), true).FirstOrDefault() as DisplayNameAttribute;

            return displayNameAttribute != null

                ? displayNameAttribute.DisplayName

                : ObjectNames.NicifyVariableName(type.Name);

        }

    }

}

namespace UnityEditor.Recorder

{

    enum FrameRateType

    {

        FR\_23, // 24 \* 1000 / 1001

        FR\_24,

        FR\_25,

        FR\_29, // 30 \* 1000 / 1001,

        FR\_30,

        FR\_50,

        FR\_59, // 60 \* 1000 / 1001,

        FR\_60,

        FR\_CUSTOM,

    }

    enum ImageAspect

    {

        x19\_10,

        x16\_9,

        x16\_10,

        x3\_2,

        x4\_3,

        x5\_4,

        x1\_1,

        Custom

    }

    enum ImageHeight

    {

        Window = 0,

        x4320p\_8K = 4320,

        x2880p\_5K = 2880,

        x2160p\_4K = 2160,

        x1440p\_QHD = 1440,

        x1080p\_FHD = 1080,

        x720p\_HD = 720,

        x480p = 480,

        x240p = 240,

        Custom = int.MaxValue

    }

}

пїњ

пїњusing System;

using System.Text.RegularExpressions;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [Serializable]

    class OutputPath

    {

        public enum Root

        {

            Project,

            AssetsFolder,

            StreamingAssets,

            PersistentData,

            TemporaryCache,

            Absolute

        }

        [SerializeField] Root m\_Root;

        [SerializeField] string m\_Leaf;

        [SerializeField] bool m\_ForceAssetFolder;

        public Root root

        {

            get { return m\_Root; }

            set { m\_Root = value; }

        }

        public string leaf

        {

            get { return m\_Leaf; }

            set { m\_Leaf = value; }

        }

        public bool forceAssetsFolder

        {

            get { return m\_ForceAssetFolder;}

            set

            {

                m\_ForceAssetFolder = value;

                if (m\_ForceAssetFolder)

                    m\_Root = Root.AssetsFolder;

            }

        }

        public static OutputPath FromPath(string path)

        {

            var result = new OutputPath();

            if (path.Contains(Application.streamingAssetsPath))

            {

                result.m\_Root = Root.StreamingAssets;

                result.m\_Leaf = path.Replace(Application.streamingAssetsPath, string.Empty);

            }

            else if (path.Contains(Application.dataPath))

            {

                result.m\_Root = Root.AssetsFolder;

                result.m\_Leaf = path.Replace(Application.dataPath, string.Empty);

            }

            else if (path.Contains(Application.persistentDataPath))

            {

                result.m\_Root = Root.PersistentData;

                result.m\_Leaf = path.Replace(Application.persistentDataPath, string.Empty);

            }

            else if (path.Contains(Application.temporaryCachePath))

            {

                result.m\_Root = Root.TemporaryCache;

                result.m\_Leaf = path.Replace(Application.temporaryCachePath, string.Empty);

            }

            else if (path.Contains(ProjectPath()))

            {

                result.m\_Root = Root.Project;

                result.m\_Leaf = path.Replace(ProjectPath(), string.Empty);

            }

            else

            {

                result.m\_Root = Root.Absolute;

                result.m\_Leaf = path;

            }

            return result;

        }

        public static string GetFullPath(Root root, string leaf)

        {

            var ret = string.Empty;

            switch (root)

            {

                case Root.PersistentData:

                    ret = Application.persistentDataPath;

                    break;

                case Root.StreamingAssets:

                    ret = Application.streamingAssetsPath;

                    break;

                case Root.TemporaryCache:

                    ret = Application.temporaryCachePath;

                    break;

                case Root.AssetsFolder:

                    ret = Application.dataPath;

                    break;

                case Root.Project:

                    ret = ProjectPath();

                    break;

            }

            if (root != Root.Absolute && !leaf.StartsWith("/"))

            {

                ret += "/";

            }

            ret += leaf;

            return ret;

        }

        public string GetFullPath()

        {

            return GetFullPath(m\_Root, m\_Leaf);

        }

        static string ProjectPath()

        {

            return Regex.Replace(Application.dataPath, "/Assets$", string.Empty);

        }

    }

}

пїњusing System.IO;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [CustomPropertyDrawer(typeof(OutputPath))]

    class OutputPathDrawer : TargetedPropertyDrawer<OutputPath>

    {

        SerializedProperty m\_RootProperty;

        SerializedProperty m\_LeafProperty;

        SerializedProperty m\_ForceAssetFolder;

        protected override void Initialize(SerializedProperty property)

        {

            base.Initialize(property);

            if (m\_RootProperty == null)

                m\_RootProperty = property.FindPropertyRelative("m\_Root");

            if (m\_LeafProperty == null)

                m\_LeafProperty = property.FindPropertyRelative("m\_Leaf");

            if (m\_ForceAssetFolder == null)

                m\_ForceAssetFolder = property.FindPropertyRelative("m\_ForceAssetFolder");

        }

        public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)

        {

            Initialize(property);

            EditorGUI.BeginProperty(position, label, property);

            position = EditorGUI.PrefixLabel(position, GUIUtility.GetControlID(FocusType.Passive), label);

            var indent = EditorGUI.indentLevel;

            EditorGUI.indentLevel = 0;

            const float rootWidth = 110.0f;

            const float btnWidth = 30.0f;

            var leafWidth = target.forceAssetsFolder ? position.width - rootWidth : position.width - rootWidth - btnWidth - 10;

            var rootRect = new Rect(position.x, position.y, rootWidth, position.height);

            var leafRect = new Rect(position.x + rootWidth + 5, position.y, leafWidth, position.height);

            var btnRect = new Rect(position.x + rootWidth  + leafWidth + 10, position.y, btnWidth, position.height);

            if (target.forceAssetsFolder)

            {

                var root = (OutputPath.Root) m\_RootProperty.intValue;

                GUI.Label(rootRect, root + " " + Path.DirectorySeparatorChar);

            }

            else

            {

                EditorGUI.PropertyField(rootRect, m\_RootProperty, GUIContent.none);

            }

            EditorGUI.PropertyField(leafRect, m\_LeafProperty, GUIContent.none);

            var fullPath = OutputPath.GetFullPath((OutputPath.Root)m\_RootProperty.intValue, m\_LeafProperty.stringValue);

            if (!target.forceAssetsFolder)

            {

                if (GUI.Button(btnRect, new GUIContent("...", fullPath)))

                {

                    var newPath = EditorUtility.OpenFolderPanel("Select output location", fullPath, "");

                    if (!string.IsNullOrEmpty(newPath))

                    {

                        var newValue = OutputPath.FromPath(newPath);

                        m\_RootProperty.intValue = (int) newValue.root;

                        m\_LeafProperty.stringValue = newValue.leaf;

                    }

                }

            }

            EditorGUI.indentLevel = indent;

            EditorGUI.EndProperty();

        }

    }

}

пїњusing System;

using UnityEditor.Recorder.Input;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [Serializable]

    class OutputResolution

    {

        [SerializeField] int m\_CustomWidth = 1024;

        [SerializeField] int m\_CustomHeight = 1024;

        [SerializeField] internal ImageHeight imageHeight = ImageHeight.x720p\_HD;

        [SerializeField] internal ImageHeight maxSupportedHeight = ImageHeight.x4320p\_8K;

        [SerializeField] AspectRatio m\_AspectRatio = new AspectRatio();

        public int GetWidth()

        {

            if (imageHeight == ImageHeight.Custom)

                return m\_CustomWidth;

            if (imageHeight == ImageHeight.Window)

            {

                int w, h;

                GameViewSize.GetGameRenderSize(out w, out h);

                return w;

            }

            var aspect = m\_AspectRatio.GetAspect();

            return (int) (aspect \* (int)imageHeight);

        }

        public int GetHeight()

        {

            if (imageHeight == ImageHeight.Custom)

                return m\_CustomHeight;

            if (imageHeight == ImageHeight.Window)

            {

                int w, h;

                GameViewSize.GetGameRenderSize(out w, out h);

                return h;

            }

            return (int)imageHeight;

        }

        public void SetWidth(int w)

        {

            imageHeight = ImageHeight.Custom;

            m\_CustomWidth = w;

        }

        public void SetHeight(int h)

        {

            imageHeight = ImageHeight.Custom;

            m\_CustomHeight = h;

        }

    }

}

пїњusing UnityEngine;

namespace UnityEditor.Recorder

{

    [CustomPropertyDrawer(typeof(OutputResolution))]

    class OutputResolutionPropertyDrawer : PropertyDrawer

    {

        SerializedProperty m\_CustomWidth;

        SerializedProperty m\_CustomHeight;

        SerializedProperty m\_ImageHeight;

        SerializedProperty m\_AspectRatio;

        SerializedProperty m\_MaxSupportedHeight;

        ImageHeightSelector m\_HeightSelector;

        bool m\_Initialized;

        static class Styles

        {

            internal static readonly GUIContent ImageAspectLabel = new GUIContent("Aspect Ratio");

            static readonly GUIContent s\_CustomWidthLabel = new GUIContent("W");

            static readonly GUIContent s\_CustomHeightLabel = new GUIContent("H");

            internal static readonly GUIContent[] CustomDimensionsLabels = { s\_CustomWidthLabel, s\_CustomHeightLabel };

        }

        public override float GetPropertyHeight(SerializedProperty property, GUIContent label)

        {

            return 0.0f;

        }

        void Initialize(SerializedProperty property)

        {

            if (m\_Initialized )

                return;

            m\_Initialized = true;

            m\_CustomWidth = property.FindPropertyRelative("m\_CustomWidth");

            m\_CustomHeight = property.FindPropertyRelative("m\_CustomHeight");

            m\_ImageHeight = property.FindPropertyRelative("imageHeight");

            m\_AspectRatio = property.FindPropertyRelative("m\_AspectRatio");

            m\_MaxSupportedHeight = property.FindPropertyRelative("maxSupportedHeight");

            m\_HeightSelector = new ImageHeightSelector(m\_MaxSupportedHeight.intValue);

        }

        public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)

        {

            Initialize(property);

            m\_ImageHeight.intValue = m\_HeightSelector.Popup(label, m\_ImageHeight.intValue, m\_MaxSupportedHeight.intValue);

            var selected = (ImageHeight) m\_ImageHeight.intValue;

            if (selected == ImageHeight.Custom)

            {

                var outputDimensions = new int[2];

                outputDimensions[0] = m\_CustomWidth.intValue;

                outputDimensions[1] = m\_CustomHeight.intValue;

                if (UIElementHelper.MultiIntField(GUIContent.none, Styles.CustomDimensionsLabels, outputDimensions))

                {

                    m\_CustomWidth.intValue = outputDimensions[0];

                    m\_CustomHeight.intValue = outputDimensions[1];

                }

            }

            if (selected != ImageHeight.Custom && selected != ImageHeight.Window)

            {

                EditorGUILayout.PropertyField(m\_AspectRatio, Styles.ImageAspectLabel);

            }

        }

    }

}

using System;

using UnityEditor.Experimental.UIElements;

using UnityEngine;

using UnityEngine.Experimental.UIElements;

namespace UnityEditor.Recorder

{

    class PanelSplitter : VisualElement

    {

        readonly VisualElement m\_AffectedElement;

        bool m\_Grabbed;

        Vector2 m\_GrabbedMousePosition;

        float m\_ElementOriginalWidth;

        const float k\_SplitterWidth = 5.0f;

        [Serializable]

        class Width

        {

            public float value;

        }

        Width m\_Width;

        void SetWidth(float value)

        {

            if (m\_Width == null)

                return;

            m\_Width.value = value;

            m\_AffectedElement.style.width = value;

            SavePersistentData();

        }

        public PanelSplitter(VisualElement affectedElement)

        {

            m\_AffectedElement = affectedElement;

            style.cursor = UIElementsEditorUtility.CreateDefaultCursorStyle(MouseCursor.ResizeHorizontal);

            style.width = k\_SplitterWidth;

            style.minWidth = k\_SplitterWidth;

            style.maxWidth = k\_SplitterWidth;

            RegisterCallback<MouseDownEvent>(OnMouseDown, Capture.Capture);

            RegisterCallback<MouseMoveEvent>(OnMouseMove, Capture.Capture);

            RegisterCallback<MouseUpEvent>(OnMouseUp, Capture.Capture);

        }

        void OnMouseDown(MouseDownEvent evt)

        {

            if (evt.button != (int) MouseButton.LeftMouse)

                return;

            if (m\_Grabbed)

                return;

            this.TakeMouseCapture();

            m\_Grabbed = true;

            m\_GrabbedMousePosition = evt.mousePosition;

            m\_ElementOriginalWidth = m\_AffectedElement.style.width;

            evt.StopImmediatePropagation();

        }

        void OnMouseMove(MouseMoveEvent evt)

        {

            if (!m\_Grabbed)

                return;

            var delta = evt.mousePosition.x - m\_GrabbedMousePosition.x;

            var newWidth = Mathf.Max(m\_ElementOriginalWidth + delta, m\_AffectedElement.style.minWidth);

            if (m\_AffectedElement.style.maxWidth > 0.0f)

                newWidth = Mathf.Min(newWidth, m\_AffectedElement.style.maxWidth);

            SetWidth(newWidth);

        }

        void OnMouseUp(MouseUpEvent evt)

        {

            if (evt.button != (int) MouseButton.LeftMouse)

                return;

            if (!m\_Grabbed)

                return;

            m\_Grabbed = false;

            this.ReleaseMouseCapture();

            evt.StopImmediatePropagation();

        }

        public override void OnPersistentDataReady()

        {

            base.OnPersistentDataReady();

            var key = GetFullHierarchicalPersistenceKey();

            m\_Width = GetOrCreatePersistentData<Width>(m\_Width, key);

            if (m\_Width.value > 0.0f)

                m\_AffectedElement.style.width = m\_Width.value;

        }

    }

}

using System;

using System.Collections.Generic;

using UnityEngine;

namespace UnityEditor.Recorder

{

    enum ERecordingSessionStage

    {

        BeginRecording,

        NewFrameStarting,

        NewFrameReady,

        FrameDone,

        EndRecording,

        SessionCreated

    }

    abstract class Recorder : ScriptableObject

    {

        static int sm\_CaptureFrameRateCount;

        bool m\_ModifiedCaptureFR;

        public int recordedFramesCount { get; set; }

        protected List<RecorderInput> m\_Inputs;

        public virtual void Awake()

        {

            sm\_CaptureFrameRateCount = 0;

        }

        public virtual void Reset()

        {

            recordedFramesCount = 0;

            recording = false;

        }

        protected virtual void OnDestroy()

        {

            if (m\_ModifiedCaptureFR )

            {

                sm\_CaptureFrameRateCount--;

                if (sm\_CaptureFrameRateCount == 0)

                {

                    Time.captureFramerate = 0;

                    if (Options.verboseMode)

                        Debug.Log("Recorder resetting 'CaptureFrameRate' to zero");

                }

            }

        }

        public abstract RecorderSettings settings { get; set; }

        public virtual void SessionCreated(RecordingSession session)

        {

            if (Options.verboseMode)

                Debug.Log(string.Format("Recorder {0} session created", GetType().Name));

            settings.SelfAdjustSettings(); // ignore return value.

            var fixedRate = settings.frameRatePlayback == FrameRatePlayback.Constant ? (int)settings.frameRate : 0;

            if (fixedRate > 0)

            {

                if (Time.captureFramerate != 0 && fixedRate != Time.captureFramerate )

                    Debug.LogError(string.Format("Recorder {0} is set to record at a fixed rate and another component has already set a conflicting value for [Time.captureFramerate], new value being applied : {1}!", GetType().Name, fixedRate));

                else if( Time.captureFramerate == 0 && Options.verboseMode )

                    Debug.Log("Frame recorder set fixed frame rate to " + fixedRate);

                Time.captureFramerate = fixedRate;

                sm\_CaptureFrameRateCount++;

                m\_ModifiedCaptureFR = true;

            }

            m\_Inputs = new List<RecorderInput>();

            foreach (var inputSettings in settings.inputsSettings)

            {

                var input = (RecorderInput)Activator.CreateInstance(inputSettings.inputType);

                input.settings = inputSettings;

                m\_Inputs.Add(input);

                SignalInputsOfStage(ERecordingSessionStage.SessionCreated, session);

            }

        }

        public virtual bool BeginRecording(RecordingSession session)

        {

            if (recording)

                throw new Exception("Already recording!");

            if (Options.verboseMode)

                Debug.Log(string.Format("Recorder {0} starting to record", GetType().Name));

            return recording = true;

        }

        public virtual void EndRecording(RecordingSession session)

        {

            if (!recording)

                return;

            recording = false;

            if (m\_ModifiedCaptureFR )

            {

                m\_ModifiedCaptureFR = false;

                sm\_CaptureFrameRateCount--;

                if (sm\_CaptureFrameRateCount == 0)

                {

                    Time.captureFramerate = 0;

                    if (Options.verboseMode)

                        Debug.Log("Recorder resetting 'CaptureFrameRate' to zero");

                }

            }

            foreach (var input in m\_Inputs)

            {

                if (input != null)

                    input.Dispose();

            }

            if(Options.verboseMode)

                Debug.Log(string.Format("{0} recording stopped, total frame count: {1}", GetType().Name, recordedFramesCount));

            ++settings.take;

        }

        public abstract void RecordFrame(RecordingSession ctx);

        public virtual void PrepareNewFrame(RecordingSession ctx)

        {

        }

        public virtual bool SkipFrame(RecordingSession ctx)

        {

            return !recording

                || (ctx.frameIndex % settings.captureEveryNthFrame) != 0

                || ( settings.recordMode == RecordMode.TimeInterval && ctx.currentFrameStartTS < settings.startTime )

                || ( settings.recordMode == RecordMode.FrameInterval && ctx.frameIndex < settings.startFrame )

                || ( settings.recordMode == RecordMode.SingleFrame && ctx.frameIndex < settings.startFrame );

        }

        public bool recording { get; protected set; }

        public void SignalInputsOfStage(ERecordingSessionStage stage, RecordingSession session)

        {

            if (m\_Inputs == null)

                return;

            switch (stage)

            {

                case ERecordingSessionStage.SessionCreated:

                    foreach( var input in m\_Inputs )

                        input.SessionCreated(session);

                    break;

                case ERecordingSessionStage.BeginRecording:

                    foreach( var input in m\_Inputs )

                        input.BeginRecording(session);

                    break;

                case ERecordingSessionStage.NewFrameStarting:

                    foreach( var input in m\_Inputs )

                        input.NewFrameStarting(session);

                    break;

                case ERecordingSessionStage.NewFrameReady:

                    foreach( var input in m\_Inputs )

                        input.NewFrameReady(session);

                    break;

                case ERecordingSessionStage.FrameDone:

                    foreach( var input in m\_Inputs )

                        input.FrameDone(session);

                    break;

                case ERecordingSessionStage.EndRecording:

                    foreach( var input in m\_Inputs )

                        input.EndRecording(session);

                    break;

                default:

                    throw new ArgumentOutOfRangeException("stage", stage, null);

            }

        }

    }

}

пїњ

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using UnityEditorInternal;

using UnityEngine;

using UnityObject = UnityEngine.Object;

namespace UnityEditor.Recorder

{

    public class RecorderControllerSettings : ScriptableObject

    {

        [SerializeField] RecordMode m\_RecordMode = RecordMode.Manual;

        [SerializeField] FrameRatePlayback m\_FrameRatePlayback = FrameRatePlayback.Constant;

        [SerializeField] FrameRateType m\_FrameRateType = FrameRateType.FR\_30;

        [SerializeField] [Range(1.0f, 120.0f)] float m\_CustomFrameRateValue = 30.0f;

        [SerializeField] int m\_StartFrame;

        [SerializeField] int m\_EndFrame;

        [SerializeField] float m\_StartTime;

        [SerializeField] float m\_EndTime;

        [SerializeField] bool m\_CapFrameRate = true;

        static readonly Dictionary<FrameRateType, float> s\_FPSToValue = new Dictionary<FrameRateType, float>()

        {

            { FrameRateType.FR\_23, 24 \* 1000 / 1001f },

            { FrameRateType.FR\_24, 24 },

            { FrameRateType.FR\_25, 25 },

            { FrameRateType.FR\_29, 30 \* 1000 / 1001f },

            { FrameRateType.FR\_30, 30 },

            { FrameRateType.FR\_50, 50 },

            { FrameRateType.FR\_59, 60 \* 1000 / 1001f },

            { FrameRateType.FR\_60, 60 }

        };

        public FrameRatePlayback frameRatePlayback

        {

            get { return m\_FrameRatePlayback; }

            set { m\_FrameRatePlayback = value; }

        }

        public float frameRate

        {

            get

            {

                return m\_FrameRateType == FrameRateType.FR\_CUSTOM ? m\_CustomFrameRateValue : s\_FPSToValue[m\_FrameRateType];

            }

            set

            {

                m\_FrameRateType = FrameRateType.FR\_CUSTOM;

                m\_CustomFrameRateValue = value;

            }

        }

        public void SetRecordModeToManual()

        {

            m\_RecordMode = RecordMode.Manual;

        }

        public void SetRecordModeToSingleFrame(int frameNumber)

        {

            m\_RecordMode = RecordMode.SingleFrame;

            m\_StartFrame = m\_EndFrame = frameNumber;

        }

        public void SetRecordModeToFrameInterval(int startFrame, int endFrame)

        {

            m\_RecordMode = RecordMode.FrameInterval;

            m\_StartFrame = startFrame;

            m\_EndFrame = endFrame;

        }

        public void SetRecordModeToTimeInterval(float startTime, float endTime)

        {

            m\_RecordMode = RecordMode.TimeInterval;

            m\_StartTime = startTime;

            m\_EndTime = endTime;

        }

        public bool capFrameRate

        {

            get { return m\_CapFrameRate; }

            set { m\_CapFrameRate = value; }

        }

        [SerializeField] List<RecorderSettings> m\_RecorderSettings = new List<RecorderSettings>();

        string m\_Path;

        public static RecorderControllerSettings LoadOrCreate(string path)

        {

            RecorderControllerSettings prefs;

            try

            {

                var objs = InternalEditorUtility.LoadSerializedFileAndForget(path);

                prefs = objs.FirstOrDefault(p => p is RecorderControllerSettings) as RecorderControllerSettings;

            }

            catch (Exception e)

            {

                Debug.LogError("Unhandled exception while loading Recorder preferences: " + e);

                prefs = null;

            }

            if (prefs == null)

            {

                prefs = CreateInstance<RecorderControllerSettings>();

                prefs.hideFlags = HideFlags.DontSave | HideFlags.HideInHierarchy;

                prefs.name = "Global Settings";

                prefs.Save();

            }

            prefs.m\_Path = path;

            return prefs;

        }

        internal void ReleaseRecorderSettings()

        {

            foreach (var recorder in m\_RecorderSettings)

            {

                DestroyImmediate(recorder);

            }

            ClearRecorderSettings();

        }

        internal void ClearRecorderSettings()

        {

            m\_RecorderSettings.Clear();

        }

        public IEnumerable<RecorderSettings> recorderSettings

        {

            get { return m\_RecorderSettings; }

        }

        public void AddRecorderSettings(RecorderSettings recorder)

        {

            if (!m\_RecorderSettings.Contains(recorder))

            {

                AddRecorderInternal(recorder);

                Save();

            }

        }

        public void RemoveRecorder(RecorderSettings recorder)

        {

            if (m\_RecorderSettings.Contains(recorder))

            {

                m\_RecorderSettings.Remove(recorder);

                Save();

            }

        }

        public void Save()

        {

            if (string.IsNullOrEmpty(m\_Path))

                return;

            try

            {

                var directory = Path.GetDirectoryName(m\_Path);

                if (!string.IsNullOrEmpty(directory) && !Directory.Exists(directory))

                    Directory.CreateDirectory(directory);

                var recordersCopy = recorderSettings.ToArray();

                var objs = new UnityObject[recordersCopy.Length + 1];

                objs[0] = this;

                for (int i = 0; i < recordersCopy.Length; ++i)

                    objs[i + 1] = recordersCopy[i];

                InternalEditorUtility.SaveToSerializedFileAndForget(objs, m\_Path, true);

            }

            catch (Exception e)

            {

                Debug.LogError("Unhandled exception while saving Recorder settings: " + e);

            }

        }

        internal void ApplyGlobalSetting(RecorderSettings recorder)

        {

            recorder.recordMode = m\_RecordMode;

            recorder.frameRatePlayback = m\_FrameRatePlayback;

            recorder.frameRate = frameRate;

            recorder.startFrame = m\_StartFrame;

            recorder.endFrame = m\_EndFrame;

            recorder.startTime = m\_StartTime;

            recorder.endTime = m\_EndTime;

            recorder.capFrameRate = m\_CapFrameRate;

            recorder.hideFlags = HideFlags.DontSave | HideFlags.HideInHierarchy;

            recorder.SelfAdjustSettings();

        }

        internal void ApplyGlobalSettingToAllRecorders()

        {

            foreach (var recorder in recorderSettings)

                ApplyGlobalSetting(recorder);

        }

        void AddRecorderInternal(RecorderSettings recorder)

        {

            ApplyGlobalSetting(recorder);

            m\_RecorderSettings.Add(recorder);

        }

    }

}

using System.Collections.Generic;

namespace UnityEditor.Recorder

{

    [CustomEditor(typeof(RecorderControllerSettingsPreset))]

    class RecorderControllerSettingsPresetEditor : Editor

    {

        Editor m\_Editor;

        class PresetEditorState

        {

            public bool expanded;

            public Editor presetEditor;

        }

        readonly List<PresetEditorState> m\_RecordersEditors = new List<PresetEditorState>();

        void OnEnable()

        {

            if (target == null)

                return;

            var preset = (RecorderControllerSettingsPreset) target;

            m\_Editor = CreateEditor(preset.model);

            m\_RecordersEditors.Clear();

            var recorderPresets = preset.recorderPresets;

            foreach (var p in recorderPresets)

            {

                var state = new PresetEditorState

                {

                    presetEditor = CreateEditor(p),

                    expanded = false

                };

                m\_RecordersEditors.Add(state);

            }

        }

        public override void OnInspectorGUI()

        {

            if (target == null)

                return;

            m\_Editor.OnInspectorGUI();

            EditorGUILayout.Separator();

            foreach (var state in m\_RecordersEditors)

            {

                if (FoldoutPresetEditorStateHeader(state))

                {

                    EditorGUILayout.Separator();

                    state.presetEditor.OnInspectorGUI();

                }

            }

        }

        static bool FoldoutPresetEditorStateHeader(PresetEditorState state)

        {

            var r = EditorGUILayout.GetControlRect();

            state.expanded = EditorGUI.Foldout(r, state.expanded, state.presetEditor.target.name);

            return state.expanded;

        }

        void OnDestroy()

        {

            if (m\_Editor != null)

            {

                DestroyImmediate(m\_Editor);

                m\_Editor = null;

            }

            foreach (var state in m\_RecordersEditors)

                DestroyImmediate(state.presetEditor);

            m\_RecordersEditors.Clear();

        }

    }

}

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

namespace UnityEditor.Recorder

{

    abstract class RecorderEditor : Editor

    {

        SerializedProperty m\_CaptureEveryNthFrame;

        SerializedProperty m\_FileNameGenerator;

        SerializedProperty m\_Take;

        static Texture2D s\_SeparatorTexture;

        static readonly Color s\_SeparatorColor = new Color(1.0f, 1.0f, 1.0f, 0.1f);

        static class Styles

        {

            internal static readonly GUIContent FileNameLabel = new GUIContent("File Name");

            internal static readonly GUIContent CaptureLabel = new GUIContent("Capture");

            internal static readonly GUIContent TakeNumberLabel = new GUIContent("Take Number");

            internal static readonly GUIContent RenderStepFrameLabel = new GUIContent("Render Step Frame");

        }

        protected virtual void OnEnable()

        {

            if (target != null)

            {

                var pf = new PropertyFinder<RecorderSettings>(serializedObject);

                m\_CaptureEveryNthFrame = pf.Find(w => w.captureEveryNthFrame);

                m\_FileNameGenerator = pf.Find(w => w.fileNameGenerator);

                m\_Take = pf.Find(w => w.take);

                s\_SeparatorTexture = Resources.Load<Texture2D>("vertical\_gradient");

            }

        }

        static void DrawSeparator()

        {

            EditorGUILayout.Separator();

            var r = EditorGUILayout.GetControlRect();

            r.xMin -= 10.0f;

            r.xMax += 10.0f;

            r.yMin += 5.0f;

            r.height = 10;

            var orgColor = GUI.color;

            GUI.color = s\_SeparatorColor;

            GUI.DrawTexture(r, s\_SeparatorTexture);

            GUI.color = orgColor;

            EditorGUILayout.Separator();

        }

        public override void OnInspectorGUI()

        {

            if (target == null)

                return;

            EditorGUI.BeginChangeCheck();

            serializedObject.Update();

            FileTypeAndFormatGUI();

            DrawSeparator();

            NameAndPathGUI();

            ImageRenderOptionsGUI();

            EditorGUILayout.Separator();

            ExtraOptionsGUI();

            EditorGUILayout.Separator();

            OnEncodingGui();

            serializedObject.ApplyModifiedProperties();

            EditorGUI.EndChangeCheck();

            if (GUI.changed)

                ((RecorderSettings) target).SelfAdjustSettings();

            OnValidateSettingsGUI();

        }

        protected virtual void OnValidateSettingsGUI()

        {

            var errors = new List<string>();

            if (!((RecorderSettings) target).ValidityCheck(errors))

            {

                foreach (var error in errors)

                    EditorGUILayout.HelpBox(error, MessageType.Warning);

            }

        }

        protected virtual void NameAndPathGUI()

        {

            EditorGUILayout.PropertyField(m\_FileNameGenerator, Styles.FileNameLabel);

            EditorGUILayout.Space();

            EditorGUI.BeginChangeCheck();

            EditorGUILayout.PropertyField(m\_Take, Styles.TakeNumberLabel);

            if (EditorGUI.EndChangeCheck())

                m\_Take.intValue = Mathf.Max(0, m\_Take.intValue);

        }

        protected virtual void ImageRenderOptionsGUI()

        {

            var recorder = (RecorderSettings) target;

            foreach (var inputsSetting in recorder.inputsSettings)

            {

                var p = GetInputSerializedProperty(serializedObject, inputsSetting);

                EditorGUILayout.Separator();

                EditorGUILayout.PropertyField(p, Styles.CaptureLabel);

            }

        }

        static SerializedProperty GetInputSerializedProperty(SerializedObject owner, object fieldValue)

        {

            var targetObject = (object)owner.targetObject;

            var type = targetObject.GetType();

            foreach (var info in InputSettingsSelector.GetInputFields(type))

            {

                if (info.GetValue(targetObject) == fieldValue)

                {

                    return owner.FindProperty(info.Name);

                }

                if (typeof(InputSettingsSelector).IsAssignableFrom(info.FieldType))

                {

                    var selector = info.GetValue(targetObject);

                    var fields = InputSettingsSelector.GetInputFields(selector.GetType());

                    var selectorInput = fields.FirstOrDefault(i => i.GetValue(selector) == fieldValue);

                    if (selectorInput != null)

                    {

                        return owner.FindProperty(info.Name);

                    }

                }

            }

            return null;

        }

        protected virtual void ExtraOptionsGUI()

        {

            if (((RecorderSettings)target).frameRatePlayback == FrameRatePlayback.Variable)

                EditorGUILayout.PropertyField(m\_CaptureEveryNthFrame, Styles.RenderStepFrameLabel);

        }

        protected virtual void FileTypeAndFormatGUI()

        {

        }

        protected virtual void OnEncodingGui()

        {

        }

    }

}

using System;

namespace UnityEditor.Recorder

{

    class RecorderInput : IDisposable

    {

        public RecorderInputSettings settings { get; set; }

        ~RecorderInput()

        {

            Dispose(false);

        }

        public void Dispose()

        {

            Dispose(true);

        }

        protected virtual void Dispose(bool disposing)

        {

            GC.SuppressFinalize(this);

        }

        public virtual void SessionCreated(RecordingSession session) {}

        public virtual void BeginRecording(RecordingSession session) {}

        public virtual void NewFrameStarting(RecordingSession session) {}

        public virtual void NewFrameReady(RecordingSession session) {}

        public virtual void FrameDone(RecordingSession session) {}

        public virtual void EndRecording(RecordingSession session) {}

    }

}

using System;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Experimental.UIElements;

using UnityEngine.Experimental.UIElements.StyleEnums;

using UnityObject = UnityEngine.Object;

namespace UnityEditor.Recorder

{

    class RecorderItem : VisualElement

    {

        public RecorderSettings settings { get; private set; }

        public Editor editor { get; private set; }

        readonly EditableLabel m\_EditableLabel;

        readonly Toggle m\_Toggle;

        readonly Texture2D m\_RecorderIcon;

        Texture2D m\_Icon;

        public event Action<bool> OnEnableStateChanged;

        static readonly Dictionary<string, Texture2D> s\_IconCache = new Dictionary<string, Texture2D>();

        bool m\_Selected;

        bool m\_Disabled;

        public void SetItemSelected(bool value)

        {

            m\_Selected = value;

            if (value)

                AddToClassList("selected");

            else

                RemoveFromClassList("selected");

        }

        public void SetItemEnabled(RecorderControllerSettings prefs, bool value)

        {

            m\_Disabled = !value;

            settings.enabled = value;

            prefs.Save();

            m\_EditableLabel.SetLabelEnabled(value);

            if (m\_Toggle != null)

                UIElementHelper.SetToggleValue(m\_Toggle, value);

            if (value)

                RemoveFromClassList("disabled");

            else

                AddToClassList("disabled");

            if (OnEnableStateChanged != null)

                OnEnableStateChanged.Invoke(value);

        }

        public enum State

        {

            None,

            Normal,

            HasWarnings,

            HasErrors

        }

        State m\_State = State.None;

        public void UpdateState(bool checkForWarnings = true)

        {

            try

            {

                if (settings == null || settings.HasErrors())

                {

                    state = State.HasErrors;

                    return;

                }

                if (checkForWarnings && settings.HasWarnings())

                {

                    state = State.HasWarnings;

                    return;

                }

            }

            catch (Exception e)

            {

                Debug.LogWarning("Exception when getting recorder state: " + e);

            }

            state = State.Normal;

        }

        public State state

        {

            get { return m\_State; }

            set

            {

                if (value == State.None)

                    return;

                if (m\_State == value)

                    return;

                switch (m\_State)

                {

                    case State.HasWarnings:

                        RemoveFromClassList("hasWarnings");

                        break;

                    case State.HasErrors:

                        RemoveFromClassList("hasErrors");

                        break;

                }

                switch (value)

                {

                    case State.HasWarnings:

                        AddToClassList("hasWarnings");

                        m\_Icon = StatusBarHelper.warningIcon;

                        break;

                    case State.HasErrors:

                        AddToClassList("hasErrors");

                        m\_Icon = StatusBarHelper.errorIcon;

                        break;

                    case State.Normal:

                        m\_Icon = m\_RecorderIcon;

                        break;

                }

                m\_State = value;

            }

        }

        static Texture2D LoadIcon(string iconName)

        {

            if (string.IsNullOrEmpty(iconName))

                return null;

            Texture2D icon;

            if (s\_IconCache.TryGetValue(iconName, out icon))

                return icon;

            if (EditorGUIUtility.isProSkin)

                icon = Resources.Load<Texture2D>("d\_" + iconName);

            if (icon == null)

                icon = Resources.Load<Texture2D>(iconName);

            s\_IconCache[iconName] = icon;

            return icon;

        }

        public RecorderItem(RecorderControllerSettings prefs, RecorderSettings recorderSettings, string iconName)

        {

            settings = recorderSettings;

            if (settings != null)

                editor = Editor.CreateEditor(settings);

            UIElementHelper.SetFlex(this, 1.0f);

            style.flexDirection = FlexDirection.Row;

            m\_Toggle = new Toggle(null);

            m\_Toggle.OnToggle(() =>

            {

                SetItemEnabled(prefs, UIElementHelper.GetToggleValue(m\_Toggle));

            });

            Add(m\_Toggle);

            m\_RecorderIcon = LoadIcon(iconName);

            if (m\_RecorderIcon == null)

                m\_RecorderIcon = LoadIcon("customrecorder\_16");

            UpdateState(false);

            var iconContainer = new IMGUIContainer(() => // UIElement Image doesn't support tint yet. Use IMGUI instead.

            {

                var r = EditorGUILayout.GetControlRect();

                r.width = r.height = Mathf.Max(r.width, r.height);

                var c = GUI.color;

                var newColor = Color.white;

                if (m\_Disabled)

                {

                    newColor.a = 0.5f;

                }

                else

                {

                    if (!m\_Selected)

                        newColor.a = 0.8f;

                }

                GUI.color = newColor;

                GUI.DrawTexture(r, m\_Icon);

                GUI.color = c;

            });

            iconContainer.AddToClassList("RecorderItemIcon");

            iconContainer.SetEnabled(false);

            Add(iconContainer);

            m\_EditableLabel = new EditableLabel { text = settings.name };

            m\_EditableLabel.OnValueChanged(newValue =>

            {

                settings.name = newValue;

                prefs.Save();

            });

            Add(m\_EditableLabel);

            var recorderEnabled = settings.enabled;

            UIElementHelper.SetToggleValue(m\_Toggle, recorderEnabled);

            SetItemEnabled(prefs, recorderEnabled);

        }

        public void StartRenaming()

        {

            m\_EditableLabel.StartEditing();

        }

    }

}

using System;

namespace UnityEditor.Recorder

{

    [AttributeUsage(AttributeTargets.Class, Inherited = false)]

    class RecorderSettingsAttribute : Attribute

    {

        public readonly Type recorderType;

        public readonly string displayName;

        public readonly string iconName;

        public RecorderSettingsAttribute(Type recorderType, string displayName)

        {

            this.recorderType = recorderType;

            this.displayName = displayName;

        }

        public RecorderSettingsAttribute(Type recorderType, string displayName, string iconName)

        {

            this.iconName = iconName;

            this.recorderType = recorderType;

            this.displayName = displayName;

        }

    }

}

using UnityEngine;

namespace UnityEditor.Recorder

{

    [CustomEditor(typeof(RecorderControllerSettings))]

    class RecorderSettingsPrefsEditor : Editor

    {

        SerializedProperty m\_RecordModeProperty;

        SerializedProperty m\_PlaybackProperty;

        SerializedProperty m\_FrameRateTypeProperty;

        SerializedProperty m\_CustomFrameRateValueProperty;

        SerializedProperty m\_StartFrameProperty;

        SerializedProperty m\_EndFrameProperty;

        SerializedProperty m\_StartTimeProperty;

        SerializedProperty m\_EndTimeProperty;

        SerializedProperty m\_CapFrameRateProperty;

        GenericMenu m\_FrameRateMenu;

        static class Styles

        {

            internal static readonly GUIContent RecordModeLabel = new GUIContent("Record Mode");

            internal static readonly GUIContent SingleFrameLabel = new GUIContent("Frame");

            internal static readonly GUIContent StartLabel = new GUIContent("Start");

            internal static readonly GUIContent EndLabel = new GUIContent("End");

            internal static readonly GUIContent FrameRateTitle   = new GUIContent("Frame Rate");

            internal static readonly GUIContent PlaybackLabel    = new GUIContent("Playback");

            internal static readonly GUIContent TargetFPSLabel   = new GUIContent("Target");

            internal static readonly GUIContent MaxFPSLabel      = new GUIContent("Max");

            internal static readonly GUIContent CapFPSLabel      = new GUIContent("Cap");

            internal static readonly GUIContent ValueLabel       = new GUIContent("Value");

        }

        void OnEnable()

        {

            if (target == null)

                return;

            m\_RecordModeProperty = serializedObject.FindProperty("m\_RecordMode");

            m\_PlaybackProperty = serializedObject.FindProperty("m\_FrameRatePlayback");

            m\_FrameRateTypeProperty  = serializedObject.FindProperty("m\_FrameRateType");

            m\_CustomFrameRateValueProperty = serializedObject.FindProperty("m\_CustomFrameRateValue");

            m\_StartFrameProperty = serializedObject.FindProperty("m\_StartFrame");

            m\_EndFrameProperty = serializedObject.FindProperty("m\_EndFrame");

            m\_StartTimeProperty = serializedObject.FindProperty("m\_StartTime");

            m\_EndTimeProperty = serializedObject.FindProperty("m\_EndTime");

            m\_CapFrameRateProperty = serializedObject.FindProperty("m\_CapFrameRate");

        }

        public override void OnInspectorGUI()

        {

            RecordModeGUI();

            EditorGUILayout.Separator();

            FrameRateGUI();

        }

        internal bool RecordModeGUI()

        {

            serializedObject.Update();

            EditorGUILayout.PropertyField(m\_RecordModeProperty, Styles.RecordModeLabel);

            ++EditorGUI.indentLevel;

            switch ((RecordMode)m\_RecordModeProperty.enumValueIndex)

            {

                case RecordMode.Manual:

                {

                    // Nothing

                    break;

                }

                case RecordMode.SingleFrame:

                {

                    var value = EditorGUILayout.IntField(Styles.SingleFrameLabel, m\_StartFrameProperty.intValue);

                    m\_StartFrameProperty.intValue = Mathf.Max(value, 0);

                    break;

                }

                case RecordMode.FrameInterval:

                {

                    var outputDimensions = new int[2];

                    outputDimensions[0] = m\_StartFrameProperty.intValue;

                    outputDimensions[1] = m\_EndFrameProperty.intValue;

                    if (UIElementHelper.MultiIntField(GUIContent.none, new [] { Styles.StartLabel, Styles.EndLabel },

                        outputDimensions))

                    {

                        m\_StartFrameProperty.intValue = Mathf.Max(outputDimensions[0], 0);

                        m\_EndFrameProperty.intValue = Mathf.Max(outputDimensions[1], m\_StartFrameProperty.intValue);

                    }

                    break;

                }

                case RecordMode.TimeInterval:

                {

                    var outputDimensions = new float[2];

                    outputDimensions[0] = m\_StartTimeProperty.floatValue;

                    outputDimensions[1] = m\_EndTimeProperty.floatValue;

                    if (UIElementHelper.MultiFloatField(GUIContent.none, new [] { Styles.StartLabel, Styles.EndLabel },

                        outputDimensions))

                    {

                        m\_StartTimeProperty.floatValue = Mathf.Max(outputDimensions[0], 0);

                        m\_EndTimeProperty.floatValue = Mathf.Max(outputDimensions[1], m\_StartTimeProperty.floatValue);

                    }

                    break;

                }

            }

            --EditorGUI.indentLevel;

            serializedObject.ApplyModifiedProperties();

            return GUI.changed;

        }

        internal bool FrameRateGUI()

        {

            serializedObject.Update();

            EditorGUILayout.LabelField(Styles.FrameRateTitle);

            ++EditorGUI.indentLevel;

            EditorGUILayout.PropertyField(m\_PlaybackProperty, Styles.PlaybackLabel);

            var variableFPS = m\_PlaybackProperty.enumValueIndex == (int) FrameRatePlayback.Variable;

            EditorGUILayout.PropertyField(m\_FrameRateTypeProperty, variableFPS ? Styles.MaxFPSLabel : Styles.TargetFPSLabel);

            if (m\_FrameRateTypeProperty.enumValueIndex == (int) FrameRateType.FR\_CUSTOM)

            {

                ++EditorGUI.indentLevel;

                EditorGUILayout.PropertyField(m\_CustomFrameRateValueProperty, Styles.ValueLabel);

                --EditorGUI.indentLevel;

            }

            if (!variableFPS)

            {

                EditorGUILayout.PropertyField(m\_CapFrameRateProperty, Styles.CapFPSLabel);

            }

            --EditorGUI.indentLevel;

            serializedObject.ApplyModifiedProperties();

            return GUI.changed;

        }

    }

}

using System;

using System.Collections.Generic;

using UnityEngine;

using System.Linq;

using UnityEditor.Recorder.FrameCapturer;

namespace UnityEditor.Recorder

{

    class RecorderInfo

    {

        public Type recorderType;

        public Type settingsType;

        public string displayName;

        public string iconName;

    }

    static class RecordersInventory

    {

        static Dictionary<Type, RecorderInfo> s\_Recorders;

        static HashSet<RecorderInfo> s\_BuiltInRecorderInfos;

        static HashSet<RecorderInfo> s\_LegacyRecorderInfos;

        static IEnumerable<KeyValuePair<Type, object[]>> FindRecorders()

        {

            var attribType = typeof(RecorderSettingsAttribute);

            foreach (var a in AppDomain.CurrentDomain.GetAssemblies())

            {

                Type[] types;

                try

                {

                    types = a.GetTypes();

                }

                catch (Exception)

                {

                    Debug.LogError( "Failed reflecting assembly: " + a.FullName );

                    continue;

                }

                foreach (var t in types)

                {

                    var attributes = t.GetCustomAttributes(attribType, false);

                    if (attributes.Length != 0)

                        yield return new KeyValuePair<Type, object[]>(t, attributes);

                }

            }

        }

        static void Init()

        {

            if (s\_Recorders == null)

            {

                s\_Recorders = new Dictionary<Type, RecorderInfo>();

                foreach (var recorder in FindRecorders())

                {

                    var settingsType = recorder.Key;

                    var settingsAttribs = recorder.Value;

                    if (settingsType == null || string.IsNullOrEmpty(settingsType.FullName))

                        continue;

                    if (settingsAttribs.Length == 1)

                    {

                        var settingsAttrib = (RecorderSettingsAttribute) settingsAttribs[0];

                        var info = new RecorderInfo

                        {

                            settingsType = settingsType,

                            recorderType = settingsAttrib.recorderType,

                            displayName = settingsAttrib.displayName,

                            iconName = settingsAttrib.iconName

                        };

                        s\_Recorders.Add(settingsType, info);

                    }

                }

            }

            if (s\_Recorders != null)

            {

                if (s\_BuiltInRecorderInfos == null)

                {

                    s\_BuiltInRecorderInfos = new HashSet<RecorderInfo>

                    {

                        s\_Recorders[typeof(AnimationRecorderSettings)],

                        s\_Recorders[typeof(MovieRecorderSettings)],

                        s\_Recorders[typeof(ImageRecorderSettings)],

                        s\_Recorders[typeof(GIFRecorderSettings)]

                    };

                }

                if (s\_LegacyRecorderInfos == null)

                {

                    s\_LegacyRecorderInfos = new HashSet<RecorderInfo>

                    {

                        s\_Recorders[typeof(MP4RecorderSettings)],

                        s\_Recorders[typeof(EXRRecorderSettings)],

                        s\_Recorders[typeof(PNGRecorderSettings)],

                        s\_Recorders[typeof(WEBMRecorderSettings)]

                    };

                }

            }

        }

        internal static RecorderInfo GetRecorderInfo(Type settingsType)

        {

            Init();

            if (settingsType == null || string.IsNullOrEmpty(settingsType.FullName))

                return null;

            return s\_Recorders.ContainsKey(settingsType) ? s\_Recorders[settingsType] : null;

        }

        internal static IEnumerable<RecorderInfo> builtInRecorderInfos

        {

            get

            {

                Init();

                return s\_BuiltInRecorderInfos;

            }

        }

        internal static IEnumerable<RecorderInfo> legacyRecorderInfos

        {

            get

            {

                Init();

                return s\_LegacyRecorderInfos;

            }

        }

        internal static IEnumerable<RecorderInfo> customRecorderInfos

        {

            get

            {

                Init();

                return s\_Recorders.Values.Where(r => !s\_BuiltInRecorderInfos.Contains(r) && !s\_LegacyRecorderInfos.Contains(r));

            }

        }

        internal static Recorder CreateDefaultRecorder(RecorderSettings recorderSettings)

        {

            Init();

            var factory = GetRecorderInfo(recorderSettings.GetType());

            if (factory != null)

            {

                var recorder = (Recorder)ScriptableObject.CreateInstance(factory.recorderType);

                recorder.Reset();

                recorder.settings = recorderSettings;

                return recorder;

            }

            throw new ArgumentException("No factory was registered for " + recorderSettings.GetType().Name);

        }

        internal static RecorderSettings CreateDefaultRecorderSettings(Type settingsType)

        {

            Init();

            var recorderinfo = GetRecorderInfo(settingsType);

            if (recorderinfo != null)

            {

                var settings = (RecorderSettings)ObjectFactory.CreateInstance(recorderinfo.settingsType);

                settings.name = settingsType.Name;

                return settings;

            }

            throw new ArgumentException("No factory was registered for " + settingsType.Name);

        }

    }

}

using System;

using UnityEngine;

namespace UnityEditor.Recorder

{

    class RecordingSession : IDisposable

    {

        internal Recorder recorder;

        internal GameObject recorderGameObject;

        internal RecorderComponent recorderComponent;

        int m\_FrameIndex = 0;

        int m\_InitialFrame = 0;

        int m\_FirstRecordedFrameCount = -1;

        float m\_FPSTimeStart;

        float m\_FPSNextTimeStart;

        int m\_FPSNextFrameCount;

        internal double currentFrameStartTS { get; private set; }

        internal double recordingStartTS { get; private set; }

        internal DateTime sessionStartTS { get; private set; }

        internal RecorderSettings settings

        {

            get { return recorder.settings; }

        }

        internal bool isRecording

        {

            get { return recorder.recording; }

        }

        internal int frameIndex

        {

            get { return m\_FrameIndex; }

        }

        internal int RecordedFrameSpan

        {

            get { return m\_FirstRecordedFrameCount == -1 ? 0 : Time.renderedFrameCount - m\_FirstRecordedFrameCount; }

        }

        internal float recorderTime

        {

            get { return (float)(currentFrameStartTS - settings.startTime); }

        }

        static void AllowInBackgroundMode()

        {

            if (!Application.runInBackground)

            {

                Application.runInBackground = true;

                if (Options.verboseMode)

                    Debug.Log("Recording sessions is enabling Application.runInBackground!");

            }

        }

        internal bool SessionCreated()

        {

            try

            {

                AllowInBackgroundMode();

                recordingStartTS = (Time.time / Time.timeScale);

                sessionStartTS = DateTime.Now;

                recorder.SessionCreated(this);

                return true;

            }

            catch (Exception ex)

            {

                Debug.LogException(ex);

                return false;

            }

        }

        internal bool BeginRecording()

        {

            try

            {

                if (!settings.isPlatformSupported)

                {

                    Debug.LogError(string.Format("Recorder {0} does not support current platform", recorder.GetType().Name));

                    return false;

                }

                AllowInBackgroundMode();

                recordingStartTS = (Time.time / Time.timeScale);

                recorder.SignalInputsOfStage(ERecordingSessionStage.BeginRecording, this);

                if (!recorder.BeginRecording(this))

                    return false;

                m\_InitialFrame = Time.renderedFrameCount;

                m\_FPSTimeStart = Time.unscaledTime;

                return true;

            }

            catch (Exception ex)

            {

                Debug.LogException(ex);

                return false;

            }

        }

        internal void EndRecording()

        {

            if (!isRecording)

                return;

            try

            {

                recorder.SignalInputsOfStage(ERecordingSessionStage.EndRecording, this);

                recorder.EndRecording(this);

            }

            catch (Exception ex)

            {

                Debug.LogException(ex);

            }

        }

        internal void RecordFrame()

        {

            try

            {

                recorder.SignalInputsOfStage(ERecordingSessionStage.NewFrameReady, this);

                if (!recorder.SkipFrame(this))

                {

                    recorder.RecordFrame(this);

                    recorder.recordedFramesCount++;

                    if (recorder.recordedFramesCount == 1)

                        m\_FirstRecordedFrameCount = Time.renderedFrameCount;

                }

                recorder.SignalInputsOfStage(ERecordingSessionStage.FrameDone, this);

            }

            catch (Exception ex)

            {

                Debug.LogException(ex);

            }

            // Note: This is not great when multiple recorders are simultaneously active...

            if (settings.frameRatePlayback == FrameRatePlayback.Variable ||

                settings.frameRatePlayback == FrameRatePlayback.Constant && recorder.settings.capFrameRate)

            {

                var frameCount = Time.renderedFrameCount - m\_InitialFrame;

                var frameLen = 1.0f / recorder.settings.frameRate;

                var elapsed = Time.unscaledTime - m\_FPSTimeStart;

                var target = frameLen \* (frameCount + 1);

                var sleep = (int)((target - elapsed) \* 1000);

                if (sleep > 2)

                {

                    if (Options.verboseMode)

                        Debug.Log(string.Format("Recording session info => dT: {0:F1}s, Target dT: {1:F1}s, Retarding: {2}ms, fps: {3:F1}", elapsed, target, sleep, frameCount / elapsed));

                    System.Threading.Thread.Sleep(Math.Min(sleep, 1000));

                }

                else if (sleep < -frameLen)

                    m\_InitialFrame--;

                else if (Options.verboseMode)

                    Debug.Log(string.Format("Recording session info => fps: {0:F1}", frameCount / elapsed));

                // reset every 30 frames

                if (frameCount % 50 == 49)

                {

                    m\_FPSNextTimeStart = Time.unscaledTime;

                    m\_FPSNextFrameCount = Time.renderedFrameCount;

                }

                if (frameCount % 100 == 99)

                {

                    m\_FPSTimeStart = m\_FPSNextTimeStart;

                    m\_InitialFrame = m\_FPSNextFrameCount;

                }

            }

            m\_FrameIndex++;

        }

        internal void PrepareNewFrame()

        {

            try

            {

                AllowInBackgroundMode();

                currentFrameStartTS = (Time.time / Time.timeScale) - recordingStartTS;

                recorder.SignalInputsOfStage(ERecordingSessionStage.NewFrameStarting, this);

                recorder.PrepareNewFrame(this);

            }

            catch (Exception ex)

            {

                Debug.LogException(ex);

            }

        }

        public void Dispose()

        {

            if (recorder != null)

            {

                EndRecording();

                UnityHelpers.Destroy(recorder);

            }

        }

    }

}

namespace UnityEditor.Recorder

{

    [CustomPropertyDrawer(typeof(RecordMode))]

    class RecordModePropertyDrawer : EnumProperyDrawer<RecordMode>

    {

        protected override string ToLabel(RecordMode value)

        {

            switch (value)

            {

                case RecordMode.Manual:

                    return "Manual";

                case RecordMode.SingleFrame:

                    return "Single Frame";

                case RecordMode.FrameInterval:

                    return "Frame Interval";

                case RecordMode.TimeInterval:

                    return "Time Interval (sec)";

                default:

                    return "unknown";

            }

        }

    }

}

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

using UnityEngine.Recorder;

using UnityEngine.SceneManagement;

namespace UnityEditor.Recorder

{

    static class BindingManager

    {

        const string k\_HostGoName = "Unity-RecorderBindings";

        public static Object Get(string id)

        {

            var rcs = FindRecorderBindings();

            var rc = rcs.FirstOrDefault(r => r.HasBindingValue(id));

            return rc != null ? rc.GetBindingValue(id) : null;

        }

        public static void Duplicate(string id, string newId)

        {

            var rcs = FindRecorderBindings();

            foreach (var rc in rcs)

            {

                rc.DuplicateBinding(id, newId);

            }

        }

        public static void Set(string id, Object obj)

        {

            var rbs = FindRecorderBindings();

            if (obj == null)

            {

                // Remove

                foreach (var rb in rbs)

                {

                    RemoveBinding(id, rb);

                }

            }

            else

            {

                var scene = GetObjectScene(obj);

                var rb = rbs.FirstOrDefault(r => r.gameObject.scene == scene);

                if (rb == null)

                {

                    // Add

                    var gameObject = UnityHelpers.CreateRecorderGameObject(k\_HostGoName);

                    rb = gameObject.AddComponent<RecorderBindings>();

                    SceneManager.MoveGameObjectToScene(rb.gameObject, scene);

                }

                // Replace

                rb.SetBindingValue(id, obj);

                foreach (var r in rbs)

                {

                    if (r == rb)

                        continue;

                    RemoveBinding(id, r);

                }

            }

        }

        static void RemoveBinding(string id, RecorderBindings rb)

        {

            rb.RemoveBinding(id);

            if (rb.IsEmpty())

                Object.DestroyImmediate(rb.gameObject);

        }

        public static RecorderBindings[] FindRecorderBindings()

        {

            return Object.FindObjectsOfType<RecorderBindings>();

        }

        static Scene GetObjectScene(Object obj)

        {

            var gameObject = obj as GameObject;

            if (gameObject != null)

                return gameObject.scene;

            var component = obj as Component;

            if (component != null)

                return component.gameObject.scene;

            return SceneManager.GetActiveScene();

        }

    }

    class SceneHook

    {

        const string k\_HostGoName = "Unity-RecorderSessions";

        static GameObject s\_SessionHooksRoot;

        readonly string m\_SessionId;

        GameObject m\_SessionHook;

        public SceneHook(string sessionId)

        {

            m\_SessionId = sessionId;

        }

        static GameObject GetSessionHooksRoot(bool createIfNecessary = true)

        {

            if (s\_SessionHooksRoot == null)

            {

                s\_SessionHooksRoot = GameObject.Find(k\_HostGoName);

                if (s\_SessionHooksRoot == null)

                {

                    if (!createIfNecessary)

                        return null;

                    s\_SessionHooksRoot = UnityHelpers.CreateRecorderGameObject(k\_HostGoName);

                }

            }

            return s\_SessionHooksRoot;

        }

        GameObject GetSessionHook()

        {

            if (m\_SessionHook != null)

                return m\_SessionHook;

            var host = GetSessionHooksRoot();

            if (host == null)

                return null;

            m\_SessionHook = GameObject.Find(m\_SessionId);

            if (m\_SessionHook == null)

            {

                m\_SessionHook = new GameObject(m\_SessionId);

                m\_SessionHook.transform.parent = host.transform;

            }

            return m\_SessionHook;

        }

        public IEnumerable<RecordingSession> GetRecordingSessions()

        {

            var sessionHook = GetSessionHook();

            if (sessionHook != null)

            {

                var components = sessionHook.GetComponents<RecorderComponent>();

                foreach (var component in components)

                {

                    yield return component.session;

                }

            }

        }

        public static void PrepareSessionRoot()

        {

            var host = GetSessionHooksRoot();

            if (host != null)

            {

                host.hideFlags = HideFlags.None;

                Object.DontDestroyOnLoad(host);

            }

        }

        public RecordingSession CreateRecorderSessionWithRecorderComponent(RecorderSettings settings)

        {

            var component = GetRecorderComponent(settings);

            var session = new RecordingSession

            {

                recorder = RecordersInventory.CreateDefaultRecorder(settings),

                recorderGameObject = component.gameObject,

                recorderComponent = component

            };

            component.session = session;

            return session;

        }

        public RecordingSession CreateRecorderSession(RecorderSettings settings)

        {

            var sceneHook = GetSessionHook();

            if (sceneHook == null)

                return null;

            var session = new RecordingSession

            {

                recorder = RecordersInventory.CreateDefaultRecorder(settings),

                recorderGameObject = sceneHook

            };

            return session;

        }

        RecorderComponent GetRecorderComponent(RecorderSettings settings)

        {

            var sceneHook = GetSessionHook();

            if (sceneHook == null)

                return null;

            var component = sceneHook.GetComponentsInChildren<RecorderComponent>().FirstOrDefault(r => r.session.settings == settings);

            if (component == null)

                component = sceneHook.AddComponent<RecorderComponent>();

            return component;

        }

    }

}

пїњusing System.Reflection;

namespace UnityEditor.Recorder

{

    class TargetedPropertyDrawer<T> : PropertyDrawer where T : class

    {

        protected T target;

        protected virtual void Initialize(SerializedProperty prop)

        {

            if (target == null)

            {

                var path = prop.propertyPath.Split('.');

                object obj = prop.serializedObject.targetObject;

                foreach (var pathNode in path)

                    obj = GetSerializedField(obj, pathNode).GetValue(obj);

                target = obj as T;

            }

        }

        static FieldInfo GetSerializedField(object target, string pathNode)

        {

            return target.GetType().GetField(pathNode, BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public);

        }

    }

}

пїњusing System;

using UnityEngine;

namespace UnityEditor.Recorder

{

    class TextureFlipper : IDisposable

    {

        RenderTexture m\_WorkTexture;

        internal void Flip(RenderTexture target)

        {

            if (m\_WorkTexture == null || m\_WorkTexture.width != target.width || m\_WorkTexture.height != target.height)

            {

                UnityHelpers.Destroy(m\_WorkTexture);

                m\_WorkTexture = new RenderTexture(target);

            }

            var sRGBWrite = GL.sRGBWrite;

            GL.sRGBWrite = PlayerSettings.colorSpace == ColorSpace.Linear;

            Graphics.Blit(target, m\_WorkTexture, new Vector2(1.0f, -1.0f), new Vector2(0.0f, 1.0f));

            Graphics.Blit(m\_WorkTexture, target);

            GL.sRGBWrite = sRGBWrite;

        }

        public void Dispose()

        {

            UnityHelpers.Destroy(m\_WorkTexture);

            m\_WorkTexture = null;

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using UnityEngine;

using UnityEngine.Experimental.UIElements;

using UnityEngine.Experimental.UIElements.StyleEnums;

namespace UnityEditor.Recorder

{

    class VisualListItem<T> : VisualElement where T : VisualElement

    {

        public event Action OnSelectionChanged;

        public event Action OnContextMenu;

        public event Action<T> OnItemContextMenu;

        public event Action<T> OnItemRename;

        [Serializable]

        class Selection

        {

            public int index = -1;

        }

        Selection m\_Selection;

        public int selectedIndex

        {

            get { return m\_Selection != null ? m\_Selection.index : 0; }

            set

            {

                if (m\_Selection == null)

                    return;

                m\_Selection.index = value;

                if (OnSelectionChanged != null)

                    OnSelectionChanged.Invoke();

                SavePersistentData();

            }

        }

        readonly ScrollView m\_ScrollView;

        readonly List<T> m\_ItemsCache = new List<T>();

        protected VisualListItem()

        {

            m\_ScrollView = new ScrollView

            {

                style =

                {

                    flexDirection = FlexDirection.Column

                }

            };

            UIElementHelper.SetFlex(m\_ScrollView, 1.0f);

            m\_ScrollView.contentContainer.style.positionLeft = 0;

            m\_ScrollView.contentContainer.style.positionRight = 0;

            Add(m\_ScrollView);

            RegisterCallback<MouseUpEvent>(OnMouseUp);

        }

        public void Reload(IEnumerable<T> itemList)

        {

            m\_ScrollView.Clear();

            m\_ItemsCache.Clear();

            selectedIndex = -1;

            foreach (var item in itemList)

                Add(item);

            selection = m\_ItemsCache.FirstOrDefault();

        }

        public List<T> items

        {

            get { return m\_ItemsCache; }

        }

        public T selection

        {

            get

            {

                if(selectedIndex < 0 || selectedIndex > m\_ItemsCache.Count - 1)

                    return null;

                return m\_ItemsCache[selectedIndex];

            }

            set

            {

                if (selection == value)

                    return;

                selectedIndex = m\_ItemsCache.IndexOf(value);

            }

        }

        public void Add(T item)

        {

            item.RegisterCallback<MouseDownEvent>(OnItemMouseDown);

            item.RegisterCallback<MouseUpEvent>(OnItemMouseUp);

            m\_ScrollView.Add(item);

            m\_ItemsCache.Add(item);

        }

        public void Remove(T item)

        {

            var selected = selection == item;

            m\_ScrollView.Remove(item);

            m\_ItemsCache.Remove(item);

            if (selected)

                selectedIndex = Math.Min(selectedIndex, items.Count - 1);

        }

        void OnMouseUp(MouseUpEvent evt)

        {

            if (evt.clickCount != 1)

                return;

            if (evt.button == (int) MouseButton.RightMouse)

            {

                if (OnContextMenu != null)

                    OnContextMenu.Invoke();

            }

            evt.StopImmediatePropagation();

        }

        public bool HasFocus()

        {

            return focusController.focusedElement == this;

        }

        void OnItemMouseDown(MouseDownEvent evt)

        {

            if (evt.clickCount != 1)

                return;

            if (evt.button != (int) MouseButton.LeftMouse && evt.button != (int) MouseButton.RightMouse)

                return;

            var item = (T) evt.currentTarget;

            if (evt.modifiers == EventModifiers.None)

            {

                var alreadySelected = selection == item;

                if (evt.button == (int) MouseButton.LeftMouse && alreadySelected)

                {

                    if (HasFocus() && OnItemRename != null)

                        OnItemRename.Invoke(item);

                }

                else

                {

                    selection = item;

                }

            }

            evt.StopImmediatePropagation();

        }

        void OnItemMouseUp(MouseUpEvent evt)

        {

            if (evt.clickCount != 1)

                return;

            if (evt.modifiers != EventModifiers.None || evt.button != (int) MouseButton.RightMouse)

                return;

            if (OnItemContextMenu != null)

            {

                var item = (T) evt.currentTarget;

                OnItemContextMenu.Invoke(item);

            }

            evt.StopImmediatePropagation();

        }

        public override void OnPersistentDataReady()

        {

            base.OnPersistentDataReady();

            var key = GetFullHierarchicalPersistenceKey();

            m\_Selection = GetOrCreatePersistentData<Selection>(m\_Selection, key);

            if (OnSelectionChanged != null)

                OnSelectionChanged.Invoke();

        }

    }

}

using System.Collections;

using UnityEngine;

namespace UnityEditor.Recorder

{

    [ExecuteInEditMode]

    class RecorderComponent : MonoBehaviour

    {

        public RecordingSession session { get; set; }

        public void Update()

        {

            if (session != null && session.isRecording)

                session.PrepareNewFrame();

        }

        IEnumerator RecordFrame()

        {

            yield return new WaitForEndOfFrame();

            if (session != null && session.isRecording)

            {

                session.RecordFrame();

                switch (session.recorder.settings.recordMode)

                {

                    case RecordMode.Manual:

                        break;

                    case RecordMode.SingleFrame:

                    {

                        if (session.recorder.recordedFramesCount == 1)

                            enabled = false;

                        break;

                    }

                    case RecordMode.FrameInterval:

                    {

                        if (session.frameIndex > session.settings.endFrame)

                            enabled = false;

                        break;

                    }

                    case RecordMode.TimeInterval:

                    {

                        if (session.settings.frameRatePlayback == FrameRatePlayback.Variable)

                        {

                            if (session.currentFrameStartTS >= session.settings.endTime)

                                enabled = false;

                        }

                        else

                        {

                            var expectedFrames = (session.settings.endTime - session.settings.startTime) \* session.settings.frameRate;

                            if (session.RecordedFrameSpan >= expectedFrames)

                                enabled = false;

                        }

                        break;

                    }

                }

            }

        }

        public void LateUpdate()

        {

            if (session != null && session.isRecording)

                StartCoroutine(RecordFrame());

        }

        public void OnDisable()

        {

            if (session != null)

            {

                session.Dispose();

                session = null;

            }

        }

        public void OnDestroy()

        {

            if (session != null)

                session.Dispose();

        }

    }

}

пїњusing System;

using System.Collections.Generic;

namespace UnityEditor.Recorder

{

    static class EnumHelper

    {

        internal static int GetEnumValueFromMaskedIndex<TEnum>(int index, int mask)

        {

            if (!typeof(TEnum).IsEnum) throw new ArgumentException("Arg not an enum");

            var values = Enum.GetValues(typeof(TEnum));

            for( int i = 0, j = -1; i < values.Length; i++ )

            {

                if (((int)values.GetValue(i) & mask) != 0)

                    j++;

                if (j == index)

                    return (int)values.GetValue(i);

            }

            throw new ArgumentException("invalid masked index");

        }

        internal static int GetMaskedIndexFromEnumValue<TEnum>(int value, int mask)

        {

            if (!typeof(TEnum).IsEnum) throw new ArgumentException("Arg not an enum");

            var values = Enum.GetValues(typeof(TEnum));

            for( int i = 0, j = -1; i < values.Length; i++ )

            {

                var v = (int)values.GetValue(i);

                if ((v & mask) != 0)

                {

                    j++;

                    if (v == value)

                        return j;

                }

            }

            return 0;

        }

        internal static string[] MaskOutEnumNames<TEnum>(int mask )

        {

            if (!typeof(TEnum).IsEnum) throw new ArgumentException("Arg not an enum");

            var names = Enum.GetNames(typeof(TEnum));

            var values = Enum.GetValues(typeof(TEnum));

            var result = new List<String>();

            for( int i = 0; i < values.Length; i++ )

            {

                if( ((int)values.GetValue(i) & mask ) != 0 )

                    result.Add( (string)names.GetValue(i) );

            }

            return result.ToArray();

        }

    }

}

using System;

using UnityEditor.Presets;

using UnityEngine;

namespace UnityEditor.Recorder

{

    static class PresetHelper

    {

        static Texture2D s\_PresetIcon;

        static GUIStyle s\_PresetButtonStyle;

        internal static Texture2D presetIcon

        {

            get

            {

                if (s\_PresetIcon == null)

                    s\_PresetIcon = (Texture2D) EditorGUIUtility.Load(EditorGUIUtility.isProSkin ? "d\_Preset.Context" : "Preset.Context");

                return s\_PresetIcon;

            }

        }

        internal static GUIStyle presetButtonStyle

        {

            get

            {

                return s\_PresetButtonStyle ?? (s\_PresetButtonStyle = new GUIStyle("iconButton") { fixedWidth = 19.0f });

            }

        }

        internal class PresetReceiver : PresetSelectorReceiver

        {

            RecorderSettings m\_Target;

            Preset m\_InitialValue;

            Action m\_OnSelectionChanged;

            Action m\_OnSelectionClosed;

            internal void Init(RecorderSettings target, Action onSelectionChanged = null, Action onSelectionClosed = null)

            {

                m\_OnSelectionChanged = onSelectionChanged;

                m\_OnSelectionClosed = onSelectionClosed;

                m\_Target = target;

                m\_InitialValue = new Preset(target);

            }

            public override void OnSelectionChanged(Preset selection)

            {

                if (selection != null)

                {

                    Undo.RecordObject(m\_Target, "Apply Preset " + selection.name);

                    selection.ApplyTo(m\_Target);

                }

                else

                {

                    Undo.RecordObject(m\_Target, "Cancel Preset");

                    m\_InitialValue.ApplyTo(m\_Target);

                }

                if (m\_OnSelectionChanged != null)

                    m\_OnSelectionChanged.Invoke();

            }

            public override void OnSelectionClosed(Preset selection)

            {

                OnSelectionChanged(selection);

                m\_Target.OnAfterDuplicate();

                if (m\_OnSelectionClosed != null)

                    m\_OnSelectionClosed.Invoke();

                DestroyImmediate(this);

            }

        }

    }

}

using System;

using System.Linq.Expressions;

namespace UnityEditor.Recorder

{

    static class SerializableObjHelper

    {

        public static SerializedProperty FindPropertyRelative(this SerializedProperty obj, Expression<Func<object>> exp)

        {

            var body = exp.Body as MemberExpression;

            if (body == null)

            {

                var ubody = (UnaryExpression)exp.Body;

                body = ubody.Operand as MemberExpression;

            }

            var name = body.Member.Name;

            return obj.FindPropertyRelative(name);

        }

    }

    class PropertyFinder<TType> where TType : class

    {

        SerializedObject m\_Obj;

        public PropertyFinder(SerializedObject obj)

        {

            m\_Obj = obj;

        }

        public delegate TResult FuncX<TResult>(TType x);

        public SerializedProperty Find( Expression<FuncX<object>> exp)

        {

            var body = exp.Body as MemberExpression;

            if (body == null)

            {

                var ubody = (UnaryExpression)exp.Body;

                body = ubody.Operand as MemberExpression;

            }

            var name = body.Member.Name;

            return m\_Obj.FindProperty(name);

        }

    }

}

пїњusing UnityEngine;

namespace UnityEditor.Recorder

{

    static class StatusBarHelper

    {

        static Texture2D s\_ErrorIcon;

        static Texture2D s\_WarningIcon;

        static Texture2D s\_InfoIcon;

        static GUIStyle s\_ErrorStyle;

        static GUIStyle s\_WarningStyle;

        static GUIStyle s\_InfoStyle;

        public static Texture2D errorIcon

        {

            get

            {

                if (s\_ErrorIcon == null)

                    s\_ErrorIcon = EditorGUIUtility.Load("Icons/console.erroricon.sml.png") as Texture2D;

                return s\_ErrorIcon;

            }

        }

        public static Texture2D warningIcon

        {

            get

            {

                if (s\_WarningIcon == null)

                    s\_WarningIcon = EditorGUIUtility.Load("Icons/console.warnicon.sml.png") as Texture2D;

                return s\_WarningIcon;

            }

        }

        public static Texture2D infoIcon

        {

            get

            {

                if (s\_InfoIcon == null)

                    s\_InfoIcon = EditorGUIUtility.Load("Icons/console.infoicon.sml.png") as Texture2D;

                return s\_InfoIcon;

            }

        }

        public static GUIStyle errorStyle

        {

            get

            {

                return s\_ErrorStyle ?? (s\_ErrorStyle = new GUIStyle("CN StatusError"));

            }

        }

        public static GUIStyle warningStyle

        {

            get

            {

                return s\_WarningStyle ?? (s\_WarningStyle = new GUIStyle("CN StatusWarn"));

            }

        }

        public static GUIStyle infoStyle

        {

            get

            {

                return s\_InfoStyle ?? (s\_InfoStyle = new GUIStyle("CN StatusInfo"));

            }

        }

    }

}

пїњusing System.Collections.Generic;

using System.Linq;

using UnityEngine;

using UnityEngine.Experimental.UIElements;

namespace UnityEditor.Recorder

{

    static class UIElementHelper

    {

        internal static void SetFlex(VisualElement element, float value)

        {

            #if UNITY\_2018\_3\_OR\_NEWER

                element.style.flex = new Flex(value);

            #else

                element.style.flex = value;

            #endif

        }

        internal static bool GetToggleValue(Toggle toggle)

        {

            #if UNITY\_2018\_2\_OR\_NEWER

                return toggle.value;

            #else

                return toggle.on;

            #endif

        }

        internal static void SetToggleValue(Toggle toggle, bool value)

        {

            #if UNITY\_2018\_2\_OR\_NEWER

                toggle.value = value;

            #else

                toggle.on = value;

            #endif

        }

        internal static bool MultiIntField(GUIContent label, GUIContent[] subLabels, int[] values)

        {

            var r = EditorGUILayout.GetControlRect();

            var rLabel = r;

            rLabel.width = EditorGUIUtility.labelWidth;

            EditorGUI.LabelField(rLabel, label);

            var rContent = r;

            rContent.xMin = rLabel.xMax;

            var width = subLabels.Select(l => GUI.skin.label.CalcSize(l).x).Max();

            EditorGUI.BeginChangeCheck();

            MultiIntField(rContent, subLabels, values, width);

            return EditorGUI.EndChangeCheck();

        }

        internal static bool MultiFloatField(GUIContent label, GUIContent[] subLabels, float[] values)

        {

            var r = EditorGUILayout.GetControlRect();

            var rLabel = r;

            rLabel.width = EditorGUIUtility.labelWidth;

            EditorGUI.LabelField(rLabel, label);

            var rContent = r;

            rContent.xMin = rLabel.xMax;

            var width = subLabels.Select(l => GUI.skin.label.CalcSize(l).x).Max();

            EditorGUI.BeginChangeCheck();

            MultiFloatField(rContent, subLabels, values, width);

            return EditorGUI.EndChangeCheck();

        }

        static void MultiIntField(Rect position, IList<GUIContent> subLabels, IList<int> values, float labelWidth)

        {

            var length = values.Count;

            var num = (position.width - (float) (length - 1) \* 2f) / (float) length;

            var position1 = new Rect(position)

            {

                width = num

            };

            var labelWidth1 = EditorGUIUtility.labelWidth;

            var indentLevel = EditorGUI.indentLevel;

            EditorGUIUtility.labelWidth = labelWidth;

            EditorGUI.indentLevel = 0;

            for (int index = 0; index < values.Count; ++index)

            {

                values[index] = EditorGUI.IntField(position1, subLabels[index], values[index]);

                position1.x += num + 2f;

            }

            EditorGUIUtility.labelWidth = labelWidth1;

            EditorGUI.indentLevel = indentLevel;

        }

        static void MultiFloatField(Rect position, IList<GUIContent> subLabels, IList<float> values, float labelWidth)

        {

            var length = values.Count;

            var num = (position.width - (float) (length - 1) \* 2f) / (float) length;

            var position1 = new Rect(position)

            {

                width = num

            };

            var labelWidth1 = EditorGUIUtility.labelWidth;

            var indentLevel = EditorGUI.indentLevel;

            EditorGUIUtility.labelWidth = labelWidth;

            EditorGUI.indentLevel = 0;

            for (int index = 0; index < values.Count; ++index)

            {

                values[index] = EditorGUI.FloatField(position1, subLabels[index], values[index]);

                position1.x += num + 2f;

            }

            EditorGUIUtility.labelWidth = labelWidth1;

            EditorGUI.indentLevel = indentLevel;

        }

    }

}