|  |
| --- |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.SceneManagement;  using BNG;  using UnityEngine.UI;  public class Manager : MonoBehaviour  {  public Animation anima;  public List<AnimationClip> clips = new List<AnimationClip>();  public GameObject viewObj;  public GameObject leftViewObj;  public GameObject mainMenu;  public GameObject gfObj;  public GameObject dzObj;  public GameObject playAnimaObj;  public GameObject pauseAnimaObj;  private bool isForward;  private bool isRewind;  public int curIndex;  private float timer1;  private float timer2;  public Image sliderImage;  public Transform collectionButtonObj;  public GameObject collectionObj;  public Transform collectionConnect;  public List<GameObject> collectionPrefabs=new List<GameObject>();  public GameObject lookObj;  public List<GameObject> lookObjs = new List<GameObject>();  // Start is called before the first frame update  void Start()  {  playAnimaObj.SetActive(false);  for (int i = 0; i < lookObj.transform.childCount; i++)  {  lookObjs.Add(lookObj.transform.GetChild(i).gameObject);  }  }  // Update is called once per frame  void Update()  {  sliderImage.fillAmount = anima[clips[curIndex].name].normalizedTime;  if (anima[clips[curIndex].name].normalizedTime >= 0.98f)  {  anima[clips[curIndex].name].normalizedTime = 0;  pauseAnimaObj.SetActive(false);  playAnimaObj.SetActive(true);  foreach (AnimationState state in anima)  {  state.speed = 0f;  }  }  //print(anima[clips[curIndex].name].normalizedTime);  if (Input.GetKeyDown(KeyCode.T) || InputBridge.Instance.AButtonDown)  {  viewObj.SetActive(!viewObj.activeInHierarchy);  if (viewObj.activeInHierarchy)  {  mainMenu.SetActive(true);  }  else  {  CloseAll();  }  }  if (Input.GetKeyDown(KeyCode.Y) || InputBridge.Instance.XButtonDown)  {  leftViewObj.SetActive(!leftViewObj.activeInHierarchy);  }  if (isForward)  {  timer1 = timer1 + Time.deltaTime;  if (timer1 > 0.1f)  {  timer1 = 0;  isForward = false;  foreach (AnimationState state in anima)  {  state.speed = 1f;  anima.Play();  }  }  }  if (isRewind)  {  timer2 = timer2 + Time.deltaTime;  if (timer2 > 0.25f)  {  timer2 = 0;  isRewind = false;  foreach (AnimationState state in anima)  {  state.speed = 1f;  anima.Play();  }  }  }  }  public void CloseAll()  {  //viewObj.SetActive(false);  mainMenu.SetActive(false);  gfObj.SetActive(false);  dzObj.SetActive(false);  cjObj.SetActive(false);  lookObj.SetActive(false);  collectionObj.SetActive(false);  if (collectionConnect.childCount != 0)  {  for (int i = 0; i < collectionConnect.childCount; i++)  {  Destroy(collectionConnect.GetChild(i).gameObject);  }  }  collectionButtonObj.transform.GetChild(1).gameObject.SetActive(false);  collectionButtonObj.transform.GetChild(0).gameObject.SetActive(true);  for (int i = 0; i < DateManager.CollectionList.Count; i++)  {  if (DateManager.CollectionList[i] == curIndex)  {  collectionButtonObj.transform.GetChild(1).gameObject.SetActive(true);  collectionButtonObj.transform.GetChild(0).gameObject.SetActive(false);  }  }  }  public void GongFa()  {  CloseAll();  gfObj.SetActive(true);  }  public void DongZuo()  {  CloseAll();  dzObj.SetActive(true);  }  public void ChangJing()  {  CloseAll();  cjObj.SetActive(true);  }  public void ButtonF(int index)  {  collectionButtonObj.transform.GetChild(1).gameObject.SetActive(false);  collectionButtonObj.transform.GetChild(0).gameObject.SetActive(true);  anima.clip = clips[index];  anima.Play();  curIndex = index;  for (int i = 0; i < DateManager.CollectionList.Count; i++)  {  if (DateManager.CollectionList[i] == curIndex)  {  collectionButtonObj.transform.GetChild(1).gameObject.SetActive(true);  collectionButtonObj.transform.GetChild(0).gameObject.SetActive(false);  }  }  }  public void LoadScene(int index)  {  SceneManager.LoadScene(index);  }  public void PauseAnima()  {  pauseAnimaObj.SetActive(false);  playAnimaObj.SetActive(true);  foreach (AnimationState state in anima)  {  state.speed = 0f;  }  }  public void PlayAnima()  {  playAnimaObj.SetActive(false);  pauseAnimaObj.SetActive(true);  foreach (AnimationState state in anima)  {  state.speed = 1f;  }  }  public void ForwardButton()  {  anima.Play();  isForward = true;  timer1 = 0;  foreach (AnimationState state in anima)  {  state.speed = 3f;  }  }  public void RewindButton()  {    isRewind = true;  timer2 = 0;  foreach (AnimationState state in anima)  {  state.speed = -3f;  }  }  public void CollectionF()  {    if (collectionButtonObj.transform.GetChild(1).gameObject.activeInHierarchy)  {  }  else  {  collectionButtonObj.transform.GetChild(1).gameObject.SetActive(true);  collectionButtonObj.transform.GetChild(0).gameObject.SetActive(false);  if (!DateManager.CollectionList.Contains(curIndex))  {  DateManager.CollectionList.Add(curIndex);  }    }  }  public void MyCollecttionButton()  {  CloseAll();  if (DateManager.CollectionList.Count != 0)  {  float curHeight=0;  for (int i = 0; i < DateManager.CollectionList.Count; i++)  {  GameObject obj = GameObject.Instantiate(collectionPrefabs[DateManager.CollectionList[i]],collectionConnect);  obj.transform.localPosition = new Vector3(0, curHeight, 0);  curHeight = -collectionPrefabs[DateManager.CollectionList[i]].GetComponent<ListCode>().height + curHeight;  }  }  collectionObj.SetActive(true);  }  public void LookButton(int index)  {  CloseAll();  for (int i = 0; i < lookObjs.Count; i++)  {  lookObjs[i].SetActive(false);  }  lookObj.SetActive(true);  lookObjs[index].SetActive(true);  }  public void QuitButton()  {  Application.Quit();  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  namespace BNG {  /// <summary>  /// A Grabbable object that can stick to objects and deal damage  /// </summary>  public class Arrow : MonoBehaviour {  Rigidbody rb;  Grabbable grab;  public bool Flying = false;  public float ZVel = 0;  public Collider ShaftCollider;  AudioSource impactSound;  float flightTime = 0f;  float destroyTime = 10f; // Time in seconds to destroy arrow  Coroutine queueDestroy;  public Projectile ProjectileObject;  // Get this value from the ProjectileObject  float arrowDamage;  // Start is called before the first frame update  void Start() {  rb = GetComponent<Rigidbody>();  impactSound = GetComponent<AudioSource>();  ShaftCollider = GetComponent<Collider>();  grab = GetComponent<Grabbable>();  if(ProjectileObject == null) {  ProjectileObject = gameObject.AddComponent<Projectile>();  ProjectileObject.Damage = 50;  ProjectileObject.StickToObject = true;  ProjectileObject.enabled = false;  }  arrowDamage = ProjectileObject.Damage;  }  void FixedUpdate() {  bool beingHeld = grab != null && grab.BeingHeld;  // Align arrow with velocity  if (!beingHeld && rb != null && rb.velocity != Vector3.zero && Flying && ZVel > 0.02) {  rb.rotation = Quaternion.LookRotation(rb.velocity);  }  ZVel = transform.InverseTransformDirection(rb.velocity).z;  if (Flying) {  flightTime += Time.fixedDeltaTime;  }  // Cancel Destroy if we just picked this up  if(queueDestroy != null && grab != null && grab.BeingHeld) {  StopCoroutine(queueDestroy);  }  }  public void ShootArrow(Vector3 shotForce) {  flightTime = 0f;  Flying = true;  transform.parent = null;  rb.isKinematic = false;  rb.useGravity = true;  rb.collisionDetectionMode = CollisionDetectionMode.Continuous;  rb.constraints = RigidbodyConstraints.None;  rb.AddForce(shotForce, ForceMode.VelocityChange);  StartCoroutine(ReEnableCollider());  queueDestroy = StartCoroutine(QueueDestroy());  }  IEnumerator QueueDestroy() {  yield return new WaitForSeconds(destroyTime);  if (grab != null && !grab.BeingHeld && transform.parent == null) {  Destroy(this.gameObject);  }  }  IEnumerator ReEnableCollider() {  int waitFrames = 3;  for (int x = 0; x < waitFrames; x++) {  yield return new WaitForFixedUpdate();  }  ShaftCollider.enabled = true;  }  private void OnCollisionEnter(Collision collision) {  // Ignore parent collisions  if (transform.parent != null && collision.transform == transform.parent) {  return;  }  // Don't count collisions if being held  if(grab != null && grab.BeingHeld) {  return;  }  // Don't Count Triggers  if(collision.collider.isTrigger) {  return;  }  string colNameLower = collision.transform.name.ToLower();  // Ignore other very close bows and arrows  if (flightTime < 1 && (colNameLower.Contains("arrow") || colNameLower.Contains("bow"))) {  Physics.IgnoreCollision(collision.collider, ShaftCollider, true);  return;  }  // ignore player collision if quick shot  if (flightTime < 1 && collision.transform.name.ToLower().Contains("player")) {  Physics.IgnoreCollision(collision.collider, ShaftCollider, true);  return;  }  // Damage if possible  float zVel = System.Math.Abs(transform.InverseTransformDirection(rb.velocity).z);  bool doStick = true;  if (zVel > 0.02f && !rb.isKinematic) {  Damageable d = collision.gameObject.GetComponent<Damageable>();  if (d) {  d.DealDamage(arrowDamage, collision.GetContact(0).point, collision.GetContact(0).normal, true, gameObject, collision.collider.gameObject);  }  // Don't stick to dead objects  if (d != null && d.Health <= 0) {  doStick = false;  }  }  // Check to stick to object  if (!rb.isKinematic && Flying) {  if (zVel > 0.02f) {  if (grab != null && grab.BeingHeld) {  grab.DropItem(false, false);  }  if (doStick) {  tryStickArrow(collision);  }  Flying = false;  playSoundInterval(2.462f, 2.68f);  }  }  }  // Attach to collider  void tryStickArrow(Collision collision) {  Rigidbody colRigid = collision.collider.GetComponent<Rigidbody>();  transform.parent = null; // Start out with arrow being in World space  // If the collider is static then we don't need to do anything. Just stop it.  if (collision.gameObject.isStatic) {  rb.collisionDetectionMode = CollisionDetectionMode.Discrete;  rb.isKinematic = true;  }  // Next try attaching to rigidbody via FixedJoint  else if (colRigid != null && !colRigid.isKinematic) {  FixedJoint joint = gameObject.AddComponent<FixedJoint>();  joint.connectedBody = colRigid;  joint.enableCollision = false;  joint.breakForce = float.MaxValue;  joint.breakTorque = float.MaxValue;  }  else if (colRigid != null && colRigid.isKinematic && collision.transform.localScale == Vector3.one) {  transform.SetParent(collision.transform);  rb.useGravity = false;  rb.collisionDetectionMode = CollisionDetectionMode.Discrete;  rb.isKinematic = true;  rb.constraints = RigidbodyConstraints.FreezeAll;  rb.WakeUp();  }  // Finally, try parenting or just setting the arrow to kinematic  else {  if (collision.transform.localScale == Vector3.one) {  transform.SetParent(collision.transform);  rb.constraints = RigidbodyConstraints.FreezeAll;  }  else {  rb.collisionDetectionMode = CollisionDetectionMode.Discrete;  rb.useGravity = false;  rb.isKinematic = true;  }  }  }  void playSoundInterval(float fromSeconds, float toSeconds) {  if (impactSound) {  if (impactSound.isPlaying) {  impactSound.Stop();  }  impactSound.time = fromSeconds;  impactSound.pitch = Time.timeScale;  impactSound.Play();  impactSound.SetScheduledEndTime(AudioSettings.dspTime + (toSeconds - fromSeconds));  }  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  namespace BNG {  public class ArrowGrabArea : MonoBehaviour {  Bow theBow;  // Start is called before the first frame update  void Start() {  theBow = transform.parent.GetComponent<Bow>();  }  void OnTriggerEnter(Collider other) {  // Grabber entered grab area. We can potentially grab an arrow  Grabber grabObject = other.GetComponent<Grabber>();  if (grabObject != null) {  theBow.ClosestGrabber = grabObject;  // Not holding anything.  if (!grabObject.HoldingItem) {  theBow.CanGrabArrow = true;  }  // Holding an arrow  else if(grabObject.HoldingItem && grabObject.HeldGrabbable != null) {  // A held Arrow entered the grab area but has not yet been knocked  Arrow arrowObject = grabObject.HeldGrabbable.GetComponent<Arrow>();  if (arrowObject != null && theBow.GrabbedArrow == null) {  theBow.GrabArrow(arrowObject);  }  }  }  }  void OnTriggerExit(Collider other) {  // Grabber exited grab area. No longer able to grab an arrow  Grabber grabObject = other.GetComponent<Grabber>();  if (theBow.ClosestGrabber != null && grabObject != null && theBow.ClosestGrabber == grabObject) {  theBow.CanGrabArrow = false;  theBow.ClosestGrabber = null;  }  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  namespace BNG {  public class AutoGrabGrabbable : GrabbableEvents {  public override void OnBecomesClosestGrabbable(Grabber touchingGrabber) {  touchingGrabber.GrabGrabbable(grab);  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using System.Linq;  using UnityEngine;  namespace BNG {  [RequireComponent(typeof(HandPoser))]  [ExecuteInEditMode]  public class AutoPoser : MonoBehaviour {  [Header("Auto Pose Settings")]  [Tooltip("(Required) A HandPose in the fully open position. AutoPose will lerp each finger joint between OpenHandPose and ClosedHandPose until contact is made at each finger tip.")]  public HandPose OpenHandPose;  [Tooltip("(Required) A HandPose in the fully closed position. AutoPose will lerp each finger between OpenHandPose and ClosedHandPose until contact is made at each finger tip.")]  public HandPose ClosedHandPose;  [Header("Finger Tip Collision")]  [Tooltip("Radius (in meters) of the fingertips to use when checking for collisions during auto-posing. Only used if a FingerTipCollider is not defined. (Default : 0.00875)")]  [Range(0.0f, 0.02f)]  public float FingerTipRadius = 0.00875f;  [Tooltip("(Optional) Index Finger Offset - Use this to manually position and scale your finger tip collider")]  public FingerTipCollider ThumbCollider;  [Tooltip("(Optional) Index Finger Offset - Use this to manually position and scale your finger tip collider")]  public FingerTipCollider IndexFingerCollider;  [Tooltip("(Optional) Index Finger Offset - Use this to manually position and scale your finger tip collider")]  public FingerTipCollider MiddleFingerCollider;  [Tooltip("(Optional) Index Finger Offset - Use this to manually position and scale your finger tip collider")]  public FingerTipCollider RingFingerCollider;  [Tooltip("(Optional) Index Finger Offset - Use this to manually position and scale your finger tip collider")]  public FingerTipCollider PinkyFingerCollider;  [Header("Continuous Update")]  [Tooltip("If true the hand will auto pose in Update(). Also works in the editor.")]  public bool UpdateContinuously = false;  [Tooltip("(Optional) The HandPose to use when UpdateContinuously = true and no collisions have been detected. If not specified, the hand will make a ClosedHandPose shape when no collisions have been found.")]  public HandPose IdleHandPose;  public LayerMask CollisionLayerMask = ~0;  // Gizmo Customization  [Header("Editor Gizmos")]  public bool ShowGizmos = true;  public GizmoDisplayType GizmoType = GizmoDisplayType.Wire;  public Color GizmoColor = Color.white;  public HandPoser InspectedPose;  // Where our hand currently is this frame  HandPoseDefinition currentPose;  // Used to store temp poses and prevent GC  HandPoseDefinition tempPose;  // Where our hand would after checking for collisions  public HandPoseDefinition CollisionPose {  get {  return collisionPose;  }  }  HandPoseDefinition collisionPose;  public bool CollisionDetected {  get {  return \_thumbHit || \_indexHit || \_middleHit || \_ringHit || \_pinkyHit;  }  }  #region private variables  private int \_count = 0;  private bool \_thumbHit = false;  private bool \_indexHit = false;  private bool \_middleHit = false;  private bool \_ringHit = false;  private bool \_pinkyHit = false;  #endregion  void Start() {  if(InspectedPose == null) {  InspectedPose = GetComponent<HandPoser>();  }  }  void OnEnable() {  if(Application.isEditor && OpenHandPose == null && ClosedHandPose == null) {  // Try to auto fill open / closed hand pose  OpenHandPose = (HandPose)Resources.Load("Default", typeof(HandPose));  ClosedHandPose = (HandPose)Resources.Load("Fist", typeof(HandPose));  }  }  // Update is called once per frame  void Update() {  // Auto Update Auto Pose  if (UpdateContinuously) {  bool useIdlePose = CollisionDetected == false && IdleHandPose != null && InspectedPose != null;  if(useIdlePose) {  collisionPose = GetAutoPose();  InspectedPose.UpdateHandPose(IdleHandPose.Joints, true);  }  else {  UpdateAutoPose(true);  }  }  }  public virtual void UpdateAutoPose(bool lerp) {  collisionPose = GetAutoPose();  // Lerp to collision hand pose  if (collisionPose != null) {  InspectedPose.UpdateHandPose(collisionPose, lerp);  }  }  public virtual void UpdateAutoPoseOnce() {  StartCoroutine(updateAutoPoseRoutine());  }  IEnumerator updateAutoPoseRoutine() {  UpdateContinuously = true;  yield return new WaitForSeconds(0.2f);  UpdateContinuously = false;  }  public HandPoseDefinition GetAutoPose() {  if (InspectedPose == null || OpenHandPose == null || ClosedHandPose == null) {  return null;  }  // Store our current hand state so we can snap back to it after we determine a collision  currentPose = CopyHandDefinition(InspectedPose.GetHandPoseDefinition());  // Start in open pose before  InspectedPose.UpdateHandPose(OpenHandPose.Joints, false);  // Lerp between Open and Closed hand position, stopping if we hit something  \_count = 0;  \_thumbHit = false;  \_indexHit = false;  \_middleHit = false;  \_ringHit = false;  \_pinkyHit = false;  while (\_count < 300) {  // Check Tip Collision  if (!\_thumbHit) { \_thumbHit = GetThumbHit(InspectedPose); }  if (!\_indexHit) { \_indexHit = GetIndexHit(InspectedPose); }  if (!\_middleHit) { \_middleHit = GetMiddleHit(InspectedPose); }  if (!\_ringHit) { \_ringHit = GetRingHit(InspectedPose); }  if (!\_pinkyHit) { \_pinkyHit = GetPinkyHit(InspectedPose); }  // Can bail if everything is touching  if (\_thumbHit && \_indexHit && \_middleHit && \_ringHit && \_pinkyHit) {  break;  }  \_count++;  }  // Store the collision pose so we can return it after we've reset out hand back to it's original position  tempPose = CopyHandDefinition(InspectedPose.GetHandPoseDefinition());    // Switch back to the original hand pose we were in  InspectedPose.UpdateHandPose(currentPose, false);  return tempPose;  }  public HandPoseDefinition CopyHandDefinition(HandPoseDefinition ToCopy) {  return new HandPoseDefinition() {  IndexJoints = GetJointsCopy(ToCopy.IndexJoints),  MiddleJoints = GetJointsCopy(ToCopy.MiddleJoints),  OtherJoints = GetJointsCopy(ToCopy.OtherJoints),  PinkyJoints = GetJointsCopy(ToCopy.PinkyJoints),  RingJoints = GetJointsCopy(ToCopy.RingJoints),  ThumbJoints = GetJointsCopy(ToCopy.ThumbJoints),  WristJoint = GetJointCopy(ToCopy.WristJoint),  };  }  public FingerJoint GetJointCopy(FingerJoint ToClone) {  // Null check  if(ToClone == null) {  return null;  }  return new FingerJoint() {  LocalPosition = ToClone.LocalPosition,  LocalRotation = ToClone.LocalRotation,  TransformName = ToClone.TransformName  };  }  public List<FingerJoint> GetJointsCopy(List<FingerJoint> ToClone) {  List<FingerJoint> joints = new List<FingerJoint>();  for (int x = 0; x < ToClone.Count; x++) {  joints.Add(GetJointCopy(ToClone[x]));  }  return joints;  }  public bool GetThumbHit(HandPoser poser) {  if (ThumbCollider != null) {  return LoopThroughJoints(poser.ThumbJoints, ClosedHandPose.Joints.ThumbJoints, ThumbCollider.transform.position, ThumbCollider.Radius);  }  else {  return LoopThroughJoints(poser.ThumbJoints, ClosedHandPose.Joints.ThumbJoints, HandPoser.GetFingerTipPositionWithOffset(poser.ThumbJoints, FingerTipRadius), FingerTipRadius);  }  }  public bool GetIndexHit(HandPoser poser) {  if (IndexFingerCollider != null) {  return LoopThroughJoints(poser.IndexJoints, ClosedHandPose.Joints.IndexJoints, IndexFingerCollider.transform.position, IndexFingerCollider.Radius);  }  else {  return LoopThroughJoints(poser.IndexJoints, ClosedHandPose.Joints.IndexJoints, HandPoser.GetFingerTipPositionWithOffset(poser.IndexJoints, FingerTipRadius), FingerTipRadius);  }  }  public bool GetMiddleHit(HandPoser poser) {  if (MiddleFingerCollider != null) {  return LoopThroughJoints(poser.MiddleJoints, ClosedHandPose.Joints.MiddleJoints, MiddleFingerCollider.transform.position, MiddleFingerCollider.Radius);  }  else {  return LoopThroughJoints(poser.MiddleJoints, ClosedHandPose.Joints.MiddleJoints, HandPoser.GetFingerTipPositionWithOffset(poser.MiddleJoints, FingerTipRadius), FingerTipRadius);  }  }  public bool GetRingHit(HandPoser poser) {  if (RingFingerCollider != null) {  return LoopThroughJoints(poser.RingJoints, ClosedHandPose.Joints.RingJoints, RingFingerCollider.transform.position, RingFingerCollider.Radius);  }  else {  return LoopThroughJoints(poser.RingJoints, ClosedHandPose.Joints.RingJoints, HandPoser.GetFingerTipPositionWithOffset(poser.RingJoints, FingerTipRadius), FingerTipRadius);  }  }  public bool GetPinkyHit(HandPoser poser) {  if (PinkyFingerCollider != null) {  return LoopThroughJoints(poser.PinkyJoints, ClosedHandPose.Joints.PinkyJoints, PinkyFingerCollider.transform.position, PinkyFingerCollider.Radius);  }  else {  return LoopThroughJoints(poser.PinkyJoints, ClosedHandPose.Joints.PinkyJoints, HandPoser.GetFingerTipPositionWithOffset(poser.PinkyJoints, FingerTipRadius), FingerTipRadius);  }  }  public virtual bool LoopThroughJoints(List<Transform> fromJoints, List<FingerJoint> toJoints, Vector3 tipPosition, float tipRadius) {  // Not a complete set  if(fromJoints == null || toJoints == null || toJoints.Count == 0) {  return false;  }  int fromLength = fromJoints.Count;  int toLength = toJoints.Count;  float movementAmount = 1f;  for (int x = 0; x < fromLength; x++) {  // move towards destination  Transform thisJoint = fromJoints[x];  // Bone count mismatch  if(x >= toLength) {  return false;  }  thisJoint.localPosition = Vector3.MoveTowards(thisJoint.localPosition, toJoints[x].LocalPosition, movementAmount);  thisJoint.localRotation = Quaternion.RotateTowards(thisJoint.localRotation, toJoints[x].LocalRotation, movementAmount);  // Do Sphere cast at tip of finger  bool isFingerTip = x == fromLength - 1;  if (isFingerTip) {  Collider[] hitColliders = Physics.OverlapSphere(tipPosition, tipRadius, CollisionLayerMask, QueryTriggerInteraction.Ignore);  if (hitColliders != null && hitColliders.Length > 0) {  for(int y = 0; y < hitColliders.Length; y++) {  if(IsValidCollision(hitColliders[y])) {  return true;  }  }  }  }  }  return false;  }  /// <summary>  /// You can overrride this method to add your own collision validation logic  /// </summary>  public virtual bool IsValidCollision(Collider col) {  // Ignore Triggers  if (col == null || col.isTrigger) {  return false;  }  // Default to a valid collisions  return true;  }  #region EditorGizmos  void OnDrawGizmos() {  // Don't draw gizmos if this component has been disabled  if(!this.isActiveAndEnabled) {  return;  }  if(InspectedPose == null) {  InspectedPose = GetComponent<HandPoser>();  }  if (ShowGizmos && InspectedPose != null) {  Gizmos.color = GizmoColor;  DrawJointGizmo(ThumbCollider, HandPoser.GetFingerTipPositionWithOffset(InspectedPose.ThumbJoints, FingerTipRadius), FingerTipRadius, GizmoType);  DrawJointGizmo(IndexFingerCollider, HandPoser.GetFingerTipPositionWithOffset(InspectedPose.IndexJoints, FingerTipRadius), FingerTipRadius, GizmoType);  DrawJointGizmo(MiddleFingerCollider, HandPoser.GetFingerTipPositionWithOffset(InspectedPose.MiddleJoints, FingerTipRadius), FingerTipRadius, GizmoType);  DrawJointGizmo(RingFingerCollider, HandPoser.GetFingerTipPositionWithOffset(InspectedPose.RingJoints, FingerTipRadius), FingerTipRadius, GizmoType);  DrawJointGizmo(PinkyFingerCollider, HandPoser.GetFingerTipPositionWithOffset(InspectedPose.PinkyJoints, FingerTipRadius), FingerTipRadius, GizmoType);  }  }  public void DrawJointGizmo(FingerTipCollider tipCollider, Vector3 defaultPosition, float radius, GizmoDisplayType gizmoType) {  if(tipCollider != null) {  defaultPosition = tipCollider.transform.position;  radius = tipCollider.Radius;  }  if (gizmoType == GizmoDisplayType.Wire) {  Gizmos.DrawWireSphere(defaultPosition, radius);  }  else if (GizmoType == GizmoDisplayType.Solid) {  Gizmos.DrawSphere(defaultPosition, radius);  }  }  #endregion  }  public enum GizmoDisplayType {  Wire,  Solid,  None  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEditor;  using UnityEngine;  namespace BNG {  [CustomEditor(typeof(AutoPoser))]  [CanEditMultipleObjects]  public class AutoPoserEditor : Editor {    // Auto Pose properties  SerializedProperty openHandPose;  SerializedProperty closedHandPose;  SerializedProperty autoUpdateAutoPose;  SerializedProperty idleHandPose;    SerializedProperty fingerTipRadius;  SerializedProperty collisionLayerMask;  SerializedProperty showGizmos;  SerializedProperty gizmoType;  SerializedProperty gizmoColor;  SerializedProperty thumbCollider;  SerializedProperty indexFingerCollider;  SerializedProperty middleFingerCollider;  SerializedProperty ringFingerCollider;  SerializedProperty pinkyFingerCollider;  AutoPoser poser;  bool showGizmoProps;  bool showColliderOffsets;  void OnEnable() {  openHandPose = serializedObject.FindProperty("OpenHandPose");  closedHandPose = serializedObject.FindProperty("ClosedHandPose");  autoUpdateAutoPose = serializedObject.FindProperty("UpdateContinuously");  idleHandPose = serializedObject.FindProperty("IdleHandPose");  fingerTipRadius = serializedObject.FindProperty("FingerTipRadius");  collisionLayerMask = serializedObject.FindProperty("CollisionLayerMask");  showGizmos = serializedObject.FindProperty("ShowGizmos");  gizmoType = serializedObject.FindProperty("GizmoType");  gizmoColor = serializedObject.FindProperty("GizmoColor");  thumbCollider = serializedObject.FindProperty("ThumbCollider");  indexFingerCollider = serializedObject.FindProperty("IndexFingerCollider");  middleFingerCollider = serializedObject.FindProperty("MiddleFingerCollider");  ringFingerCollider = serializedObject.FindProperty("RingFingerCollider");  pinkyFingerCollider = serializedObject.FindProperty("PinkyFingerCollider");  }  public override void OnInspectorGUI() {  poser = (AutoPoser)target;  serializedObject.Update();  GUILayout.Label("Auto Pose", EditorStyles.boldLabel);  EditorGUILayout.PropertyField(openHandPose);  EditorGUILayout.PropertyField(closedHandPose);  EditorGUILayout.PropertyField(idleHandPose);  EditorGUILayout.PropertyField(collisionLayerMask);  EditorGUILayout.PropertyField(fingerTipRadius);  showColliderOffsets = EditorGUILayout.Foldout(showColliderOffsets, "Finger Tip Offsets");  if (showColliderOffsets) {  EditorGUILayout.HelpBox("You can manually adjust each fingertip's position / scale by specifying a FingerTipCollider object below. Press the 'Auto Add Tip Colliders' to create and populate the objects for you.", MessageType.Info);  if (GUILayout.Button("Auto Setup Finger Tip Colliders")) {  AutoAddFingerTipColliders(poser);  }  EditorGUILayout.PropertyField(thumbCollider);  EditorGUILayout.PropertyField(indexFingerCollider);  EditorGUILayout.PropertyField(middleFingerCollider);  EditorGUILayout.PropertyField(ringFingerCollider);  EditorGUILayout.PropertyField(pinkyFingerCollider);  }  EditorGUILayout.Separator();  EditorGUILayout.PropertyField(autoUpdateAutoPose);  EditorGUILayout.Separator();  if (GUILayout.Button("Auto Pose")) {  poser.UpdateAutoPose(false);  }  // GUILayout.Label("Editor Gizmos", EditorStyles.boldLabel);  showGizmoProps = EditorGUILayout.Foldout(showGizmoProps, "Editor Gizmos");  if (showGizmoProps) {  EditorGUILayout.PropertyField(showGizmos);  EditorGUILayout.PropertyField(gizmoType);  EditorGUILayout.PropertyField(gizmoColor);  }  serializedObject.ApplyModifiedProperties();  }  public void AutoAddFingerTipColliders(AutoPoser poser) {  poser.ThumbCollider = GetOrAddTipCollider(poser.InspectedPose.GetThumbTip(), "tip\_collider\_t");  poser.IndexFingerCollider = GetOrAddTipCollider(poser.InspectedPose.GetIndexTip(), "tip\_collider\_i");  poser.MiddleFingerCollider = GetOrAddTipCollider(poser.InspectedPose.GetMiddleTip(), "tip\_collider\_m");  poser.RingFingerCollider = GetOrAddTipCollider(poser.InspectedPose.GetRingTip(), "tip\_collider\_r");  poser.PinkyFingerCollider = GetOrAddTipCollider(poser.InspectedPose.GetPinkyTip(), "tip\_collider\_p");  }  public FingerTipCollider GetOrAddTipCollider(Transform tipTransform, string tipName) {  if (tipTransform != null) {  // Check for existing and return that if available  FingerTipCollider col = tipTransform.GetComponentInChildren<FingerTipCollider>();  if(col != null) {  return col;  }  // Otherwise create a new one and parent it to the tip of the finger  GameObject tipCollider = new GameObject(tipName);  col = tipCollider.AddComponent<FingerTipCollider>();  tipCollider.transform.parent = tipTransform;  tipCollider.transform.localPosition = Vector3.zero;  tipCollider.transform.localEulerAngles = Vector3.zero;  tipCollider.transform.localScale = Vector3.one;  return col;  }  return null;  }  }  } |
| using System;  using UnityEngine;  namespace UnityStandardAssets.ImageEffects  {  [ExecuteInEditMode]  [RequireComponent (typeof(Camera))]  [AddComponentMenu ("Image Effects/Bloom and Glow/Bloom")]  public class Bloom : PostEffectsBase  {  public enum LensFlareStyle  {  Ghosting = 0,  Anamorphic = 1,  Combined = 2,  }  public enum TweakMode  {  Basic = 0,  Complex = 1,  }  public enum HDRBloomMode  {  Auto = 0,  On = 1,  Off = 2,  }  public enum BloomScreenBlendMode  {  Screen = 0,  Add = 1,  }  public enum BloomQuality  {  Cheap = 0,  High = 1,  }  public TweakMode tweakMode = 0;  public BloomScreenBlendMode screenBlendMode = BloomScreenBlendMode.Add;  public HDRBloomMode hdr = HDRBloomMode.Auto;  private bool doHdr = false;  public float sepBlurSpread = 2.5f;  public BloomQuality quality = BloomQuality.High;  public float bloomIntensity = 0.5f;  public float bloomThreshold = 0.5f;  public Color bloomThresholdColor = Color.white;  public int bloomBlurIterations = 2;  public int hollywoodFlareBlurIterations = 2;  public float flareRotation = 0.0f;  public LensFlareStyle lensflareMode = (LensFlareStyle) 1;  public float hollyStretchWidth = 2.5f;  public float lensflareIntensity = 0.0f;  public float lensflareThreshold = 0.3f;  public float lensFlareSaturation = 0.75f;  public Color flareColorA = new Color (0.4f, 0.4f, 0.8f, 0.75f);  public Color flareColorB = new Color (0.4f, 0.8f, 0.8f, 0.75f);  public Color flareColorC = new Color (0.8f, 0.4f, 0.8f, 0.75f);  public Color flareColorD = new Color (0.8f, 0.4f, 0.0f, 0.75f);  public Texture2D lensFlareVignetteMask;  public Shader lensFlareShader;  private Material lensFlareMaterial;  public Shader screenBlendShader;  private Material screenBlend;  public Shader blurAndFlaresShader;  private Material blurAndFlaresMaterial;  public Shader brightPassFilterShader;  private Material brightPassFilterMaterial;  public override bool CheckResources ()  {  CheckSupport (false);  screenBlend = CheckShaderAndCreateMaterial (screenBlendShader, screenBlend);  lensFlareMaterial = CheckShaderAndCreateMaterial(lensFlareShader,lensFlareMaterial);  blurAndFlaresMaterial = CheckShaderAndCreateMaterial (blurAndFlaresShader, blurAndFlaresMaterial);  brightPassFilterMaterial = CheckShaderAndCreateMaterial(brightPassFilterShader, brightPassFilterMaterial);  if (!isSupported)  ReportAutoDisable ();  return isSupported;  }  public void OnRenderImage (RenderTexture source, RenderTexture destination)  {  if (CheckResources()==false)  {  Graphics.Blit (source, destination);  return;  }  // screen blend is not supported when HDR is enabled (will cap values)  doHdr = false;  if (hdr == HDRBloomMode.Auto)  doHdr = source.format == RenderTextureFormat.ARGBHalf && GetComponent<Camera>().allowHDR;  else {  doHdr = hdr == HDRBloomMode.On;  }  doHdr = doHdr && supportHDRTextures;  BloomScreenBlendMode realBlendMode = screenBlendMode;  if (doHdr)  realBlendMode = BloomScreenBlendMode.Add;  var rtFormat= (doHdr) ? RenderTextureFormat.ARGBHalf : RenderTextureFormat.Default;  var rtW2= source.width/2;  var rtH2= source.height/2;  var rtW4= source.width/4;  var rtH4= source.height/4;  float widthOverHeight = (1.0f \* source.width) / (1.0f \* source.height);  float oneOverBaseSize = 1.0f / 512.0f;  // downsample  RenderTexture quarterRezColor = RenderTexture.GetTemporary (rtW4, rtH4, 0, rtFormat);  RenderTexture halfRezColorDown = RenderTexture.GetTemporary (rtW2, rtH2, 0, rtFormat);  if (quality > BloomQuality.Cheap) {  Graphics.Blit (source, halfRezColorDown, screenBlend, 2);  RenderTexture rtDown4 = RenderTexture.GetTemporary (rtW4, rtH4, 0, rtFormat);  Graphics.Blit (halfRezColorDown, rtDown4, screenBlend, 2);  Graphics.Blit (rtDown4, quarterRezColor, screenBlend, 6);  RenderTexture.ReleaseTemporary(rtDown4);  }  else {  Graphics.Blit (source, halfRezColorDown);  Graphics.Blit (halfRezColorDown, quarterRezColor, screenBlend, 6);  }  RenderTexture.ReleaseTemporary (halfRezColorDown);  // cut colors (thresholding)  RenderTexture secondQuarterRezColor = RenderTexture.GetTemporary (rtW4, rtH4, 0, rtFormat);  BrightFilter (bloomThreshold \* bloomThresholdColor, quarterRezColor, secondQuarterRezColor);  // blurring  if (bloomBlurIterations < 1) bloomBlurIterations = 1;  else if (bloomBlurIterations > 10) bloomBlurIterations = 10;  for (int iter = 0; iter < bloomBlurIterations; iter++)  {  float spreadForPass = (1.0f + (iter \* 0.25f)) \* sepBlurSpread;  // vertical blur  RenderTexture blur4 = RenderTexture.GetTemporary (rtW4, rtH4, 0, rtFormat);  blurAndFlaresMaterial.SetVector ("\_Offsets", new Vector4 (0.0f, spreadForPass \* oneOverBaseSize, 0.0f, 0.0f));  Graphics.Blit (secondQuarterRezColor, blur4, blurAndFlaresMaterial, 4);  RenderTexture.ReleaseTemporary(secondQuarterRezColor);  secondQuarterRezColor = blur4;  // horizontal blur  blur4 = RenderTexture.GetTemporary (rtW4, rtH4, 0, rtFormat);  blurAndFlaresMaterial.SetVector ("\_Offsets", new Vector4 ((spreadForPass / widthOverHeight) \* oneOverBaseSize, 0.0f, 0.0f, 0.0f));  Graphics.Blit (secondQuarterRezColor, blur4, blurAndFlaresMaterial, 4);  RenderTexture.ReleaseTemporary (secondQuarterRezColor);  secondQuarterRezColor = blur4;  if (quality > BloomQuality.Cheap)  {  if (iter == 0)  {  Graphics.SetRenderTarget(quarterRezColor);  GL.Clear(false, true, Color.black); // Clear to avoid RT restore  Graphics.Blit (secondQuarterRezColor, quarterRezColor);  }  else  {  quarterRezColor.MarkRestoreExpected(); // using max blending, RT restore expected  Graphics.Blit (secondQuarterRezColor, quarterRezColor, screenBlend, 10);  }  }  }  if (quality > BloomQuality.Cheap)  {  Graphics.SetRenderTarget(secondQuarterRezColor);  GL.Clear(false, true, Color.black); // Clear to avoid RT restore  Graphics.Blit (quarterRezColor, secondQuarterRezColor, screenBlend, 6);  }  // lens flares: ghosting, anamorphic or both (ghosted anamorphic flares)  if (lensflareIntensity > Mathf.Epsilon)  {  RenderTexture rtFlares4 = RenderTexture.GetTemporary (rtW4, rtH4, 0, rtFormat);  if (lensflareMode == 0)  {  // ghosting only  BrightFilter (lensflareThreshold, secondQuarterRezColor, rtFlares4);  if (quality > BloomQuality.Cheap)  {  // smooth a little  blurAndFlaresMaterial.SetVector ("\_Offsets", new Vector4 (0.0f, (1.5f) / (1.0f \* quarterRezColor.height), 0.0f, 0.0f));  Graphics.SetRenderTarget(quarterRezColor);  GL.Clear(false, true, Color.black); // Clear to avoid RT restore  Graphics.Blit (rtFlares4, quarterRezColor, blurAndFlaresMaterial, 4);  blurAndFlaresMaterial.SetVector ("\_Offsets", new Vector4 ((1.5f) / (1.0f \* quarterRezColor.width), 0.0f, 0.0f, 0.0f));  Graphics.SetRenderTarget(rtFlares4);  GL.Clear(false, true, Color.black); // Clear to avoid RT restore  Graphics.Blit (quarterRezColor, rtFlares4, blurAndFlaresMaterial, 4);  }  // no ugly edges!  Vignette (0.975f, rtFlares4, rtFlares4);  BlendFlares (rtFlares4, secondQuarterRezColor);  }  else  {  //Vignette (0.975ff, rtFlares4, rtFlares4);  //DrawBorder(rtFlares4, screenBlend, 8);  float flareXRot = 1.0f \* Mathf.Cos(flareRotation);  float flareyRot = 1.0f \* Mathf.Sin(flareRotation);  float stretchWidth = (hollyStretchWidth \* 1.0f / widthOverHeight) \* oneOverBaseSize;  blurAndFlaresMaterial.SetVector ("\_Offsets", new Vector4 (flareXRot, flareyRot, 0.0f, 0.0f));  blurAndFlaresMaterial.SetVector ("\_Threshhold", new Vector4 (lensflareThreshold, 1.0f, 0.0f, 0.0f));  blurAndFlaresMaterial.SetVector ("\_TintColor", new Vector4 (flareColorA.r, flareColorA.g, flareColorA.b, flareColorA.a) \* flareColorA.a \* lensflareIntensity);  blurAndFlaresMaterial.SetFloat ("\_Saturation", lensFlareSaturation);  // "pre and cut"  quarterRezColor.DiscardContents();  Graphics.Blit (rtFlares4, quarterRezColor, blurAndFlaresMaterial, 2);  // "post"  rtFlares4.DiscardContents();  Graphics.Blit (quarterRezColor, rtFlares4, blurAndFlaresMaterial, 3);  blurAndFlaresMaterial.SetVector ("\_Offsets", new Vector4 (flareXRot \* stretchWidth, flareyRot \* stretchWidth, 0.0f, 0.0f));  // stretch 1st  blurAndFlaresMaterial.SetFloat ("\_StretchWidth", hollyStretchWidth);  quarterRezColor.DiscardContents();  Graphics.Blit (rtFlares4, quarterRezColor, blurAndFlaresMaterial, 1);  // stretch 2nd  blurAndFlaresMaterial.SetFloat ("\_StretchWidth", hollyStretchWidth \* 2.0f);  rtFlares4.DiscardContents();  Graphics.Blit (quarterRezColor, rtFlares4, blurAndFlaresMaterial, 1);  // stretch 3rd  blurAndFlaresMaterial.SetFloat ("\_StretchWidth", hollyStretchWidth \* 4.0f);  quarterRezColor.DiscardContents();  Graphics.Blit (rtFlares4, quarterRezColor, blurAndFlaresMaterial, 1);  // additional blur passes  for (int iter = 0; iter < hollywoodFlareBlurIterations; iter++)  {  stretchWidth = (hollyStretchWidth \* 2.0f / widthOverHeight) \* oneOverBaseSize;  blurAndFlaresMaterial.SetVector ("\_Offsets", new Vector4 (stretchWidth \* flareXRot, stretchWidth \* flareyRot, 0.0f, 0.0f));  rtFlares4.DiscardContents();  Graphics.Blit (quarterRezColor, rtFlares4, blurAndFlaresMaterial, 4);  blurAndFlaresMaterial.SetVector ("\_Offsets", new Vector4 (stretchWidth \* flareXRot, stretchWidth \* flareyRot, 0.0f, 0.0f));  quarterRezColor.DiscardContents();  Graphics.Blit (rtFlares4, quarterRezColor, blurAndFlaresMaterial, 4);  }  if (lensflareMode == (LensFlareStyle) 1)  // anamorphic lens flares  AddTo (1.0f, quarterRezColor, secondQuarterRezColor);  else  {  // "combined" lens flares  Vignette (1.0f, quarterRezColor, rtFlares4);  BlendFlares (rtFlares4, quarterRezColor);  AddTo (1.0f, quarterRezColor, secondQuarterRezColor);  }  }  RenderTexture.ReleaseTemporary (rtFlares4);  }  int blendPass = (int) realBlendMode;  //if (Mathf.Abs(chromaticBloom) < Mathf.Epsilon)  // blendPass += 4;  screenBlend.SetFloat ("\_Intensity", bloomIntensity);  screenBlend.SetTexture ("\_ColorBuffer", source);  if (quality > BloomQuality.Cheap)  {  RenderTexture halfRezColorUp = RenderTexture.GetTemporary (rtW2, rtH2, 0, rtFormat);  Graphics.Blit (secondQuarterRezColor, halfRezColorUp);  Graphics.Blit (halfRezColorUp, destination, screenBlend, blendPass);  RenderTexture.ReleaseTemporary (halfRezColorUp);  }  else  Graphics.Blit (secondQuarterRezColor, destination, screenBlend, blendPass);  RenderTexture.ReleaseTemporary (quarterRezColor);  RenderTexture.ReleaseTemporary (secondQuarterRezColor);  }  private void AddTo (float intensity\_, RenderTexture from, RenderTexture to)  {  screenBlend.SetFloat ("\_Intensity", intensity\_);  to.MarkRestoreExpected(); // additive blending, RT restore expected  Graphics.Blit (from, to, screenBlend, 9);  }  private void BlendFlares (RenderTexture from, RenderTexture to)  {  lensFlareMaterial.SetVector ("colorA", new Vector4 (flareColorA.r, flareColorA.g, flareColorA.b, flareColorA.a) \* lensflareIntensity);  lensFlareMaterial.SetVector ("colorB", new Vector4 (flareColorB.r, flareColorB.g, flareColorB.b, flareColorB.a) \* lensflareIntensity);  lensFlareMaterial.SetVector ("colorC", new Vector4 (flareColorC.r, flareColorC.g, flareColorC.b, flareColorC.a) \* lensflareIntensity);  lensFlareMaterial.SetVector ("colorD", new Vector4 (flareColorD.r, flareColorD.g, flareColorD.b, flareColorD.a) \* lensflareIntensity);  to.MarkRestoreExpected(); // additive blending, RT restore expected  Graphics.Blit (from, to, lensFlareMaterial);  }  private void BrightFilter (float thresh, RenderTexture from, RenderTexture to)  {  brightPassFilterMaterial.SetVector ("\_Threshhold", new Vector4 (thresh, thresh, thresh, thresh));  Graphics.Blit (from, to, brightPassFilterMaterial, 0);  }  private void BrightFilter (Color threshColor, RenderTexture from, RenderTexture to)  {  brightPassFilterMaterial.SetVector ("\_Threshhold", threshColor);  Graphics.Blit (from, to, brightPassFilterMaterial, 1);  }  private void Vignette (float amount, RenderTexture from, RenderTexture to)  {  if (lensFlareVignetteMask)  {  screenBlend.SetTexture ("\_ColorBuffer", lensFlareVignetteMask);  to.MarkRestoreExpected(); // using blending, RT restore expected  Graphics.Blit (from == to ? null : from, to, screenBlend, from == to ? 7 : 3);  }  else if (from != to)  {  Graphics.SetRenderTarget (to);  GL.Clear(false, true, Color.black); // clear destination to avoid RT restore  Graphics.Blit (from, to);  }  }  }  } |
| using System;  using UnityEngine;  namespace UnityStandardAssets.ImageEffects  {  public enum LensflareStyle34  {  Ghosting = 0,  Anamorphic = 1,  Combined = 2,  }  public enum TweakMode34  {  Basic = 0,  Complex = 1,  }  public enum HDRBloomMode  {  Auto = 0,  On = 1,  Off = 2,  }  public enum BloomScreenBlendMode  {  Screen = 0,  Add = 1,  }  [ExecuteInEditMode]  [RequireComponent(typeof(Camera))]  [AddComponentMenu("Image Effects/Bloom and Glow/BloomAndFlares (3.5, Deprecated)")]  public class BloomAndFlares : PostEffectsBase  {  public TweakMode34 tweakMode = 0;  public BloomScreenBlendMode screenBlendMode = BloomScreenBlendMode.Add;  public HDRBloomMode hdr = HDRBloomMode.Auto;  private bool doHdr = false;  public float sepBlurSpread = 1.5f;  public float useSrcAlphaAsMask = 0.5f;  public float bloomIntensity = 1.0f;  public float bloomThreshold = 0.5f;  public int bloomBlurIterations = 2;  public bool lensflares = false;  public int hollywoodFlareBlurIterations = 2;  public LensflareStyle34 lensflareMode = (LensflareStyle34)1;  public float hollyStretchWidth = 3.5f;  public float lensflareIntensity = 1.0f;  public float lensflareThreshold = 0.3f;  public Color flareColorA = new Color(0.4f, 0.4f, 0.8f, 0.75f);  public Color flareColorB = new Color(0.4f, 0.8f, 0.8f, 0.75f);  public Color flareColorC = new Color(0.8f, 0.4f, 0.8f, 0.75f);  public Color flareColorD = new Color(0.8f, 0.4f, 0.0f, 0.75f);  public Texture2D lensFlareVignetteMask;  public Shader lensFlareShader;  private Material lensFlareMaterial;  public Shader vignetteShader;  private Material vignetteMaterial;  public Shader separableBlurShader;  private Material separableBlurMaterial;  public Shader addBrightStuffOneOneShader;  private Material addBrightStuffBlendOneOneMaterial;  public Shader screenBlendShader;  private Material screenBlend;  public Shader hollywoodFlaresShader;  private Material hollywoodFlaresMaterial;  public Shader brightPassFilterShader;  private Material brightPassFilterMaterial;  public override bool CheckResources()  {  CheckSupport(false);  screenBlend = CheckShaderAndCreateMaterial(screenBlendShader, screenBlend);  lensFlareMaterial = CheckShaderAndCreateMaterial(lensFlareShader, lensFlareMaterial);  vignetteMaterial = CheckShaderAndCreateMaterial(vignetteShader, vignetteMaterial);  separableBlurMaterial = CheckShaderAndCreateMaterial(separableBlurShader, separableBlurMaterial);  addBrightStuffBlendOneOneMaterial = CheckShaderAndCreateMaterial(addBrightStuffOneOneShader, addBrightStuffBlendOneOneMaterial);  hollywoodFlaresMaterial = CheckShaderAndCreateMaterial(hollywoodFlaresShader, hollywoodFlaresMaterial);  brightPassFilterMaterial = CheckShaderAndCreateMaterial(brightPassFilterShader, brightPassFilterMaterial);  if (!isSupported)  ReportAutoDisable();  return isSupported;  }  void OnRenderImage(RenderTexture source, RenderTexture destination)  {  if (CheckResources() == false)  {  Graphics.Blit(source, destination);  return;  }  // screen blend is not supported when HDR is enabled (will cap values)  doHdr = false;  if (hdr == HDRBloomMode.Auto)  doHdr = source.format == RenderTextureFormat.ARGBHalf && GetComponent<Camera>().allowHDR;  else  {  doHdr = hdr == HDRBloomMode.On;  }  doHdr = doHdr && supportHDRTextures;  BloomScreenBlendMode realBlendMode = screenBlendMode;  if (doHdr)  realBlendMode = BloomScreenBlendMode.Add;  var rtFormat = (doHdr) ? RenderTextureFormat.ARGBHalf : RenderTextureFormat.Default;  RenderTexture halfRezColor = RenderTexture.GetTemporary(source.width / 2, source.height / 2, 0, rtFormat);  RenderTexture quarterRezColor = RenderTexture.GetTemporary(source.width / 4, source.height / 4, 0, rtFormat);  RenderTexture secondQuarterRezColor = RenderTexture.GetTemporary(source.width / 4, source.height / 4, 0, rtFormat);  RenderTexture thirdQuarterRezColor = RenderTexture.GetTemporary(source.width / 4, source.height / 4, 0, rtFormat);  float widthOverHeight = (1.0f \* source.width) / (1.0f \* source.height);  float oneOverBaseSize = 1.0f / 512.0f;  // downsample  Graphics.Blit(source, halfRezColor, screenBlend, 2); // <- 2 is stable downsample  Graphics.Blit(halfRezColor, quarterRezColor, screenBlend, 2); // <- 2 is stable downsample  RenderTexture.ReleaseTemporary(halfRezColor);  // cut colors (thresholding)  BrightFilter(bloomThreshold, useSrcAlphaAsMask, quarterRezColor, secondQuarterRezColor);  quarterRezColor.DiscardContents();  // blurring  if (bloomBlurIterations < 1) bloomBlurIterations = 1;  for (int iter = 0; iter < bloomBlurIterations; iter++)  {  float spreadForPass = (1.0f + (iter \* 0.5f)) \* sepBlurSpread;  separableBlurMaterial.SetVector("offsets", new Vector4(0.0f, spreadForPass \* oneOverBaseSize, 0.0f, 0.0f));  RenderTexture src = iter == 0 ? secondQuarterRezColor : quarterRezColor;  Graphics.Blit(src, thirdQuarterRezColor, separableBlurMaterial);  src.DiscardContents();  separableBlurMaterial.SetVector("offsets", new Vector4((spreadForPass / widthOverHeight) \* oneOverBaseSize, 0.0f, 0.0f, 0.0f));  Graphics.Blit(thirdQuarterRezColor, quarterRezColor, separableBlurMaterial);  thirdQuarterRezColor.DiscardContents();  }  // lens flares: ghosting, anamorphic or a combination  if (lensflares)  {  if (lensflareMode == 0)  {  BrightFilter(lensflareThreshold, 0.0f, quarterRezColor, thirdQuarterRezColor);  quarterRezColor.DiscardContents();  // smooth a little, this needs to be resolution dependent  /\*  separableBlurMaterial.SetVector ("offsets", Vector4 (0.0ff, (2.0ff) / (1.0ff \* quarterRezColor.height), 0.0ff, 0.0ff));  Graphics.Blit (thirdQuarterRezColor, secondQuarterRezColor, separableBlurMaterial);  separableBlurMaterial.SetVector ("offsets", Vector4 ((2.0ff) / (1.0ff \* quarterRezColor.width), 0.0ff, 0.0ff, 0.0ff));  Graphics.Blit (secondQuarterRezColor, thirdQuarterRezColor, separableBlurMaterial);  \*/  // no ugly edges!  Vignette(0.975f, thirdQuarterRezColor, secondQuarterRezColor);  thirdQuarterRezColor.DiscardContents();  BlendFlares(secondQuarterRezColor, quarterRezColor);  secondQuarterRezColor.DiscardContents();  }  // (b) hollywood/anamorphic flares?  else  {  // thirdQuarter has the brightcut unblurred colors  // quarterRezColor is the blurred, brightcut buffer that will end up as bloom  hollywoodFlaresMaterial.SetVector("\_threshold", new Vector4(lensflareThreshold, 1.0f / (1.0f - lensflareThreshold), 0.0f, 0.0f));  hollywoodFlaresMaterial.SetVector("tintColor", new Vector4(flareColorA.r, flareColorA.g, flareColorA.b, flareColorA.a) \* flareColorA.a \* lensflareIntensity);  Graphics.Blit(thirdQuarterRezColor, secondQuarterRezColor, hollywoodFlaresMaterial, 2);  thirdQuarterRezColor.DiscardContents();  Graphics.Blit(secondQuarterRezColor, thirdQuarterRezColor, hollywoodFlaresMaterial, 3);  secondQuarterRezColor.DiscardContents();  hollywoodFlaresMaterial.SetVector("offsets", new Vector4((sepBlurSpread \* 1.0f / widthOverHeight) \* oneOverBaseSize, 0.0f, 0.0f, 0.0f));  hollywoodFlaresMaterial.SetFloat("stretchWidth", hollyStretchWidth);  Graphics.Blit(thirdQuarterRezColor, secondQuarterRezColor, hollywoodFlaresMaterial, 1);  thirdQuarterRezColor.DiscardContents();  hollywoodFlaresMaterial.SetFloat("stretchWidth", hollyStretchWidth \* 2.0f);  Graphics.Blit(secondQuarterRezColor, thirdQuarterRezColor, hollywoodFlaresMaterial, 1);  secondQuarterRezColor.DiscardContents();  hollywoodFlaresMaterial.SetFloat("stretchWidth", hollyStretchWidth \* 4.0f);  Graphics.Blit(thirdQuarterRezColor, secondQuarterRezColor, hollywoodFlaresMaterial, 1);  thirdQuarterRezColor.DiscardContents();  if (lensflareMode == (LensflareStyle34)1)  {  for (int itera = 0; itera < hollywoodFlareBlurIterations; itera++)  {  separableBlurMaterial.SetVector("offsets", new Vector4((hollyStretchWidth \* 2.0f / widthOverHeight) \* oneOverBaseSize, 0.0f, 0.0f, 0.0f));  Graphics.Blit(secondQuarterRezColor, thirdQuarterRezColor, separableBlurMaterial);  secondQuarterRezColor.DiscardContents();  separableBlurMaterial.SetVector("offsets", new Vector4((hollyStretchWidth \* 2.0f / widthOverHeight) \* oneOverBaseSize, 0.0f, 0.0f, 0.0f));  Graphics.Blit(thirdQuarterRezColor, secondQuarterRezColor, separableBlurMaterial);  thirdQuarterRezColor.DiscardContents();  }  AddTo(1.0f, secondQuarterRezColor, quarterRezColor);  secondQuarterRezColor.DiscardContents();  }  else  {  // (c) combined  for (int ix = 0; ix < hollywoodFlareBlurIterations; ix++)  {  separableBlurMaterial.SetVector("offsets", new Vector4((hollyStretchWidth \* 2.0f / widthOverHeight) \* oneOverBaseSize, 0.0f, 0.0f, 0.0f));  Graphics.Blit(secondQuarterRezColor, thirdQuarterRezColor, separableBlurMaterial);  secondQuarterRezColor.DiscardContents();  separableBlurMaterial.SetVector("offsets", new Vector4((hollyStretchWidth \* 2.0f / widthOverHeight) \* oneOverBaseSize, 0.0f, 0.0f, 0.0f));  Graphics.Blit(thirdQuarterRezColor, secondQuarterRezColor, separableBlurMaterial);  thirdQuarterRezColor.DiscardContents();  }  Vignette(1.0f, secondQuarterRezColor, thirdQuarterRezColor);  secondQuarterRezColor.DiscardContents();  BlendFlares(thirdQuarterRezColor, secondQuarterRezColor);  thirdQuarterRezColor.DiscardContents();  AddTo(1.0f, secondQuarterRezColor, quarterRezColor);  secondQuarterRezColor.DiscardContents();  }  }  }  // screen blend bloom results to color buffer  screenBlend.SetFloat("\_Intensity", bloomIntensity);  screenBlend.SetTexture("\_ColorBuffer", source);  Graphics.Blit(quarterRezColor, destination, screenBlend, (int)realBlendMode);  RenderTexture.ReleaseTemporary(quarterRezColor);  RenderTexture.ReleaseTemporary(secondQuarterRezColor);  RenderTexture.ReleaseTemporary(thirdQuarterRezColor);  }  private void AddTo(float intensity\_, RenderTexture from, RenderTexture to)  {  addBrightStuffBlendOneOneMaterial.SetFloat("\_Intensity", intensity\_);  Graphics.Blit(from, to, addBrightStuffBlendOneOneMaterial);  }  private void BlendFlares(RenderTexture from, RenderTexture to)  {  lensFlareMaterial.SetVector("colorA", new Vector4(flareColorA.r, flareColorA.g, flareColorA.b, flareColorA.a) \* lensflareIntensity);  lensFlareMaterial.SetVector("colorB", new Vector4(flareColorB.r, flareColorB.g, flareColorB.b, flareColorB.a) \* lensflareIntensity);  lensFlareMaterial.SetVector("colorC", new Vector4(flareColorC.r, flareColorC.g, flareColorC.b, flareColorC.a) \* lensflareIntensity);  lensFlareMaterial.SetVector("colorD", new Vector4(flareColorD.r, flareColorD.g, flareColorD.b, flareColorD.a) \* lensflareIntensity);  Graphics.Blit(from, to, lensFlareMaterial);  }  private void BrightFilter(float thresh, float useAlphaAsMask, RenderTexture from, RenderTexture to)  {  if (doHdr)  brightPassFilterMaterial.SetVector("threshold", new Vector4(thresh, 1.0f, 0.0f, 0.0f));  else  brightPassFilterMaterial.SetVector("threshold", new Vector4(thresh, 1.0f / (1.0f - thresh), 0.0f, 0.0f));  brightPassFilterMaterial.SetFloat("useSrcAlphaAsMask", useAlphaAsMask);  Graphics.Blit(from, to, brightPassFilterMaterial);  }  private void Vignette(float amount, RenderTexture from, RenderTexture to)  {  if (lensFlareVignetteMask)  {  screenBlend.SetTexture("\_ColorBuffer", lensFlareVignetteMask);  Graphics.Blit(from, to, screenBlend, 3);  }  else  {  vignetteMaterial.SetFloat("vignetteIntensity", amount);  Graphics.Blit(from, to, vignetteMaterial);  }  }  }  } |
| using System;  using UnityEditor;  using UnityEngine;  namespace UnityStandardAssets.ImageEffects  {  [CustomEditor (typeof(BloomAndFlares))]  class BloomAndFlaresEditor : Editor  {  SerializedProperty tweakMode;  SerializedProperty screenBlendMode;  SerializedObject serObj;  SerializedProperty hdr;  SerializedProperty sepBlurSpread;  SerializedProperty useSrcAlphaAsMask;  SerializedProperty bloomIntensity;  SerializedProperty bloomthreshold;  SerializedProperty bloomBlurIterations;  SerializedProperty lensflares;  SerializedProperty hollywoodFlareBlurIterations;  SerializedProperty lensflareMode;  SerializedProperty hollyStretchWidth;  SerializedProperty lensflareIntensity;  SerializedProperty lensflarethreshold;  SerializedProperty flareColorA;  SerializedProperty flareColorB;  SerializedProperty flareColorC;  SerializedProperty flareColorD;  SerializedProperty lensFlareVignetteMask;  void OnEnable () {  serObj = new SerializedObject (target);  screenBlendMode = serObj.FindProperty("screenBlendMode");  hdr = serObj.FindProperty("hdr");  sepBlurSpread = serObj.FindProperty("sepBlurSpread");  useSrcAlphaAsMask = serObj.FindProperty("useSrcAlphaAsMask");  bloomIntensity = serObj.FindProperty("bloomIntensity");  bloomthreshold = serObj.FindProperty("bloomThreshold");  bloomBlurIterations = serObj.FindProperty("bloomBlurIterations");  lensflares = serObj.FindProperty("lensflares");  lensflareMode = serObj.FindProperty("lensflareMode");  hollywoodFlareBlurIterations = serObj.FindProperty("hollywoodFlareBlurIterations");  hollyStretchWidth = serObj.FindProperty("hollyStretchWidth");  lensflareIntensity = serObj.FindProperty("lensflareIntensity");  lensflarethreshold = serObj.FindProperty("lensflareThreshold");  flareColorA = serObj.FindProperty("flareColorA");  flareColorB = serObj.FindProperty("flareColorB");  flareColorC = serObj.FindProperty("flareColorC");  flareColorD = serObj.FindProperty("flareColorD");  lensFlareVignetteMask = serObj.FindProperty("lensFlareVignetteMask");  tweakMode = serObj.FindProperty("tweakMode");  }  public override void OnInspectorGUI () {  serObj.Update();  GUILayout.Label("HDR " + (hdr.enumValueIndex == 0 ? "auto detected, " : (hdr.enumValueIndex == 1 ? "forced on, " : "disabled, ")) + (useSrcAlphaAsMask.floatValue < 0.1f ? " ignoring alpha channel glow information" : " using alpha channel glow information"), EditorStyles.miniBoldLabel);  EditorGUILayout.PropertyField (tweakMode, new GUIContent("Tweak mode"));  EditorGUILayout.PropertyField (screenBlendMode, new GUIContent("Blend mode"));  EditorGUILayout.PropertyField (hdr, new GUIContent("HDR"));  // display info text when screen blend mode cannot be used  Camera cam = (target as BloomAndFlares).GetComponent<Camera>();  if (cam != null) {  if (screenBlendMode.enumValueIndex==0 && ((cam.allowHDR && hdr.enumValueIndex==0) || (hdr.enumValueIndex==1))) {  EditorGUILayout.HelpBox("Screen blend is not supported in HDR. Using 'Add' instead.", MessageType.Info);  }  }  if (1 == tweakMode.intValue)  EditorGUILayout.PropertyField (lensflares, new GUIContent("Cast lens flares"));  EditorGUILayout.Separator ();  EditorGUILayout.PropertyField (bloomIntensity, new GUIContent("Intensity"));  bloomthreshold.floatValue = EditorGUILayout.Slider ("threshold", bloomthreshold.floatValue, -0.05f, 4.0f);  bloomBlurIterations.intValue = EditorGUILayout.IntSlider ("Blur iterations", bloomBlurIterations.intValue, 1, 4);  sepBlurSpread.floatValue = EditorGUILayout.Slider ("Blur spread", sepBlurSpread.floatValue, 0.1f, 10.0f);  if (1 == tweakMode.intValue)  useSrcAlphaAsMask.floatValue = EditorGUILayout.Slider (new GUIContent("Use alpha mask", "Make alpha channel define glowiness"), useSrcAlphaAsMask.floatValue, 0.0f, 1.0f);  else  useSrcAlphaAsMask.floatValue = 0.0f;  if (1 == tweakMode.intValue) {  EditorGUILayout.Separator ();  if (lensflares.boolValue) {  // further lens flare tweakings  if (0 != tweakMode.intValue)  EditorGUILayout.PropertyField (lensflareMode, new GUIContent("Lens flare mode"));  else  lensflareMode.enumValueIndex = 0;  EditorGUILayout.PropertyField(lensFlareVignetteMask, new GUIContent("Lens flare mask", "This mask is needed to prevent lens flare artifacts"));  EditorGUILayout.PropertyField (lensflareIntensity, new GUIContent("Local intensity"));  lensflarethreshold.floatValue = EditorGUILayout.Slider ("Local threshold", lensflarethreshold.floatValue, 0.0f, 1.0f);  if (lensflareMode.intValue == 0) {  // ghosting  EditorGUILayout.BeginHorizontal ();  EditorGUILayout.PropertyField (flareColorA, new GUIContent("1st Color"));  EditorGUILayout.PropertyField (flareColorB, new GUIContent("2nd Color"));  EditorGUILayout.EndHorizontal ();  EditorGUILayout.BeginHorizontal ();  EditorGUILayout.PropertyField (flareColorC, new GUIContent("3rd Color"));  EditorGUILayout.PropertyField (flareColorD, new GUIContent("4th Color"));  EditorGUILayout.EndHorizontal ();  }  else if (lensflareMode.intValue == 1) {  // hollywood  EditorGUILayout.PropertyField (hollyStretchWidth, new GUIContent("Stretch width"));  hollywoodFlareBlurIterations.intValue = EditorGUILayout.IntSlider ("Blur iterations", hollywoodFlareBlurIterations.intValue, 1, 4);  EditorGUILayout.PropertyField (flareColorA, new GUIContent("Tint Color"));  }  else if (lensflareMode.intValue == 2) {  // both  EditorGUILayout.PropertyField (hollyStretchWidth, new GUIContent("Stretch width"));  hollywoodFlareBlurIterations.intValue = EditorGUILayout.IntSlider ("Blur iterations", hollywoodFlareBlurIterations.intValue, 1, 4);  EditorGUILayout.BeginHorizontal ();  EditorGUILayout.PropertyField (flareColorA, new GUIContent("1st Color"));  EditorGUILayout.PropertyField (flareColorB, new GUIContent("2nd Color"));  EditorGUILayout.EndHorizontal ();  EditorGUILayout.BeginHorizontal ();  EditorGUILayout.PropertyField (flareColorC, new GUIContent("3rd Color"));  EditorGUILayout.PropertyField (flareColorD, new GUIContent("4th Color"));  EditorGUILayout.EndHorizontal ();  }  }  } else  lensflares.boolValue = false; // disable lens flares in simple tweak mode  serObj.ApplyModifiedProperties();  }  }  } |
| using System;  using UnityEditor;  using UnityEngine;  namespace UnityStandardAssets.ImageEffects  {  [CustomEditor (typeof(Bloom))]  class BloomEditor : Editor  {  SerializedProperty tweakMode;  SerializedProperty screenBlendMode;  SerializedObject serObj;  SerializedProperty hdr;  SerializedProperty quality;  SerializedProperty sepBlurSpread;  SerializedProperty bloomIntensity;  SerializedProperty bloomThresholdColor;  SerializedProperty bloomThreshold;  SerializedProperty bloomBlurIterations;  SerializedProperty hollywoodFlareBlurIterations;  SerializedProperty lensflareMode;  SerializedProperty hollyStretchWidth;  SerializedProperty lensflareIntensity;  SerializedProperty flareRotation;  SerializedProperty lensFlareSaturation;  SerializedProperty lensflareThreshold;  SerializedProperty flareColorA;  SerializedProperty flareColorB;  SerializedProperty flareColorC;  SerializedProperty flareColorD;  SerializedProperty lensFlareVignetteMask;  void OnEnable () {  serObj = new SerializedObject (target);  screenBlendMode = serObj.FindProperty("screenBlendMode");  hdr = serObj.FindProperty("hdr");  quality = serObj.FindProperty("quality");  sepBlurSpread = serObj.FindProperty("sepBlurSpread");  bloomIntensity = serObj.FindProperty("bloomIntensity");  bloomThreshold = serObj.FindProperty("bloomThreshold");  bloomThresholdColor = serObj.FindProperty("bloomThresholdColor");  bloomBlurIterations = serObj.FindProperty("bloomBlurIterations");  lensflareMode = serObj.FindProperty("lensflareMode");  hollywoodFlareBlurIterations = serObj.FindProperty("hollywoodFlareBlurIterations");  hollyStretchWidth = serObj.FindProperty("hollyStretchWidth");  lensflareIntensity = serObj.FindProperty("lensflareIntensity");  lensflareThreshold = serObj.FindProperty("lensflareThreshold");  lensFlareSaturation = serObj.FindProperty("lensFlareSaturation");  flareRotation = serObj.FindProperty("flareRotation");  flareColorA = serObj.FindProperty("flareColorA");  flareColorB = serObj.FindProperty("flareColorB");  flareColorC = serObj.FindProperty("flareColorC");  flareColorD = serObj.FindProperty("flareColorD");  lensFlareVignetteMask = serObj.FindProperty("lensFlareVignetteMask");  tweakMode = serObj.FindProperty("tweakMode");  }  public override void OnInspectorGUI () {  serObj.Update();  EditorGUILayout.LabelField("Glow and Lens Flares for bright screen pixels", EditorStyles.miniLabel);  EditorGUILayout.PropertyField (quality, new GUIContent("Quality", "High quality preserves high frequencies with bigger blurs and uses a better blending and down-/upsampling"));  EditorGUILayout.Separator ();  EditorGUILayout.PropertyField (tweakMode, new GUIContent("Mode"));  EditorGUILayout.PropertyField (screenBlendMode, new GUIContent("Blend"));  EditorGUILayout.PropertyField (hdr, new GUIContent("HDR"));  EditorGUILayout.Separator ();  // display info text when screen blend mode cannot be used  Camera cam = (target as Bloom).GetComponent<Camera>();  if (cam != null) {  if (screenBlendMode.enumValueIndex==0 && ((cam.allowHDR && hdr.enumValueIndex==0) || (hdr.enumValueIndex==1))) {  EditorGUILayout.HelpBox("Screen blend is not supported in HDR. Using 'Add' instead.", MessageType.Info);  }  }  EditorGUILayout.PropertyField (bloomIntensity, new GUIContent("Intensity"));  bloomThreshold.floatValue = EditorGUILayout.Slider ("Threshold", bloomThreshold.floatValue, -0.05f, 4.0f);  if (1 == tweakMode.intValue) {  EditorGUILayout.PropertyField(bloomThresholdColor, new GUIContent(" RGB Threshold"));  }  EditorGUILayout.Separator ();  bloomBlurIterations.intValue = EditorGUILayout.IntSlider ("Blur Iterations", bloomBlurIterations.intValue, 1, 4);  sepBlurSpread.floatValue = EditorGUILayout.Slider (" Sample Distance", sepBlurSpread.floatValue, 0.1f, 10.0f);  EditorGUILayout.Separator ();  if (1 == tweakMode.intValue) {  // further lens flare tweakings  if (0 != tweakMode.intValue)  EditorGUILayout.PropertyField (lensflareMode, new GUIContent("Lens Flares"));  else  lensflareMode.enumValueIndex = 0;  EditorGUILayout.PropertyField (lensflareIntensity, new GUIContent(" Local Intensity", "0 disables lens flares entirely (optimization)"));  lensflareThreshold.floatValue = EditorGUILayout.Slider ("Threshold", lensflareThreshold.floatValue, 0.0f, 4.0f);  if (Mathf.Abs(lensflareIntensity.floatValue) > Mathf.Epsilon) {  if (lensflareMode.intValue == 0) {  // ghosting  EditorGUILayout.BeginHorizontal ();  EditorGUILayout.PropertyField (flareColorA, new GUIContent(" 1st Color"));  EditorGUILayout.PropertyField (flareColorB, new GUIContent(" 2nd Color"));  EditorGUILayout.EndHorizontal ();  EditorGUILayout.BeginHorizontal ();  EditorGUILayout.PropertyField (flareColorC, new GUIContent(" 3rd Color"));  EditorGUILayout.PropertyField (flareColorD, new GUIContent(" 4th Color"));  EditorGUILayout.EndHorizontal ();  }  else if (lensflareMode.intValue == 1) {  // hollywood  EditorGUILayout.PropertyField (hollyStretchWidth, new GUIContent(" Stretch width"));  EditorGUILayout.PropertyField (flareRotation, new GUIContent( " Rotation"));  hollywoodFlareBlurIterations.intValue = EditorGUILayout.IntSlider (" Blur Iterations", hollywoodFlareBlurIterations.intValue, 1, 4);  EditorGUILayout.PropertyField (lensFlareSaturation, new GUIContent(" Saturation"));  EditorGUILayout.PropertyField (flareColorA, new GUIContent(" Tint Color"));  }  else if (lensflareMode.intValue == 2) {  // both  EditorGUILayout.PropertyField (hollyStretchWidth, new GUIContent(" Stretch width"));  hollywoodFlareBlurIterations.intValue = EditorGUILayout.IntSlider (" Blur Iterations", hollywoodFlareBlurIterations.intValue, 1, 4);  EditorGUILayout.PropertyField (lensFlareSaturation, new GUIContent(" Saturation"));  EditorGUILayout.BeginHorizontal ();  EditorGUILayout.PropertyField (flareColorA, new GUIContent(" 1st Color"));  EditorGUILayout.PropertyField (flareColorB, new GUIContent(" 2nd Color"));  EditorGUILayout.EndHorizontal ();  EditorGUILayout.BeginHorizontal ();  EditorGUILayout.PropertyField (flareColorC, new GUIContent(" 3rd Color"));  EditorGUILayout.PropertyField (flareColorD, new GUIContent(" 4th Color"));  EditorGUILayout.EndHorizontal ();  }  EditorGUILayout.PropertyField(lensFlareVignetteMask, new GUIContent(" Mask", "This mask is needed to prevent lens flare artifacts"));  }  }  serObj.ApplyModifiedProperties();  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  namespace BNG {  public class BoneMapping : MonoBehaviour {    [Range(0f, 1f)] public float weight = 1f;  [System.Serializable]  public enum Mode {  FromToRotation,  MatchRotation  }  public BoneObject[] Fingers;  [Header("Shown for Debug : ")]  public bool ShowGizmos = true;  void Update() {  if (weight <= 0f) {  return;  }    for (int x = 0; x < Fingers.Length; x++) {  BoneObject finger = Fingers[x];  if (finger == null) {  continue;  }  for (int i = 0; i < finger.destinationBones.Length - 1; i++) {  // Get the relative rotation from the current rotation to the target rotation  Quaternion f = Quaternion.Inverse(finger.destinationBones[i].rotation) \* finger.targetBones[i].rotation;  // Weight blending  if (weight < 1f) {  f = Quaternion.Slerp(Quaternion.identity, f, weight);  }  // Append relative rotation  finger.destinationBones[i].rotation \*= f;  }  }  }  void OnDrawGizmos() {  if (!ShowGizmos || Fingers == null) {  return;  }  for (int x = 0; x < Fingers.Length; x++) {  BoneObject finger = Fingers[x];  for (int i = 0; i < finger.targetBones.Length; i++) {  if(finger.targetBones[i] == null) {  continue;  }  // Target Bones  Gizmos.color = Color.red;  Gizmos.DrawSphere(finger.targetBones[i].position, 0.003f);  if (i < finger.targetBones.Length - 1) {  if(finger.targetBones[i] == null || finger.targetBones[i + 1] == null) {  continue;  }  Gizmos.DrawLine(finger.targetBones[i].position, finger.targetBones[i + 1].position);  }  }  for (int i = 0; i < finger.destinationBones.Length; i++) {  // Reference may have been removed from inspector  if (finger.destinationBones[i] == null) {  continue;  }  // Avatar Bones  Gizmos.color = Color.yellow;  Gizmos.DrawSphere(finger.destinationBones[i].position, 0.003f);  if (i < finger.destinationBones.Length - 1) {  if (finger.destinationBones[i] == null || finger.destinationBones[i + 1] == null) {  continue;  }  Gizmos.DrawLine(finger.destinationBones[i].position, finger.destinationBones[i + 1].position);  }  }  }  }  }  [System.Serializable]  public class BoneObject {  public Transform[] targetBones = new Transform[0];  public Transform[] destinationBones = new Transform[0];  public Vector3 avatarForwardAxis = Vector3.forward;  public Vector3 avatarUpAxis = Vector3.up;  public Vector3 targetForwardAxis = Vector3.forward;  public Vector3 targetUpAxis = Vector3.up;  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.UI;  namespace BNG {  /// <summary>  /// Add this Component to any Canvas to make sure it can be interacted with in World Space  /// </summary>  [RequireComponent(typeof(GraphicRaycaster))]  [RequireComponent(typeof(Canvas))]  public class VRCanvas : MonoBehaviour {  void Start() {  VRUISystem.Instance.AddCanvas(GetComponent<Canvas>());  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  namespace BNG {  public class VRIFGrabpointUpdater : MonoBehaviour {  [Header("Right Hand Model")]  [Tooltip("This is the local position of the hand model that was defined in your previous xr rig, as well as what was used in the previewer.")]  public Vector3 PriorModelOffsetRightPosition = new Vector3(-0.024f, 0.051f, 0.001f);  [Tooltip("This is the local position of the NEW hand model that is currently defined in your xr rig, as well as what is used in the previewer.")]  public Vector3 NewModelOffsetRightPosition = new Vector3(-0.006f, 0, -0.04f);  [Tooltip("This is the local rotation of the hand model that was defined in your previous xr rig, as well as what was used in the previewer.")]  public Vector3 PriorModelOffsetRightRotation = new Vector3(-12.041f, 13f, -90f);  [Tooltip("This is the local rotation of the NEW hand model that is currently defined in your xr rig, as well as what is used in the previewer.")]  public Vector3 NewModelOffsetRightRotation = new Vector3(-6, 0.43f, -90f);  [Header("Left Hand Model")]  [Tooltip("This is the local position of the hand model that was defined in your previous xr rig, as well as what was used in the previewer.")]  public Vector3 PriorModelOffsetLeftPosition = new Vector3(0.024f, 0.051f, 0.001f);  [Tooltip("This is the local position of the NEW hand model that is currently defined in your xr rig, as well as what is used in the previewer.")]  public Vector3 NewModelOffsetLeftPosition = new Vector3(0.006f, 0, -0.04f);  [Tooltip("This is the local rotation of the hand model that was defined in your previous xr rig, as well as what was used in the previewer.")]  public Vector3 PriorModelOffsetLeftRotation = new Vector3(-12.041f, -13f, 90f);  [Tooltip("This is the local rotation of the NEW hand model that is currently defined in your xr rig, as well as what is used in the previewer.")]  public Vector3 NewModelOffsetLeftRotation = new Vector3(-6, -0.43f, 90);  void Start() {  ApplyGrabPointUpdate();  }  public void ApplyGrabPointUpdate() {  GrabPoint[] points = GetComponentsInChildren<GrabPoint>();  foreach(var gp in points) {  // Both Hands - use Right Offset for both  if (gp.RightHandIsValid && gp.LeftHandIsValid) {  gp.transform.localPosition = gp.transform.localPosition + (PriorModelOffsetRightPosition - NewModelOffsetRightPosition);  gp.transform.localRotation \*= Quaternion.Euler(PriorModelOffsetRightRotation) \* Quaternion.Inverse(Quaternion.Euler(NewModelOffsetRightRotation));  }  // Right Hand only  else if (gp.RightHandIsValid) {  gp.transform.localPosition = gp.transform.localPosition + (PriorModelOffsetRightPosition - NewModelOffsetRightPosition);  gp.transform.localRotation \*= Quaternion.Euler(PriorModelOffsetRightRotation) \* Quaternion.Inverse(Quaternion.Euler(NewModelOffsetRightRotation));  }  // Left Hand only  else if(gp.LeftHandIsValid) {  gp.transform.localPosition = gp.transform.localPosition + (PriorModelOffsetLeftPosition - NewModelOffsetLeftPosition);  gp.transform.localRotation \*= Quaternion.Euler(PriorModelOffsetLeftRotation) \* Quaternion.Inverse(Quaternion.Euler(NewModelOffsetLeftRotation));  }  }  }  }  } |
| using System;  using System.Collections;  using System.Collections.Generic;  using UnityEditor;  using UnityEngine;  namespace BNG {  [CustomEditor(typeof(VRIFGrabpointUpdater))]  [CanEditMultipleObjects]  public class VRIFGrabpointUpdaterEditor : Editor {  VRIFGrabpointUpdater updater;  public override void OnInspectorGUI() {  updater = (VRIFGrabpointUpdater)target;  EditorGUILayout.HelpBox("This component will help you autmatically update your grab points, as the default hand model positions have changed since VRIF v1.7. Click the 'Updated Pose' button to update the grab points in the editor. You can preview the changes in play mode before applying the changes in the editor.", MessageType.Info);  if (GUILayout.Button("Update Pose")) {  updater.ApplyGrabPointUpdate();  }  EditorGUILayout.Separator();  serializedObject.ApplyModifiedProperties();  base.OnInspectorGUI();  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using System.Linq;  using UnityEngine;  namespace BNG {  public class VRKeyboard : MonoBehaviour {  public UnityEngine.UI.InputField AttachedInputField;  public bool UseShift = false;  [Header("Sound FX")]  public AudioClip KeyPressSound;  List<VRKeyboardKey> KeyboardKeys;  void Awake() {  KeyboardKeys = transform.GetComponentsInChildren<VRKeyboardKey>().ToList();  }  public void PressKey(string key) {  if(AttachedInputField != null) {  UpdateInputField(key);  }  else {  Debug.Log("Pressed Key : " + key);  }  }  public void UpdateInputField(string key) {  string currentText = AttachedInputField.text;  int caretPosition = AttachedInputField.caretPosition;  int textLength = currentText.Length;  bool caretAtEnd = AttachedInputField.isFocused == false || caretPosition == textLength;  // Formatted key based on short names  string formattedKey = key;  if (key.ToLower() == "space") {  formattedKey = " ";  }  // Find KeyCode Sequence  if (formattedKey.ToLower() == "backspace") {  // At beginning of line - nothing to back into  if(caretPosition == 0) {  PlayClickSound(); // Still play the click sound  return;  }  currentText = currentText.Remove(caretPosition - 1, 1);  if(!caretAtEnd) {  MoveCaretBack();  }  }  else if (formattedKey.ToLower() == "enter") {  // Debug.Log("Pressed Enter");  // UnityEngine.EventSystems.ExecuteEvents.Execute(AttachedInputField.gameObject, null, UnityEngine.EventSystems.ExecuteEvents.submitHandler);  }  else if (formattedKey.ToLower() == "shift") {  ToggleShift();  }  else {  // Simply append the text to the end  if(caretAtEnd) {  currentText += formattedKey;  MoveCaretUp();  }  else {  // Otherwise we need to figure out how to insert the text and where  string preText = "";  if (caretPosition > 0) {  preText = currentText.Substring(0, caretPosition);  }  MoveCaretUp();  string postText = currentText.Substring(caretPosition, textLength - preText.Length);  currentText = preText + formattedKey + postText;  }  }  // Apply the text change  AttachedInputField.text = currentText;  PlayClickSound();  // Keep Input Focused  if (!AttachedInputField.isFocused) {  AttachedInputField.Select();  }  }  public virtual void PlayClickSound() {  if(KeyPressSound != null) {  VRUtils.Instance.PlaySpatialClipAt(KeyPressSound, transform.position, 1f, 0.5f);  }  }  public void MoveCaretUp() {  StartCoroutine(IncreaseInputFieldCareteRoutine());  }  public void MoveCaretBack() {  StartCoroutine(DecreaseInputFieldCareteRoutine());  }  public void ToggleShift() {  UseShift = !UseShift;  foreach(var key in KeyboardKeys) {  if(key != null) {  key.ToggleShift();  }  }  }  IEnumerator IncreaseInputFieldCareteRoutine() {  yield return new WaitForEndOfFrame();  AttachedInputField.caretPosition = AttachedInputField.caretPosition + 1;  AttachedInputField.ForceLabelUpdate();  }  IEnumerator DecreaseInputFieldCareteRoutine() {  yield return new WaitForEndOfFrame();  AttachedInputField.caretPosition = AttachedInputField.caretPosition - 1;  AttachedInputField.ForceLabelUpdate();  }  public void AttachToInputField(UnityEngine.UI.InputField inputField) {  AttachedInputField = inputField;  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  namespace BNG {  public class VRKeyboardKey : MonoBehaviour {  UnityEngine.UI.Button thisButton;  UnityEngine.UI.Text thisButtonText;  VRKeyboard vrKeyboard;  public string Keycode;  public string KeycodeShift;  [HideInInspector]  public bool UseShiftKey = false;  void Awake() {  thisButton = GetComponent<UnityEngine.UI.Button>();  thisButtonText = GetComponentInChildren<UnityEngine.UI.Text>();  // Assign click event handler  if (thisButton != null) {  thisButton.onClick.AddListener(OnKeyHit);  }  vrKeyboard = GetComponentInParent<VRKeyboard>();  }  public virtual void ToggleShift() {  UseShiftKey = !UseShiftKey;  // Make sure the button exists  if(thisButtonText == null) {  return;  }  // Update text label  if(UseShiftKey && !string.IsNullOrEmpty(KeycodeShift)) {  thisButtonText.text = KeycodeShift;  }  else {  thisButtonText.text = Keycode;  }  }  public virtual void OnKeyHit() {  OnKeyHit(UseShiftKey && !string.IsNullOrEmpty(KeycodeShift) ? KeycodeShift : Keycode);  }  public virtual void OnKeyHit(string key) {  if(vrKeyboard != null) {  vrKeyboard.PressKey(key);  }  else {  Debug.Log("Pressed key " + key + ", but no keyboard was found");  }  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  namespace BNG {  public class VRKeyboardKeyCollider : MonoBehaviour {  public float PressedInZValue = 0.01f;  public float PressInSpeed = 15f;  UnityEngine.UI.Button uiButton;  int itemsInTrigger = 0;  bool wasSelected = false;  void Awake() {  uiButton = GetComponentInParent<UnityEngine.UI.Button>();  }  void Update() {  if (itemsInTrigger > 0) {  transform.parent.localPosition = Vector3.Lerp(transform.parent.localPosition, new Vector3(transform.parent.localPosition.x, transform.parent.localPosition.y, PressedInZValue), Time.deltaTime \* PressInSpeed);  if(!wasSelected) {  uiButton.onClick.Invoke();  // uiButton.Select();  wasSelected = true;  }  }  else {  transform.parent.localPosition = Vector3.Lerp(transform.parent.localPosition, new Vector3(transform.parent.localPosition.x, transform.parent.localPosition.y, 0), Time.deltaTime \* PressInSpeed);  wasSelected = false;  }  }  void OnTriggerEnter(Collider other) {  //if (other.GetComponent<UITrigger>() != null || other.GetComponent<Grabber>() != null) {  if (other.GetComponent<UITrigger>() != null) {  itemsInTrigger++;  }  }  void OnTriggerExit(Collider other) {  if (other.GetComponent<UITrigger>() != null) {  itemsInTrigger--;  }  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  namespace BNG {  public class VRTextInput : MonoBehaviour {  UnityEngine.UI.InputField thisInputField;  public bool AttachToVRKeyboard = true;  public bool ActivateKeyboardOnSelect = true;  public bool DeactivateKeyboardOnDeselect = false;  public VRKeyboard AttachedKeyboard;  bool isFocused, wasFocused = false;  void Awake() {  thisInputField = GetComponent<UnityEngine.UI.InputField>();    if(thisInputField) {  AttachedKeyboard.AttachToInputField(thisInputField);  }  }  void Update() {  isFocused = thisInputField != null && thisInputField.isFocused;  // Check if our input field is now focused  if(isFocused == true && wasFocused == false) {  OnInputSelect();  }  else if (isFocused == false && wasFocused == true) {  OnInputDeselect();  }  wasFocused = isFocused;  }  public void OnInputSelect() {  // Enable keyboard if disabled  if (ActivateKeyboardOnSelect && AttachedKeyboard != null && !AttachedKeyboard.gameObject.activeInHierarchy) {  AttachedKeyboard.gameObject.SetActive(true);  AttachedKeyboard.AttachToInputField(thisInputField);  }  }  public void OnInputDeselect() {  if (DeactivateKeyboardOnDeselect && AttachedKeyboard != null && AttachedKeyboard.gameObject.activeInHierarchy) {  AttachedKeyboard.gameObject.SetActive(false);  }  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.EventSystems;  using UnityEngine.InputSystem;  using UnityEngine.UI;  namespace BNG {  public class VRUISystem : BaseInputModule {  [Header("XR Controller Options : ")]  [Tooltip("This setting determines if LeftPointerTransform or RightPointerTransform will be used as a forward vector for World Space UI events")]  public ControllerHand SelectedHand = ControllerHand.Right;  [Tooltip("A transform on the left controller to use when raycasting for world space UI events")]  public Transform LeftPointerTransform;  [Tooltip("A transform on the right controller to use when raycasting for world space UI events")]  public Transform RightPointerTransform;  [Tooltip("Controller Binding to use for input down, up, etc.")]  public List<ControllerBinding> ControllerInput = new List<ControllerBinding>() { ControllerBinding.RightTrigger };  [Tooltip("Unity Input Action used to simulate a click or touch event")]  public InputActionReference UIInputAction;  [Tooltip("If true a PhysicsRaycaster component will be added to the UI camera, allowing physical objects to use IPointer events such as OnPointClick, OnPointEnter, etc.")]  public bool AddPhysicsRaycaster = false;  public LayerMask PhysicsRaycasterEventMask;  [Tooltip("If true the Right Thumbstick will send scroll events to the UI")]  public bool RightThumbstickScroll = true;  [Header("Shown for Debug : ")]  public GameObject PressingObject;  public GameObject DraggingObject;  public GameObject ReleasingObject;  public PointerEventData EventData { get; private set; }  Camera cameraCaster;  private GameObject \_initialPressObject;  private bool \_lastInputDown;  bool inputDown;  private static VRUISystem \_instance;  public static VRUISystem Instance {  get {  if (\_instance == null) {  \_instance = GameObject.FindObjectOfType<VRUISystem>();  if (\_instance == null) {  // Check for existing event system  EventSystem eventSystem = EventSystem.current;  if(eventSystem == null) {  eventSystem = new GameObject("EventSystem").AddComponent<EventSystem>(); ;  }  \_instance = eventSystem.gameObject.AddComponent<VRUISystem>();  }  }  return \_instance;  }  }  protected override void Awake() {  UpdateControllerHand(SelectedHand);  EventData = new PointerEventData(eventSystem);  EventData.position = new Vector2(cameraCaster.pixelWidth / 2, cameraCaster.pixelHeight / 2);  AssignCameraToAllCanvases(cameraCaster);  }  void init() {  if(cameraCaster == null) {  // Create the camera required for the caster.  // We can reduce the fov and disable the camera component for performance  var go = new GameObject("CameraCaster");  cameraCaster = go.AddComponent<Camera>();  cameraCaster.stereoTargetEye = StereoTargetEyeMask.None;  cameraCaster.fieldOfView = 5f;  cameraCaster.nearClipPlane = 0.01f;  cameraCaster.clearFlags = CameraClearFlags.Nothing;  cameraCaster.enabled = false;  // Add PhysicsRaycaster so other objects can subscribe to IPointer events  if(AddPhysicsRaycaster) {  var pr = go.AddComponent<PhysicsRaycaster>();  pr.eventMask = PhysicsRaycasterEventMask;  }  }  }  public override void Process() {  // Input isn't ready if this Camera Caster's gameObject isn't active  if (EventData == null || !CameraCasterReady()) {  return;  }  EventData.position = new Vector2(cameraCaster.pixelWidth / 2, cameraCaster.pixelHeight / 2);  eventSystem.RaycastAll(EventData, m\_RaycastResultCache);  EventData.pointerCurrentRaycast = FindFirstRaycast(m\_RaycastResultCache);  m\_RaycastResultCache.Clear();  // Handle Hover  HandlePointerExitAndEnter(EventData, EventData.pointerCurrentRaycast.gameObject);  // Handle Drag  ExecuteEvents.Execute(EventData.pointerDrag, EventData, ExecuteEvents.dragHandler);  // Handle scroll  if(RightThumbstickScroll) {  EventData.scrollDelta = InputBridge.Instance.RightThumbstickAxis;  if (!Mathf.Approximately(EventData.scrollDelta.sqrMagnitude, 0)) {  ExecuteEvents.Execute(ExecuteEvents.GetEventHandler<IScrollHandler>(EventData.pointerCurrentRaycast.gameObject), EventData, ExecuteEvents.scrollHandler);  }  }    // Press Events  inputDown = InputReady();  // On Trigger Down > TriggerDownValue this frame but not last  if (inputDown && \_lastInputDown == false) {  PressDown();  }  // On Held Down  else if(inputDown) {  Press();  }  // On Release  else {  Release();  }  \_lastInputDown = inputDown;  }  public virtual bool InputReady() {  // Input isn't ready if this Camera Caster's gameObject isn't active  if(!CameraCasterReady()) {  return false;  }  // Check Unity Action  if (UIInputAction != null && UIInputAction.action.ReadValue<float>() == 1f) {  return true;  }  // Check for bound controller button  for (int x = 0; x < ControllerInput.Count; x++) {  if (InputBridge.Instance.GetControllerBindingValue(ControllerInput[x])) {  return true;  }  }  return false;  }  public virtual bool CameraCasterReady() {  if (cameraCaster != null && !cameraCaster.gameObject.activeInHierarchy) {  return false;  }  return true;  }  public virtual void PressDown() {  EventData.pointerPressRaycast = EventData.pointerCurrentRaycast;  // Deselect if selection changed  if(\_initialPressObject != null) {  // ExecuteEvents.Execute(\_initialPressObject, EventData, ExecuteEvents.deselectHandler);  \_initialPressObject = null;  }  \_initialPressObject = ExecuteEvents.GetEventHandler<IPointerClickHandler>(EventData.pointerPressRaycast.gameObject);  // Set Press Objects and Events  SetPressingObject(\_initialPressObject);  ExecuteEvents.Execute(EventData.pointerPress, EventData, ExecuteEvents.pointerDownHandler);  // Set Drag Objects and Events  SetDraggingObject(ExecuteEvents.GetEventHandler<IDragHandler>(EventData.pointerPressRaycast.gameObject));  ExecuteEvents.Execute(EventData.pointerDrag, EventData, ExecuteEvents.beginDragHandler);  }  public virtual void Press() {  EventData.pointerPressRaycast = EventData.pointerCurrentRaycast;  // Set Press Objects and Events  SetPressingObject(ExecuteEvents.GetEventHandler<IPointerClickHandler>(EventData.pointerPressRaycast.gameObject));  ExecuteEvents.Execute(EventData.pointerPress, EventData, ExecuteEvents.pointerDownHandler);  // Set Drag Objects and Events  SetDraggingObject(ExecuteEvents.GetEventHandler<IDragHandler>(EventData.pointerPressRaycast.gameObject));  ExecuteEvents.Execute(EventData.pointerDrag, EventData, ExecuteEvents.beginDragHandler);  }  public virtual void Release() {  SetReleasingObject(ExecuteEvents.GetEventHandler<IPointerClickHandler>(EventData.pointerCurrentRaycast.gameObject));  // Considered a click event if released after an initial click  if (EventData.pointerPress == ReleasingObject) {  ExecuteEvents.Execute(EventData.pointerPress, EventData, ExecuteEvents.pointerClickHandler);  }  ExecuteEvents.Execute(EventData.pointerPress, EventData, ExecuteEvents.pointerUpHandler);  ExecuteEvents.Execute(EventData.pointerDrag, EventData, ExecuteEvents.endDragHandler);  // Send deselect to  // ExecuteEvents.Execute(ReleasingObject, EventData, ExecuteEvents.deselectHandler);  ClearAll();  }  public virtual void ClearAll() {  SetPressingObject(null);  SetDraggingObject(null);  EventData.pointerCurrentRaycast.Clear();  }  public virtual void SetPressingObject(GameObject pressing) {  EventData.pointerPress = pressing;  PressingObject = pressing;  }  public virtual void SetDraggingObject(GameObject dragging) {  EventData.pointerDrag = dragging;  DraggingObject = dragging;  }  public virtual void SetReleasingObject(GameObject releasing) {  ReleasingObject = releasing;  }  public virtual void AssignCameraToAllCanvases(Camera cam) {  Canvas[] allCanvas = FindObjectsOfType<Canvas>();  for (int x = 0; x < allCanvas.Length; x++) {  AddCanvasToCamera(allCanvas[x], cam);  }  }  public virtual void AddCanvas(Canvas canvas) {  AddCanvasToCamera(canvas, cameraCaster);  }  public virtual void AddCanvasToCamera(Canvas canvas, Camera cam) {  canvas.worldCamera = cam;  }  public virtual void UpdateControllerHand(ControllerHand hand) {    // Make sure variables exist  init();  // Setup the Transform  if (hand == ControllerHand.Left && LeftPointerTransform != null) {  cameraCaster.transform.parent = LeftPointerTransform;  cameraCaster.transform.localPosition = Vector3.zero;  cameraCaster.transform.localEulerAngles = Vector3.zero;  }  else if (hand == ControllerHand.Right && RightPointerTransform != null) {  cameraCaster.transform.parent = RightPointerTransform;  cameraCaster.transform.localPosition = Vector3.zero;  cameraCaster.transform.localEulerAngles = Vector3.zero;  }  }  }  } |
| using System;  using UnityEngine;  namespace UnityStandardAssets.ImageEffects  {  [ExecuteInEditMode]  [RequireComponent (typeof(Camera))]  [AddComponentMenu ("Image Effects/Rendering/Sun Shafts")]  public class SunShafts : PostEffectsBase  {  public enum SunShaftsResolution  {  Low = 0,  Normal = 1,  High = 2,  }  public enum ShaftsScreenBlendMode  {  Screen = 0,  Add = 1,  }  public SunShaftsResolution resolution = SunShaftsResolution.Normal;  public ShaftsScreenBlendMode screenBlendMode = ShaftsScreenBlendMode.Screen;  public Transform sunTransform;  public int radialBlurIterations = 2;  public Color sunColor = Color.white;  public Color sunThreshold = new Color(0.87f,0.74f,0.65f);  public float sunShaftBlurRadius = 2.5f;  public float sunShaftIntensity = 1.15f;  public float maxRadius = 0.75f;  public bool useDepthTexture = true;  public Shader sunShaftsShader;  private Material sunShaftsMaterial;  public Shader simpleClearShader;  private Material simpleClearMaterial;  public override bool CheckResources () {  CheckSupport (useDepthTexture);  sunShaftsMaterial = CheckShaderAndCreateMaterial (sunShaftsShader, sunShaftsMaterial);  simpleClearMaterial = CheckShaderAndCreateMaterial (simpleClearShader, simpleClearMaterial);  if (!isSupported)  ReportAutoDisable ();  return isSupported;  }  void OnRenderImage (RenderTexture source, RenderTexture destination) {  if (CheckResources()==false) {  Graphics.Blit (source, destination);  return;  }  // we actually need to check this every frame  if (useDepthTexture)  GetComponent<Camera>().depthTextureMode |= DepthTextureMode.Depth;  int divider = 4;  if (resolution == SunShaftsResolution.Normal)  divider = 2;  else if (resolution == SunShaftsResolution.High)  divider = 1;  Vector3 v = Vector3.one \* 0.5f;  if (sunTransform)  v = GetComponent<Camera>().WorldToViewportPoint (sunTransform.position);  else  v = new Vector3(0.5f, 0.5f, 0.0f);  int rtW = source.width / divider;  int rtH = source.height / divider;  RenderTexture lrColorB;  RenderTexture lrDepthBuffer = RenderTexture.GetTemporary (rtW, rtH, 0);  // mask out everything except the skybox  // we have 2 methods, one of which requires depth buffer support, the other one is just comparing images  sunShaftsMaterial.SetVector ("\_BlurRadius4", new Vector4 (1.0f, 1.0f, 0.0f, 0.0f) \* sunShaftBlurRadius );  sunShaftsMaterial.SetVector ("\_SunPosition", new Vector4 (v.x, v.y, v.z, maxRadius));  sunShaftsMaterial.SetVector ("\_SunThreshold", sunThreshold);  if (!useDepthTexture) {  var format= GetComponent<Camera>().allowHDR ? RenderTextureFormat.DefaultHDR: RenderTextureFormat.Default;  RenderTexture tmpBuffer = RenderTexture.GetTemporary (source.width, source.height, 0, format);  RenderTexture.active = tmpBuffer;  GL.ClearWithSkybox (false, GetComponent<Camera>());  sunShaftsMaterial.SetTexture ("\_Skybox", tmpBuffer);  Graphics.Blit (source, lrDepthBuffer, sunShaftsMaterial, 3);  RenderTexture.ReleaseTemporary (tmpBuffer);  }  else {  Graphics.Blit (source, lrDepthBuffer, sunShaftsMaterial, 2);  }  // paint a small black small border to get rid of clamping problems  DrawBorder (lrDepthBuffer, simpleClearMaterial);  // radial blur:  radialBlurIterations = Mathf.Clamp (radialBlurIterations, 1, 4);  float ofs = sunShaftBlurRadius \* (1.0f / 768.0f);  sunShaftsMaterial.SetVector ("\_BlurRadius4", new Vector4 (ofs, ofs, 0.0f, 0.0f));  sunShaftsMaterial.SetVector ("\_SunPosition", new Vector4 (v.x, v.y, v.z, maxRadius));  for (int it2 = 0; it2 < radialBlurIterations; it2++ ) {  // each iteration takes 2 \* 6 samples  // we update \_BlurRadius each time to cheaply get a very smooth look  lrColorB = RenderTexture.GetTemporary (rtW, rtH, 0);  Graphics.Blit (lrDepthBuffer, lrColorB, sunShaftsMaterial, 1);  RenderTexture.ReleaseTemporary (lrDepthBuffer);  ofs = sunShaftBlurRadius \* (((it2 \* 2.0f + 1.0f) \* 6.0f)) / 768.0f;  sunShaftsMaterial.SetVector ("\_BlurRadius4", new Vector4 (ofs, ofs, 0.0f, 0.0f) );  lrDepthBuffer = RenderTexture.GetTemporary (rtW, rtH, 0);  Graphics.Blit (lrColorB, lrDepthBuffer, sunShaftsMaterial, 1);  RenderTexture.ReleaseTemporary (lrColorB);  ofs = sunShaftBlurRadius \* (((it2 \* 2.0f + 2.0f) \* 6.0f)) / 768.0f;  sunShaftsMaterial.SetVector ("\_BlurRadius4", new Vector4 (ofs, ofs, 0.0f, 0.0f) );  }  // put together:  if (v.z >= 0.0f)  sunShaftsMaterial.SetVector ("\_SunColor", new Vector4 (sunColor.r, sunColor.g, sunColor.b, sunColor.a) \* sunShaftIntensity);  else  sunShaftsMaterial.SetVector ("\_SunColor", Vector4.zero); // no backprojection !  sunShaftsMaterial.SetTexture ("\_ColorBuffer", lrDepthBuffer);  Graphics.Blit (source, destination, sunShaftsMaterial, (screenBlendMode == ShaftsScreenBlendMode.Screen) ? 0 : 4);  RenderTexture.ReleaseTemporary (lrDepthBuffer);  }  }  } |
| using System;  using UnityEditor;  using UnityEngine;  namespace UnityStandardAssets.ImageEffects  {  [CustomEditor (typeof(SunShafts))]  class SunShaftsEditor : Editor  {  SerializedObject serObj;  SerializedProperty sunTransform;  SerializedProperty radialBlurIterations;  SerializedProperty sunColor;  SerializedProperty sunThreshold;  SerializedProperty sunShaftBlurRadius;  SerializedProperty sunShaftIntensity;  SerializedProperty useDepthTexture;  SerializedProperty resolution;  SerializedProperty screenBlendMode;  SerializedProperty maxRadius;  void OnEnable () {  serObj = new SerializedObject (target);  screenBlendMode = serObj.FindProperty("screenBlendMode");  sunTransform = serObj.FindProperty("sunTransform");  sunColor = serObj.FindProperty("sunColor");  sunThreshold = serObj.FindProperty("sunThreshold");  sunShaftBlurRadius = serObj.FindProperty("sunShaftBlurRadius");  radialBlurIterations = serObj.FindProperty("radialBlurIterations");  sunShaftIntensity = serObj.FindProperty("sunShaftIntensity");  resolution = serObj.FindProperty("resolution");  maxRadius = serObj.FindProperty("maxRadius");  useDepthTexture = serObj.FindProperty("useDepthTexture");  }  public override void OnInspectorGUI () {  serObj.Update ();  EditorGUILayout.BeginHorizontal();  EditorGUILayout.PropertyField (useDepthTexture, new GUIContent ("Rely on Z Buffer?"));  if ((target as SunShafts).GetComponent<Camera>())  GUILayout.Label("Current camera mode: "+ (target as SunShafts).GetComponent<Camera>().depthTextureMode, EditorStyles.miniBoldLabel);  EditorGUILayout.EndHorizontal();  // depth buffer need  /\*  bool newVal = useDepthTexture.boolValue;  if (newVal != oldVal) {  if (newVal)  (target as SunShafts).camera.depthTextureMode |= DepthTextureMode.Depth;  else  (target as SunShafts).camera.depthTextureMode &= ~DepthTextureMode.Depth;  }  \*/  EditorGUILayout.PropertyField (resolution, new GUIContent("Resolution"));  EditorGUILayout.PropertyField (screenBlendMode, new GUIContent("Blend mode"));  EditorGUILayout.Separator ();  EditorGUILayout.BeginHorizontal();  EditorGUILayout.PropertyField (sunTransform, new GUIContent("Shafts caster", "Chose a transform that acts as a root point for the produced sun shafts"));  if ((target as SunShafts).sunTransform && (target as SunShafts).GetComponent<Camera>()) {  if (GUILayout.Button("Center on " + (target as SunShafts).GetComponent<Camera>().name)) {  if (EditorUtility.DisplayDialog ("Move sun shafts source?", "The SunShafts caster named "+ (target as SunShafts).sunTransform.name +"\n will be centered along "+(target as SunShafts).GetComponent<Camera>().name+". Are you sure? ", "Please do", "Don't")) {  Ray ray = (target as SunShafts).GetComponent<Camera>().ViewportPointToRay(new Vector3(0.5f,0.5f,0));  (target as SunShafts).sunTransform.position = ray.origin + ray.direction \* 500.0f;  (target as SunShafts).sunTransform.LookAt ((target as SunShafts).transform);  }  }  }  EditorGUILayout.EndHorizontal();  EditorGUILayout.Separator ();  EditorGUILayout.PropertyField (sunThreshold, new GUIContent ("Threshold color"));  EditorGUILayout.PropertyField (sunColor, new GUIContent ("Shafts color"));  maxRadius.floatValue = 1.0f - EditorGUILayout.Slider ("Distance falloff", 1.0f - maxRadius.floatValue, 0.1f, 1.0f);  EditorGUILayout.Separator ();  sunShaftBlurRadius.floatValue = EditorGUILayout.Slider ("Blur size", sunShaftBlurRadius.floatValue, 1.0f, 10.0f);  radialBlurIterations.intValue = EditorGUILayout.IntSlider ("Blur iterations", radialBlurIterations.intValue, 1, 3);  EditorGUILayout.Separator ();  EditorGUILayout.PropertyField (sunShaftIntensity, new GUIContent("Intensity"));  serObj.ApplyModifiedProperties();  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  namespace BNG {  /// <summary>  /// An example Grabbable that lets you move along a designated path  /// </summary>  public class Zipline : GrabbableEvents {  public Transform ZiplineStart;  public Transform ZiplineEnd;  public float ZiplineSpeed = 1;  public bool UseLinearMovement = true;  float lastMoveTime = -1f;  bool movingForward = true;  AudioSource audioSource;  void Start() {  // Start off by orienting the zipline holder  if(ZiplineEnd != null) {  transform.LookAt(ZiplineEnd.position);  }  audioSource = GetComponent<AudioSource>();  }  void Update() {  // Play vs. stop sound  if (Time.time - lastMoveTime < 0.1f) {  // Raise the pitch a little bit if we are moving forwards  if(movingForward) {  audioSource.pitch = Time.timeScale \* 1f;  }  else {  audioSource.pitch = Time.timeScale \* 0.95f;  }  // Make sure the clip is playing  if(!audioSource.isPlaying) {  audioSource.Play();  }  }  else if(audioSource.isPlaying) {  audioSource.Stop();  }  }  void OnDrawGizmosSelected() {  if (ZiplineStart != null && ZiplineEnd != null) {  // Draws a blue line from this transform to the target  Gizmos.color = Color.green;  Gizmos.DrawLine(ZiplineStart.position, ZiplineEnd.position);  }  }  public override void OnTrigger(float triggerValue) {  if (triggerValue > 0.5f) {  moveTowards(ZiplineEnd.position, true);  }  base.OnTrigger(triggerValue);  }  public override void OnButton1() {  moveTowards(ZiplineStart.position, false);  base.OnButton1();  }  public override void OnButton2() {  moveTowards(ZiplineEnd.position, true);  base.OnButton2();  }  void moveTowards(Vector3 pos, bool forwardDirection) {  lastMoveTime = Time.time;  movingForward = forwardDirection;  // Orient Zipline  if (forwardDirection) {  transform.LookAt(pos);  }  else {  // If backward, look at object from rear  transform.LookAt(2 \* transform.position - pos);  }  // Linear Movement  if (UseLinearMovement) {  transform.position = Vector3.MoveTowards(transform.position, pos, ZiplineSpeed \* Time.fixedDeltaTime);  }  // Lerp  else {  transform.position = Vector3.Lerp(transform.position, pos, ZiplineSpeed \* Time.deltaTime);  }  // Haptics  if(input && thisGrabber) {  input.VibrateController(0.1f, 0.1f, 0.1f, thisGrabber.HandSide);  }  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.InputSystem;  namespace BNG {  public class TimeController : MonoBehaviour {  [Tooltip("Timescale to slow down to if slow down key is pressed")]  public float SlowTimeScale = 0.5f;  /// <summary>  /// If true, Y Button will always slow time. Useful for debugging. Otherwise call SlowTime / ResumeTime yourself, or use a Unity InputActionReference  /// </summary>  [Tooltip("If true, Y Button will always slow time. Useful for debugging. Otherwise call SlowTime / ResumeTime yourself")]  public bool YKeySlowsTime = true;  [Tooltip("Input Action used to initiate slow time")]  public InputActionReference SlowTimeAction;  [Tooltip("(Optional) Play this clip when starting to slow time")]  public AudioClip SlowTimeClip;  [Tooltip("(Optional) Play this clip when ending slow mo")]  public AudioClip SpeedupTimeClip;  [Tooltip("If true, will set Time.fixedDeltaTime to the device refresh rate")]  public bool SetFixedDelta = false;  [Tooltip("If true, will check for input in Update to slow down time. If false you'll need to call SlowTime() / ResumeTime() manually from script")]  public bool CheckInput = true;  public bool TimeSlowing  {  get { return \_slowingTime; }  }  bool \_slowingTime = false;  bool routineRunning = false;  float originalFixedDelta;  AudioSource audioSource;  public bool ForceTimeScale = false;  void Start() {  if(SetFixedDelta) {  Time.fixedDeltaTime = (Time.timeScale / UnityEngine.XR.XRDevice.refreshRate);  }  originalFixedDelta = Time.fixedDeltaTime;  audioSource = GetComponent<AudioSource>();  }  void Update() {  if(CheckInput) {  if (SlowTimeInputDown() || ForceTimeScale) {  SlowTime();  }  else {  ResumeTime();  }  }  }  public virtual bool SlowTimeInputDown() {  // Check default Y Key  if ((YKeySlowsTime && InputBridge.Instance.YButton)) {  return true;  }  if (SlowTimeAction != null) {  return SlowTimeAction.action.ReadValue<float>() > 0f;  }  return false;  }  public void SlowTime() {  if(!\_slowingTime) {  // Make sure we aren't running a routine  if(resumeRoutine != null) {  StopCoroutine(resumeRoutine);  }  // Play Slow time clip  audioSource.clip = SlowTimeClip;  audioSource.Play();  // Haptics  if(SpeedupTimeClip) {  InputBridge.Instance.VibrateController(0.1f, 0.2f, SpeedupTimeClip.length, ControllerHand.Left);  }  Time.timeScale = SlowTimeScale;  Time.fixedDeltaTime = originalFixedDelta \* Time.timeScale;  \_slowingTime = true;  }  }  private IEnumerator resumeRoutine;  public void ResumeTime() {  // toggled over; play audio cue  // Don't resume until we're done playing the initial sound clip  if(\_slowingTime && !audioSource.isPlaying && !routineRunning) {  resumeRoutine = resumeTimeRoutine();  StartCoroutine(resumeRoutine);  }  }  IEnumerator resumeTimeRoutine() {  routineRunning = true;  audioSource.clip = SpeedupTimeClip;  audioSource.Play();  InputBridge.Instance.VibrateController(0.1f, 0.2f, SpeedupTimeClip.length, ControllerHand.Left);  // Wait for a split second before resuming time again  yield return new WaitForSeconds(0.35f);  Time.timeScale = 1;  Time.fixedDeltaTime = originalFixedDelta;  \_slowingTime = false;  routineRunning = false;  }  }  } |
| using System.Collections;  using System.Collections.Generic;  using UnityEngine;  using UnityEngine.InputSystem;  namespace BNG {  public class VREmulator : MonoBehaviour {  [Header("Enable / Disable : ")]  [Tooltip("Use Emulator if true and HMDIsActive is false")]  public bool EmulatorEnabled = true;  [Header("Input : ")]  [SerializeField]  [Tooltip("Action set used specifically to mimic or supplement a vr setup")]  public InputActionAsset EmulatorActionSet;  [Header("Player Teleportation")]  [Tooltip("Will set the PlayerTeleport component's ForceStraightArrow = true while the emulator is active.")]  public bool ForceStraightTeleportRotation = true;  [Header("Move Player Up / Down")]  [Tooltip("If true, move the player eye offset up / down whenever PlayerUpAction / PlayerDownAction is called.")]  public bool AllowUpDownControls = true;  [Tooltip("Unity Input Action used to move the player up")]  public InputActionReference PlayerUpAction;  [Tooltip("Unity Input Action used to move the player down")]  public InputActionReference PlayerDownAction;  [Tooltip("Minimum height in meters the player can shrink to when using the PlayerDownAction")]  public float MinPlayerHeight = 0.2f;  [Tooltip("Maximum height in meters the player can grow to when using the PlayerUpAction")]  public float MaxPlayerHeight = 5f;  [Header("Head Look")]  [Tooltip("Unity Input Action used to lock the camera in game mode to look around")]  public InputActionReference LockCameraAction;  [Tooltip("Unity Input Action used to lock the camera in game mode to look around")]  public InputActionReference CameraLookAction;  [Tooltip("Multiply the CameraLookAction by this amount")]  public float CameraLookSensitivityX = 0.1f;  [Tooltip("Multiply the CameraLookAction by this amount")]  public float CameraLookSensitivityY = 0.1f;  [Tooltip("Minimum local Eulers degrees the camera can rotate")]  public float MinimumCameraY = -90f;  [Tooltip("Minimum local Eulers degrees the camera can rotate")]  public float MaximumCameraY = 90f;  [Header("Controller Emulation")]  [Tooltip("Unity Input Action used to mimic holding the Left Grip")]  public InputActionReference LeftGripAction;  [Tooltip("Unity Input Action used to mimic holding the Left Trigger")]  public InputActionReference LeftTriggerAction;  [Tooltip("Unity Input Action used to mimic having your thumb near a button")]  public InputActionReference LeftThumbNearAction;  [Tooltip("Unity Input Action used to move mimic holding the Right Grip")]  public InputActionReference RightGripAction;  [Tooltip("Unity Input Action used to move mimic holding the Right Grip")]  public InputActionReference RightTriggerAction;  [Tooltip("Unity Input Action used to mimic having your thumb near a button")]  public InputActionReference RightThumbNearAction;  float mouseRotationX;  float mouseRotationY;  Transform mainCameraTransform;  Transform leftControllerTranform;  Transform rightControllerTranform;  Transform leftHandAnchor;  Transform rightHandAnchor;  BNGPlayerController player;  SmoothLocomotion smoothLocomotion;  PlayerTeleport playerTeleport;  bool didFirstActivate = false;  Grabber grabberLeft;  Grabber grabberRight;  private float \_originalPlayerYOffset = 1.65f;  [Header("Shown for Debug : ")]  public bool HMDIsActive;  public Vector3 LeftControllerPosition = new Vector3(-0.2f, -0.2f, 0.5f);  public Vector3 RightControllerPosition = new Vector3(0.2f, -0.2f, 0.5f);  public void CheckHeadControls() {  // Hold LockCameraAction (example : right mouse button down ) to move camera around  if (LockCameraAction != null) {  // Lock  if (LockCameraAction.action.ReadValue<float>() == 1) {  // Lock Camera and cursor  Cursor.visible = false;  Cursor.lockState = CursorLockMode.Locked;  Vector3 mouseLook = Vector2.zero;  if(CameraLookAction != null) {  mouseLook = CameraLookAction.action.ReadValue<Vector2>();  }  // Fall back to mouse  else {  mouseLook = Mouse.current.delta.ReadValue();  }  // Rotation Y  mouseRotationY += mouseLook.y \* CameraLookSensitivityY;  mouseRotationY = Mathf.Clamp(mouseRotationY, MinimumCameraY, MaximumCameraY);  mainCameraTransform.localEulerAngles = new Vector3(-mouseRotationY, mainCameraTransform.localEulerAngles.y, 0);  // Move PLayer on X Axis  player.transform.Rotate(0, mouseLook.x \* CameraLookSensitivityX, 0);  }  // Unlock Camera  else {  Cursor.lockState = CursorLockMode.None;  Cursor.visible = true;  }  }  }  public bool RequireGameFocus = true;  public void CheckPlayerControls() {  // Require focus  if(RequireGameFocus && Application.isEditor && !Application.isFocused) {  return;  }  // Player Up / Down  if(AllowUpDownControls) {  if (PlayerUpAction != null && PlayerUpAction.action.ReadValue<float>() == 1) {  player.ElevateCameraHeight = Mathf.Clamp(player.ElevateCameraHeight + Time.deltaTime, MinPlayerHeight, MaxPlayerHeight);  }  else if (PlayerDownAction != null && PlayerDownAction.action.ReadValue<float>() == 1) {  player.ElevateCameraHeight = Mathf.Clamp(player.ElevateCameraHeight - Time.deltaTime, MinPlayerHeight, MaxPlayerHeight);  }  }  // Force Forward Arrow  if(ForceStraightTeleportRotation && playerTeleport != null && playerTeleport.ForceStraightArrow == false) {  playerTeleport.ForceStraightArrow = true;  }  // Player Move Forward / Back, Snap Turn  if (smoothLocomotion != null && smoothLocomotion.enabled == false) {  // Manually allow player movement if the smooth locomotion component is disabled  smoothLocomotion.CheckControllerReferences();  smoothLocomotion.UpdateInputs();  if(smoothLocomotion.ControllerType == PlayerControllerType.CharacterController) {  smoothLocomotion.MoveCharacter();  }  else if (smoothLocomotion.ControllerType == PlayerControllerType.Rigidbody) {  smoothLocomotion.MoveRigidCharacter();  }  }  }  void FixedUpdate() {  // Player Move Forward / Back, Snap Turn  //if (smoothLocomotion != null && smoothLocomotion.enabled == false && smoothLocomotion.ControllerType == PlayerControllerType.Rigidbody) {  // smoothLocomotion.MoveRigidCharacter();  //}  }  public virtual void UpdateControllerPositions() {  leftControllerTranform.transform.localPosition = LeftControllerPosition;  leftControllerTranform.transform.localEulerAngles = Vector3.zero;  rightControllerTranform.transform.localPosition = RightControllerPosition;  rightControllerTranform.transform.localEulerAngles = Vector3.zero;  }  void checkGrabbers() {  // Find Grabber Left  if (grabberLeft == null || !grabberLeft.isActiveAndEnabled) {  Grabber[] grabbers = FindObjectsOfType<Grabber>();  for (var x = 0; x < grabbers.Length; x++) {  if (grabbers[x] != null && grabbers[x].isActiveAndEnabled && grabbers[x].HandSide == ControllerHand.Left) {  grabberLeft = grabbers[x];  }  }  }  // Find Grabber Right  if (grabberRight == null || !grabberRight.isActiveAndEnabled) {  Grabber[] grabbers = FindObjectsOfType<Grabber>();  for (var x = 0; x < grabbers.Length; x++) {  if (grabbers[x] != null && grabbers[x].isActiveAndEnabled && grabbers[x].HandSide == ControllerHand.Right) {  grabberRight = grabbers[x];  }  }  }  }  public virtual void ResetHands() {  leftControllerTranform.transform.localPosition = Vector3.zero;  leftControllerTranform.transform.localEulerAngles = Vector3.zero;  rightControllerTranform.transform.localPosition = Vector3.zero;  rightControllerTranform.transform.localEulerAngles = Vector3.zero;  }  public virtual void ResetAll() {  ResetHands();  // Reset Camera  mainCameraTransform.localEulerAngles = Vector3.zero;  // Reset Player  if (player) {  player.ElevateCameraHeight = \_originalPlayerYOffset;  }  // Reset Teleport Status  if(ForceStraightTeleportRotation && playerTeleport) {  playerTeleport.ForceStraightArrow = priorStraightSetting;  }  didFirstActivate = false;  }  void OnEnable() {  if (EmulatorActionSet != null) {  foreach (var map in EmulatorActionSet.actionMaps) {  foreach (var action in map) {  if(action != null) {  action.Enable();  }  }  }  }  // Subscribe to input events  InputBridge.OnInputsUpdated += UpdateInputs;  Application.onBeforeRender += OnBeforeRender;  }  void OnDisable()  {  // Disable Input Actions  if (EmulatorActionSet != null)  {  foreach (var map in EmulatorActionSet.actionMaps)  {  foreach (var action in map) {  if (action != null)  {  action.Disable();  }  }  }  }  Application.onBeforeRender -= OnBeforeRender;  if (isQuitting)  {  return;  }  // Reset Hand Positions  ResetAll();  // Unsubscribe from input events  InputBridge.OnInputsUpdated -= UpdateInputs;  }  bool isQuitting = false;  void OnApplicationQuit()  {  isQuitting = true;  }  }  } |