# MANAGERS

## Canvas manager

/// <summary>  
/// In Charge to display and managed all the UI elements in the game  
/// </summary>  
public class CanvasManager : MonoBehaviourHelper  
{  
 public int NumberOfWinOrLoseToShowInterstitial = 10;  
  
 AudioSource \_music;  
  
 public AudioSource music  
 {  
 get  
 {  
 if (\_music == null)  
 \_music = Camera.main.GetComponentInChildren<AudioSource>();  
  
 return \_music;  
 }  
 }  
  
 public Text levelText;  
  
 public Button buttonNextLevel;  
 public Button buttonLastLevel;  
  
 public Button buttonSetting;  
  
 public Button buttonUnlock;  
  
 public Button buttonLike;  
 public Button buttonLeaderboard;  
 public Button buttonRate;  
 public Button buttonShare;  
 public Button buttonMoreGames;  
 public Button buttonSound;  
  
 int maxLevel  
 {  
 get { return Util.GetMaxLevelUnlock(); }  
 }  
  
 int lastLevel  
 {  
 get { return Util.GetLastLevelPlayed(); }  
 }  
  
 /// <summary>  
 /// Set all the UI In Game Buttons  
 /// </summary>  
 void SetButtons()  
 {  
 buttonLastLevel.onClick.AddListener(() =>  
 {  
 buttonUnlock.transform.DOKill();  
 buttonUnlock.transform.DOScale(Vector3.zero, 0.3f);  
 Util.SetCountGameOver(0);  
 ButtonLogic();  
 OnClickedButtonPreviousLevel();  
 ButtonLogic();  
 });  
  
  
 buttonNextLevel.onClick.AddListener(() =>  
 {  
 buttonNextLevel.transform.DOKill();  
 buttonNextLevel.transform.DOScale(Vector3.zero, 0.3f);  
 Util.SetCountGameOver(0);  
 ButtonLogic();  
 OnClickedButtonNextLevel();  
 ButtonLogic();  
 });  
  
  
 foreach (Transform t in buttonSetting.transform.parent)  
 {  
 if (t.GetComponent<Canvas>() != null)  
 t.GetComponent<Canvas>().sortingOrder = 10 - t.GetSiblingIndex();  
 }  
  
 var gridLayoutGroup = buttonSetting.GetComponentInParent<GridLayoutGroup>();  
 gridLayoutGroup.spacing = new Vector2(0, -43);  
  
 buttonSetting.onClick.AddListener(() =>  
 {  
 buttonSetting.enabled = false;  
  
 float startvalue = 10;  
 float endvalue = -43;  
  
 if (gridLayoutGroup.spacing.y == -43)  
 {  
 startvalue = -43;  
 endvalue = 10;  
  
 buttonSetting.transform.DORotate(new Vector3(0, 0, 360), 1, RotateMode.FastBeyond360);  
 }  
 else  
 {  
 buttonSetting.transform.DORotate(new Vector3(0, 0, -360), 1, RotateMode.FastBeyond360);  
 }  
  
  
 DOVirtual.Float(startvalue, endvalue, 1,  
 (float value) => { gridLayoutGroup.spacing = new Vector2(0, value); })  
 .OnComplete(() => { buttonSetting.enabled = true; });  
 });  
  
 buttonUnlock.onClick.AddListener(() =>  
 {  
 buttonUnlock.transform.DOScale(Vector3.zero, 0.3f);  
 ShowRewardedVideoGameOver();  
 });  
  
 buttonUnlock.transform.localScale = Vector3.zero;  
  
  
 buttonLike.onClick.AddListener(() =>  
 {  
 string facebookApp = "fb://profile/515431001924232";  
 string facebookAddress = "https://www.facebook.com/appadvisory";  
  
 float startTime;  
 startTime = Time.timeSinceLevelLoad;  
  
 Application.OpenURL(facebookApp);  
  
 if (Time.timeSinceLevelLoad - startTime <= 1f)  
 {  
 Application.OpenURL(facebookAddress);  
 }  
 });  
  
 buttonLeaderboard.onClick.AddListener(() => { Debug.LogWarning("PUT YOUR CODE HERE"); });  
  
  
 buttonRate.onClick.AddListener(() => { });  
  
  
 buttonShare.onClick.AddListener(() => { Debug.LogWarning("PUT YOUR CODE HERE"); });  
  
  
 buttonMoreGames.onClick.AddListener(() => { Application.OpenURL("https://barouch.fr/moregames.php"); });  
  
 if (!Util.SoundIsOn())  
 {  
 music.Stop();  
 buttonSound.transform.GetChild(0).gameObject.SetActive(false);  
 buttonSound.transform.GetChild(1).gameObject.SetActive(true);  
 }  
 else  
 {  
 music.Play();  
 buttonSound.transform.GetChild(0).gameObject.SetActive(true);  
 buttonSound.transform.GetChild(1).gameObject.SetActive(false);  
 }  
  
 buttonSound.onClick.AddListener(() => { TurnSound(); });  
 }  
  
 /// <summary>  
 /// Turn on/off the sounds in the game  
 /// </summary>  
 void TurnSound()  
 {  
 if (Util.SoundIsOn())  
 {  
 music.Stop();  
 Util.SetSoundOff();  
 buttonSound.transform.GetChild(0).gameObject.SetActive(false);  
 buttonSound.transform.GetChild(1).gameObject.SetActive(true);  
 }  
 else  
 {  
 music.Play();  
 Util.SetSoundOn();  
 buttonSound.transform.GetChild(0).gameObject.SetActive(true);  
 buttonSound.transform.GetChild(1).gameObject.SetActive(false);  
 }  
  
 PlayerPrefs.Save();  
 }  
  
 void Awake()  
 {  
 DOTween.Init();  
  
  
 SetButtons();  
  
  
 ButtonLogic();  
 }  
  
 /// <summary>  
 /// Show rewarded video at game over  
 /// </summary>  
 private void ShowRewardedVideoGameOver()  
 {  
 gameManager.success = false;  
 gameManager.isGameOver = false;  
  
#if APPADVISORY\_ADS  
 if(AdsManager.instance.IsReadyRewardedVideo())  
 {  
 AdsManager.instance.ShowRewardedVideo ((bool success) => {  
 if(success)  
 PlayNextLevel ();  
 });  
 }  
 #endif  
 }  
  
  
 /// <summary>  
 /// Display the next and/or last button (the arrow around the level at the top of the screen)  
 /// </summary>  
 public void ButtonLogic()  
 {  
// if (gameManager.isGameOver || gameManager.success)   
// {  
// SetButtonActive(buttonLastLevel,false);  
// SetButtonActive(buttonNextLevel,false);  
// return;  
// }  
  
 SetButtonActive(buttonLastLevel, Util.ActivateButtonLast());  
  
 SetButtonActive(buttonNextLevel, Util.ActivateButtonNext());  
 }  
  
 /// <summary>  
 /// Activate and enable - or not - buttons  
 /// </summary>  
 void SetButtonActive(Button b, bool isActive)  
 {  
 if (isActive)  
 {  
 b.GetComponent<Image>().color = new Color(b.GetComponent<Image>().color.r, b.GetComponent<Image>().color.g,  
 b.GetComponent<Image>().color.b, 1);  
 b.interactable = true;  
 b.transform.localScale = new Vector3(0.7f, 0.7f, 1f);  
 }  
 else  
 {  
 b.GetComponent<Image>().color = new Color(b.GetComponent<Image>().color.r, b.GetComponent<Image>().color.g,  
 b.GetComponent<Image>().color.b, 0);  
 b.interactable = false;  
 b.transform.localScale = new Vector3(0.7f, 0.7f, 1f);  
 }  
 }  
  
 IEnumerator Start()  
 {  
 yield return new WaitForSeconds(0.1f);  
  
 PlayLevel(lastLevel);  
 }  
  
 /// <summary>  
 /// When the player failed, we show an unlock button ONLY IF there is a rewarded video available  
 /// </summary>  
 void ShowButtonUnlock()  
 {  
#if APPADVISORY\_ADS  
 if (AdsManager.instance.IsReadyRewardedVideo())   
 {  
 if (buttonUnlock.transform.localScale.x == 1)   
 {  
 buttonUnlock.transform.DOScale (Vector3.one \* 1.5f, 0.3f).SetLoops (6, LoopType.Yoyo);  
 }  
 else  
 {  
 buttonUnlock.transform.DOScale (Vector3.one, 0.3f);  
 }  
 }   
 return;  
 #endif  
 }  
  
 void ShowAdsLogic()  
 {  
#if APPADVISORY\_ADS  
 int count = PlayerPrefs.GetInt("GAMEOVER\_COUNT",0);  
 count ++;  
 if(count > NumberOfWinOrLoseToShowInterstitial)  
 {  
 if(AdsManager.instance.IsReadyInterstitial())  
 {  
 PlayerPrefs.SetInt("GAMEOVER\_COUNT",0);  
 AdsManager.instance.ShowInterstitial ();  
 }  
 else  
 {  
 PlayerPrefs.SetInt("GAMEOVER\_COUNT",count);  
 }  
 }  
 else  
 {  
 PlayerPrefs.SetInt("GAMEOVER\_COUNT",count);  
 }  
 PlayerPrefs.Save();  
 #endif  
 }  
  
 /// <summary>  
 /// Animation when the player fails   
 /// </summary>  
 public void AnimationCameraGameOver(Vector3 impactPosition)  
 {  
 FindObjectOfType<RateUsManager>().CheckIfPromptRateDialogue();  
  
 ShowAdsLogic();  
  
 ShowButtonUnlock();  
  
 ReplayCurrentLevel(lastLevel);  
 }  
  
 /// <summary>  
 /// Animation when the player cleared a level   
 /// </summary>  
 public void AnimationCameraSuccess()  
 {  
 Util.SetCountGameOver(0);  
  
 FindObjectOfType<RateUsManager>().CheckIfPromptRateDialogue();  
  
 ShowAdsLogic();  
  
 buttonUnlock.transform.DOScale(Vector3.zero, 0.3f);  
  
 PlayNextLevel();  
 }  
  
 /// <summary>  
 /// Run the level logic on the UI side  
 /// </summary>  
 private void PlayLevel(int level)  
 {  
 levelText.text = "Level " + level.ToString() + " / 1200";  
  
 if (level > maxLevel)  
 Util.SetMaxLevelUnlock(level);  
  
 Util.SetLastLevelPlayed(level);  
  
 ButtonLogic();  
  
 gameManager.CreateGame(level);  
 }  
  
 /// <summary>  
 /// Method called when the player clicked on the left arrow on the left of the level text on the top of the screen during the game  
 /// </summary>  
 private void OnClickedButtonPreviousLevel()  
 {  
 int last = lastLevel;  
  
 last--;  
  
 if (last < 1)  
 last = 1;  
  
 levelText.text = "Level " + last.ToString();  
 // levelTextMesh.text = last.ToString();  
 PlayLevel(last);  
 }  
  
 /// <summary>  
 /// Method called when the player clicked on the right arrow on the roght of the level text on the top of the screen during the game  
 /// </summary>  
 private void OnClickedButtonNextLevel()  
 {  
 PlayNextLevel();  
 }  
  
 /// <summary>  
 /// Method called when the player failed and so ... we replay the current level  
 /// </summary>  
 private void ReplayCurrentLevel(int level)  
 {  
 Camera.main.transform.DOMove(new Vector3(0, Camera.main.transform.position.y, -10), 0.3f)  
 .OnComplete(() => { PlayLevel(level); });  
 }  
  
 /// <summary>  
 /// Method called when the player have to play the next level (if the current level is cleared, or if the payer taps/Clicks on the next button or if the player see a rewarded video to unlock the current level  
 /// </summary>  
 private void PlayNextLevel()  
 {  
 int last = lastLevel;  
  
 last++;  
  
 levelText.text = "Level " + last.ToString();  
  
 PlayLevel(last);  
 }  
}

## Dot Manager

/// <summary>  
/// Attached to the each obstacles  
/// </summary>  
public class DotManager : MonoBehaviourHelper  
{  
 public bool isEnable = false;  
  
 public float position = 0;  
  
 public SpriteRenderer lineSprite;  
  
 public bool isOnCircle;  
  
 public SpriteRenderer DotSprite;  
  
 bool \_isBlack;  
  
 /// <summary>  
 /// is black = hazard. If write = to destroy  
 /// </summary>  
 public bool isBlack  
 {  
 get { return \_isBlack; }  
  
 set  
 {  
 \_isBlack = value;  
  
 if (value)  
 {  
 DotSprite.color = constant.SquareColor;  
 DotSprite.sortingOrder = 10;  
 }  
 else  
 {  
 DotSprite.color = constant.DotColor;  
 DotSprite.sortingOrder = 1;  
 transform.localPosition = new Vector3(transform.localPosition.x, transform.localPosition.y,  
 transform.localPosition.z - 0.01f);  
 }  
  
 lineSprite.color = DotSprite.color;  
 }  
 }  
  
  
 public Vector3 collisionPoint;  
  
  
 void Awake()  
 {  
 isBlack = false;  
  
 DotSprite.color = constant.DotColor;  
  
 Reset();  
 }  
  
  
 void Reset()  
 {  
 transform.rotation = Quaternion.identity;  
 if (lineSprite != null)  
 lineSprite.color = Color.clear;  
  
 isOnCircle = true;  
  
 transform.localScale = Vector3.one;  
  
 transform.rotation = Quaternion.identity;  
  
 isBlack = false;  
  
 DotSprite.color = new Color(DotSprite.color.r, DotSprite.color.g, DotSprite.color.b, 1f);  
 }  
  
 int ratio;  
  
 /// <summary>  
 /// Display the line of each hazard  
 /// </summary>  
 public void ActivateLine(Vector3 target, Transform CircleBorder)  
 {  
 transform.position = target;  
  
 position = Vector2.Distance(target, CircleBorder.position);  
  
 transform.parent = CircleBorder;  
 transform.localScale = Vector3.one;  
  
 if (lineSprite != null)  
 lineSprite.transform.localScale = new Vector3(position \* 100000 / 2, lineSprite.transform.localScale.y,  
 lineSprite.transform.localScale.z);  
 }  
  
 /// <summary>  
 /// If player touch a black square => game over  
 /// </summary>  
 void GameOverLogic(Collider2D col)  
 {  
 if (gameObject.name.Contains("Square"))  
 {  
 if (col.CompareTag("Player"))  
 {  
 if (col.gameObject.activeInHierarchy && gameObject.activeInHierarchy && !gameManager.isGameOver)  
 {  
 if (isBlack)  
 {  
 gameManager.GameOver(transform);  
 }  
 else  
 {  
 gameManager.SpawnParticleExplosionSquare(this);  
  
 StopAll();  
 }  
 }  
 }  
 }  
 }  
  
 public void StopAll()  
 {  
 if (lineSprite != null)  
 {  
 lineSprite.color = Color.clear;  
 }  
 }  
  
 /// <summary>  
 /// Trerred when enter an obtacle  
 /// </summary>  
 void OnTriggerEnter2D(Collider2D col)  
 {  
 if (Application.isEditor)  
 {  
 }  
  
 GameOverLogic(col);  
 }  
}

## Game Manager

public class GameManager : MonoBehaviourHelper  
{  
 public float cameraSize = 20;  
  
 float floorPosition = 4.19f;  
  
 public int Level;  
  
 bool firstStart = true;  
  
 [NonSerialized] public bool success;  
  
 [NonSerialized] public bool isGameOver;  
  
 float height;  
  
 public Transform CircleBorder;  
  
 public SpriteRenderer CircleCenterSprite;  
  
 public float speed = 1f;  
  
 public float positionTouchBorder;  
  
 DotManager lastShoot;  
  
 public Transform rotatePlayer;  
  
 public SpriteRenderer spriteDotGameOverZoom;  
  
 Vector3 rotateVector;  
  
 private float SizeRayonRatio = 1f;  
  
 private Ease easeType = Ease.Linear;  
  
 private LoopType loopType = LoopType.Incremental;  
  
 private float rotateCircleDelay = 6f;  
  
 private int numberDotsOnCircle;  
  
 Tweener jumpTweener;  
  
 public float waitTime = 0.15f;  
  
 public int segments = 10;  
  
 Vector3 rotateDOTVector = new Vector3(0, 0, 1);  
  
 /// <summary>  
 /// Keep a reference of the Dotween sequence use to rotate the circle and the dots linked to the circle  
 /// </summary>  
 Sequence sequence;  
  
 /// <summary>  
 /// Keep a reference of the Dotween sequence use to move around the player  
 /// </summary>  
 Sequence sequenceDOT;  
  
 /// <summary>  
 /// list of all the dots the player have to shoot in the level  
 /// </summary>  
 List<DotManager> DotsBottom;  
  
 float positionGravity;  
  
 float sizeDot = 0;  
  
 Vector3 pos  
 {  
 get { return new Vector3(0, -positionTouchBorder, 0); }  
 }  
  
 Quaternion rot  
 {  
 get { return Quaternion.identity; }  
 }  
  
 Transform parent  
 {  
 get { return CircleBorder; }  
 }  
  
 /// <summary>  
 /// List of square to be used in the game.  
 /// </summary>  
 List<DotManager> LIST\_SQUARE = new List<DotManager>();  
  
 /// <summary>  
 /// Do it at first. Some configurations.  
 /// </summary>  
 void Awake()  
 {  
 height = 2f \* 20;  
  
 Camera.main.transform.position = new Vector3(0, 0, -10);  
  
  
 DotsBottom = new List<DotManager>();  
  
 AddTouchListener();  
  
 ResetPosition();  
  
 Util.SetCountGameOver(0);  
 }  
  
 /// <summary>  
 /// Adding the touch listener to control player.  
 /// </summary>  
 void AddTouchListener()  
 {  
 InputTouch.onTouchDown += delegate(Vector3 pos)  
 {  
 if (pos.x < Screen.width \* 0.9f && pos.y < Screen.height \* 0.9f)  
 DoJump();  
 };  
  
 InputTouch.onTouchUp += delegate(Vector3 pos)  
 {  
 if (pos.x < Screen.width \* 0.9f && pos.y < Screen.height \* 0.9f)  
 DoWalk();  
 };  
 }  
  
 void Start()  
 {  
 guyAnim.MakeItBlink();  
  
 spriteDotGameOverZoom.transform.localScale = Vector3.zero;  
 }  
  
 /// <summary>  
 /// Do the aniamtion jump of the player  
 /// </summary>  
 void DoJump()  
 {  
 if (isGameOver)  
 return;  
  
 if (jumpTweener != null)  
 jumpTweener.Kill();  
  
 var ratio = Mathf.Abs(PLAYER.localPosition.y - positionTouchBorder) / (positionTouchBorder - floorPosition);  
  
 guyAnim.DoJump();  
  
 jumpTweener = PLAYER.DOLocalMoveY(positionTouchBorder, ratio \* waitTime, false);  
 }  
  
 /// <summary>  
 /// Do the aniamtion walk of the player  
 /// </summary>  
 void DoWalk()  
 {  
 if (isGameOver)  
 return;  
  
 if (jumpTweener != null)  
 jumpTweener.Kill();  
  
 var ratio = Mathf.Abs(PLAYER.localPosition.y - floorPosition) / (positionTouchBorder - floorPosition);  
 jumpTweener = PLAYER.DOLocalMoveY(floorPosition, ratio \* waitTime, false)  
 .OnComplete(guyAnim.DoWalk);  
 }  
  
 /// <summary>  
 /// Reset all position. We have to do this at start of each level  
 /// </summary>  
 void ResetPosition()  
 {  
 rotatePlayer.transform.position = Vector3.zero;  
 rotatePlayer.transform.localRotation = Quaternion.Euler(Vector3.zero);  
 PLAYER.localRotation = Quaternion.Euler(Vector3.zero);  
 rotateDOTVector = new Vector3(0, 0, 1);  
 }  
  
 /// <summary>  
 /// All the game level creation logic. We will create the current level.  
 /// </summary>  
 public void CreateGame(int level)  
 {  
 if (sequence != null)  
 sequence.Kill(false);  
  
 if (sequenceDOT == null)  
 {  
 ResetPosition();  
 SequenceDOTLogic();  
 }  
  
 canvasManager.ButtonLogic();  
  
 DOTween.Kill(CircleBorder);  
  
 DOTween.Kill(CircleBorder);  
  
  
 CancelInvoke();  
  
 StopAllCoroutines();  
  
 isGameOver = false;  
 success = false;  
  
 Level = Util.GetLastLevelPlayed();  
  
 Level l = levelManager.GetLevel(Level);  
 numberDotsOnCircle = l.numberDotsOnCircle;  
  
 SizeRayonRatio = l.sizeRayonRation;  
 rotateCircleDelay = l.rotateDelay;  
 easeType = l.rotateEaseType;  
 loopType = l.rotateLoopType;  
  
 positionTouchBorder = height \* SizeRayonRatio;  
  
 canvasManager.ButtonLogic();  
  
 PLAYER.localRotation = Quaternion.identity;  
  
 Time.timeScale = 1;  
  
 Application.targetFrameRate = 60;  
  
 GC.Collect();  
  
 PLAYER.localRotation = Quaternion.identity;  
  
  
 PLAYER.localScale = new Vector3(-rotateDOTVector.z, 1, 1);  
  
 PLAYER.localRotation = Quaternion.identity;  
  
 spriteDotGameOverZoom.transform.DOScale(Vector3.zero, 0.5f);  
  
 guyAnim.DoWalk();  
 this.Level = level;  
  
 Camera.main.orthographicSize = cameraSize;  
 Camera.main.transform.position = new Vector3(0, 0, -10);  
  
 StopAllCoroutines();  
  
 CircleCenterSprite.color = constant.SquareColor;  
  
 poolSystem.DespawnAll();  
  
  
 Camera.main.transform.position = new Vector3(0, 0, -10);  
  
 rotateVector = new Vector3(0, 0, 1);  
  
 if (Level % 2 == 0)  
 rotateVector = new Vector3(0, 0, -1);  
  
 CircleBorder.gameObject.SetActive(false);  
  
 CircleBorder.localScale = Vector3.one;  
  
 DOTween.Kill(PLAYER);  
  
 PLAYER.localPosition = new Vector3(PLAYER.localPosition.x, floorPosition, PLAYER.localPosition.z);  
  
 guyAnim.MakeItBlink();  
  
 guyAnim.m\_collider.enabled = false;  
  
 CreateDotOnCircle();  
  
 CircleBorder.localScale = Vector3.one \* 0.001f;  
 CircleBorder.gameObject.SetActive(true);  
  
 int count = LIST\_SQUARE.Count;  
  
 guyAnim.MakeItBlink();  
  
 guyAnim.m\_collider.enabled = false;  
  
 for (int i = 0; i < count; i++)  
 {  
 DotManager dm = LIST\_SQUARE[i];  
  
 dm.ActivateLine(dm.transform.position, dm.transform.parent);  
  
 guyAnim.m\_collider.enabled = false;  
 }  
  
 CircleBorder.DOScale(Vector3.one, 1)  
 .SetDelay(0.3f)  
 .SetEase(Ease.InBack)  
 .OnComplete(() =>  
 {  
 canvasManager.ButtonLogic();  
 guyAnim.m\_collider.enabled = false;  
  
 Invoke("StopBlink", 1);  
 });  
 }  
  
 /// <summary>  
 /// Stop the player blinking (when the player blinks, we are invicible)  
 /// </summary>  
 void StopBlink()  
 {  
 canvasManager.ButtonLogic();  
  
 guyAnim.StopBlink();  
  
 guyAnim.enabled = true;  
  
 LaunchRotateCircle();  
 }  
  
 /// <summary>  
 /// Rotate the circle and the dots linked to it  
 /// </summary>  
 void LaunchRotateCircle()  
 {  
 SequenceLogic();  
 }  
  
 /// <summary>  
 /// The method we will continuously call to move the player around  
 /// </summary>  
 void SequenceDOTLogic()  
 {  
 PLAYER.localRotation = Quaternion.identity;  
  
 if (sequenceDOT != null)  
 sequenceDOT.Kill(false);  
  
 if (firstStart)  
 ResetPosition();  
  
 firstStart = false;  
  
 sequenceDOT = DOTween.Sequence();  
  
 rotateDOTVector \*= -1f;  
  
 PLAYER.DOScaleX(-rotateDOTVector.z, 0.2f);  
  
 LoopType loopDot = LoopType.Incremental;  
  
 sequenceDOT.Append(rotatePlayer.DOLocalRotate(rotateDOTVector \* 360, 5, RotateMode.FastBeyond360)  
 .SetEase(Ease.Linear).SetRelative(true));  
 sequenceDOT.SetLoops(1, loopDot);  
  
 sequenceDOT.OnStepComplete(() => { SequenceDOTLogic(); });  
 }  
  
 /// <summary>  
 /// The method we will continuously call to move the the world around  
 /// </summary>  
 void SequenceLogic()  
 {  
 if (sequence != null)  
 sequence.Kill(false);  
  
 sequence = DOTween.Sequence();  
  
 if (loopType == LoopType.Incremental)  
 {  
 sequence.Append(CircleBorder.DORotate(-rotateVector \* Random.Range(360, 520), rotateCircleDelay,  
 RotateMode.FastBeyond360).SetEase(easeType));  
 sequence.SetLoops(1, loopType);  
 }  
 else  
 {  
 sequence.Append(CircleBorder.DORotate(-rotateVector \* Random.Range(360, 520), rotateCircleDelay,  
 RotateMode.FastBeyond360).SetEase(easeType));  
 sequence.SetLoops(2, loopType);  
 }  
  
 sequence.OnStepComplete(SequenceLogic);  
  
 sequence.Play();  
 }  
  
 /// <summary>  
 /// Create the dots on the circle and activate the line to link the dots to the circle  
 /// </summary>  
 void CreateDotOnCircle()  
 {  
 LIST\_SQUARE = new List<DotManager>();  
  
 var rand = Level % 6;  
  
  
 if (rand == 0)  
 CreateParralax(1, 1, 1);  
 else if (rand == 1)  
 CreateParralax(1, 2, 3);  
 else if (rand == 2)  
 CreateSpiral();  
 else if (rand == 3)  
 CreateTriangle();  
 else if (rand == 4)  
 CreateUpAndDown();  
 else if (rand == 5)  
 CreateEscalier();  
 else if (rand == 6)  
 CreateBalagan();  
  
 CreateBlackSquare();  
 }  
  
  
 void CreateBlackSquare()  
 {  
 int n = LIST\_SQUARE.Count;  
  
 for (int i = 0; i < 5; i++)  
 {  
 System.Random rng = new System.Random();  
 while (n > 1)  
 {  
 n--;  
 int k = rng.Next(n + 1);  
 var value = LIST\_SQUARE[k];  
 LIST\_SQUARE[k] = LIST\_SQUARE[n];  
 LIST\_SQUARE[n] = value;  
 }  
 }  
  
 for (int i = 0; i < LIST\_SQUARE.Count; i++)  
 {  
 LIST\_SQUARE[i].isBlack = false;  
 }  
  
 int numBlackTotal = 1;  
  
 float temp = 5 - this.Level % 4f;  
 temp = LIST\_SQUARE.Count / (1 + temp + Util.GetCountGameOver());  
  
 int iTemp = (int) (temp);  
  
 numBlackTotal = iTemp + 1;  
  
 if (numBlackTotal <= 0)  
 numBlackTotal = 1;  
  
  
 if (LIST\_SQUARE.Count == 1)  
 {  
 LIST\_SQUARE[0].isBlack = false;  
 }  
 else  
 {  
 for (int i = 0; i < LIST\_SQUARE.Count; i++)  
 {  
 if (i < numBlackTotal)  
 LIST\_SQUARE[i].isBlack = true;  
 else  
 LIST\_SQUARE[i].isBlack = false;  
 }  
 }  
 }  
  
 void CreateParralax(int decal, int gap, int parralaxLength)  
 {  
 for (int i = 0; i < numberDotsOnCircle; i++)  
 {  
 float variable = 1;  
  
 if (Level % 5 > 3)  
 {  
 if (i % 2 == 0)  
 {  
 variable = 0.8f - ((Level % 2) / 10f);  
 }  
 else  
 {  
 variable = 1f - ((Level % 2) / 10f);  
 ;  
 }  
 }  
  
 CircleBorder.rotation = Quaternion.Euler(new Vector3(0, 0, ((float) i) \* 360f / numberDotsOnCircle));  
  
 DotManager dm = poolSystem.SpawnSquare(  
 variable \* pos \* parralaxLength \* decal / (parralaxLength \* decal + gap \* i % parralaxLength), rot,  
 parent);  
  
 LIST\_SQUARE.Add(dm);  
 }  
 }  
  
 void CreateSpiral()  
 {  
 for (int i = 0; i < numberDotsOnCircle; i++)  
 {  
 CircleBorder.rotation = Quaternion.Euler(new Vector3(0, 0, ((float) i) \* 360f / numberDotsOnCircle));  
  
 float var = 1;  
  
 DotManager dm = poolSystem.SpawnSquare(pos \* (100f - var \* i) / 100f, rot, parent);  
  
 LIST\_SQUARE.Add(dm);  
 }  
 }  
  
 void CreateTriangle()  
 {  
 int value = 1;  
 int sign = 1;  
  
 for (int i = 0; i < numberDotsOnCircle; i++)  
 {  
 CircleBorder.rotation = Quaternion.Euler(new Vector3(0, 0, ((float) i) \* 360f / numberDotsOnCircle));  
  
 if (value > 3)  
 sign = -1;  
  
 if (value < 2)  
 sign = 1;  
  
 DotManager dm = poolSystem.SpawnSquare(pos \* 5 / (5 + value), rot, parent);  
  
 value += sign;  
  
 LIST\_SQUARE.Add(dm);  
 }  
 }  
  
 void CreateUpAndDown()  
 {  
 int i = 0;  
  
 while (LIST\_SQUARE.Count < numberDotsOnCircle)  
 {  
 CircleBorder.rotation =  
 Quaternion.Euler(new Vector3(0, 0, ((float) i) \* 360f / (numberDotsOnCircle \* 0.5f)));  
  
 var dm = poolSystem.SpawnSquare(pos, rot, parent);  
  
 LIST\_SQUARE.Add(dm);  
  
 var dm2 = poolSystem.SpawnSquare(pos \* 0.7f, rot, parent);  
  
 LIST\_SQUARE.Add(dm2);  
  
 i++;  
 }  
 }  
  
 void CreateEscalier()  
 {  
 int i = 0;  
  
 int j = 0;  
  
 float lastPos = 0f;  
  
 float min = 0.3f;  
  
 float decalRotate = 20;  
  
 if (numberDotsOnCircle < 20)  
 min = 0.7f;  
 else if (numberDotsOnCircle < 30)  
 min = 0.5f;  
 else if (numberDotsOnCircle < 40)  
 {  
 min = 0.4f;  
 decalRotate = 20;  
 }  
  
 min = 0.7f;  
  
 while (i < numberDotsOnCircle)  
 {  
 float rotation = ((float) i) \* 360f;  
  
 if (lastPos < min)  
 {  
 j = 0;  
 lastPos = 1f;  
 }  
 else  
 {  
 j++;  
 lastPos -= 0.15f;  
 }  
  
 CircleBorder.rotation =  
 Quaternion.Euler(new Vector3(0, 0, j \* decalRotate + (rotation) / (numberDotsOnCircle \* 0.5f)));  
  
 var dm = poolSystem.SpawnSquare(pos \* lastPos, rot, parent);  
  
 LIST\_SQUARE.Add(dm);  
  
 i++;  
 }  
 }  
  
 void CreateBalagan()  
 {  
 int i = 0;  
  
 int j = 0;  
  
 float lastPosUp = 1f;  
 float lastPosDown = 0.3f;  
 float signPosUp = -1;  
 float signPosDown = +1;  
  
 float decalRotate = 20;  
  
 while (i < numberDotsOnCircle)  
 {  
 float rotation = ((float) i) \* 360f;  
  
 if (lastPosUp > 1)  
 {  
 j = 0;  
 lastPosUp = 1;  
 signPosDown = -1;  
 }  
  
 if (lastPosUp < 0.4f)  
 {  
 j++;  
 lastPosUp = 0.4f;  
 signPosDown = +1;  
 }  
  
 CircleBorder.rotation =  
 Quaternion.Euler(new Vector3(0, 0, j \* decalRotate + (rotation) / (numberDotsOnCircle \* 0.5f)));  
  
 var dmUp = poolSystem.SpawnSquare(pos \* lastPosUp, rot, parent);  
  
 LIST\_SQUARE.Add(dmUp);  
  
 i++;  
  
 if (lastPosDown > 1)  
 {  
 lastPosDown = 1;  
 signPosDown = -1;  
 }  
  
 if (lastPosDown < 0.4f)  
 {  
 lastPosDown = 0.4f;  
 signPosDown = +1;  
 }  
  
 CircleBorder.rotation =  
 Quaternion.Euler(new Vector3(0, 0, j \* decalRotate + 30 + (rotation) / (numberDotsOnCircle \* 0.5f)));  
  
 var dmDown = poolSystem.SpawnSquare(pos \* lastPosDown, rot, parent);  
  
 LIST\_SQUARE.Add(dmDown);  
  
 lastPosUp += signPosUp \* 0.1f;  
 lastPosDown += signPosDown \* 0.1f;  
  
 i++;  
 }  
 }  
  
 /// <summary>  
 /// Game Over logic  
 /// </summary>  
 public void GameOver(Transform d)  
 {  
 Util.SetCountGameOver(Util.GetCountGameOver() + 1);  
  
 CheckIfSuccess();  
  
 if (success)  
 return;  
  
 StopAllCoroutines();  
  
 soundManager.PlaySoundFail();  
 isGameOver = true;  
  
 canvasManager.ButtonLogic();  
  
 jumpTweener.Kill();  
  
 sequence.Kill();  
  
 sequenceDOT.Kill();  
  
 sequenceDOT = null;  
  
 guyAnim.StopAll();  
  
 Vector3 targetPosition = new Vector3(PLAYER.position.x, PLAYER.position.y, Camera.main.transform.position.z);  
  
 canvasManager.music.DOPitch(-1, 1f)  
 .OnComplete(() => { canvasManager.music.DOPitch(1, 1f); });  
  
 Camera.main.transform.DOShakePosition(0.3f, new Vector3(1, 1, 0), 10, 90, false)  
 .OnComplete(() =>  
 {  
 DOVirtual.Float(cameraSize, 5f, 0.3f, (float size) => { Camera.main.orthographicSize = size; });  
  
 Camera.main.transform.DOMove(targetPosition, 0.3f)  
 .OnComplete(() =>  
 {  
 PLAYER.DOLocalMoveY(floorPosition, 0.3f, false)  
 .OnUpdate(() =>  
 {  
 Camera.main.transform.position = new Vector3(PLAYER.position.x, PLAYER.position.y,  
 Camera.main.transform.position.z);  
 })  
 .OnComplete(() =>  
 {  
 Camera.main.transform.position = new Vector3(PLAYER.position.x, PLAYER.position.y,  
 Camera.main.transform.position.z);  
 });  
  
 PLAYER.DOLocalRotate(new Vector3(0, 0, 180), 0.3f)  
 .OnComplete(() =>  
 {  
 DOVirtual.DelayedCall(0.3f, () =>  
 {  
 guyAnim.DoWalk();  
  
 DOVirtual.DelayedCall(0.5f, () =>  
 {  
 guyAnim.StopAll();  
 spriteDotGameOverZoom.transform.DOScale(Vector3.one \* 10, 1)  
 .OnComplete(() => { canvasManager.AnimationCameraGameOver(d.position); });  
 });  
 });  
 });  
 });  
 });  
 }  
  
 /// <summary>  
 /// Move the list of the dots to shoot when a dot is shooted.  
 /// </summary>  
 IEnumerator PositioningDots()  
 {  
 for (int i = 0; i < DotsBottom.Count; i++)  
 {  
 if (DotsBottom.Count > 0)  
 {  
 DotsBottom[i].transform.localScale = Vector3.one;  
 DotsBottom[i].transform.DOMove(new Vector3(0, -positionTouchBorder + (-i - 2) \* sizeDot), 0.001f);  
 }  
 }  
  
 yield return new WaitForFixedUpdate();  
  
 for (int i = 0; i < DotsBottom.Count; i++)  
 {  
 DotsBottom[i].transform.localScale = Vector3.one;  
 DotsBottom[i].transform.position = new Vector3(DotsBottom[i].transform.position.x,  
 -positionTouchBorder + (-i - 2) \* sizeDot, 0);  
 }  
  
 yield return null;  
 }  
  
 /// <summary>  
 /// Display particle when we distroy a square  
 /// </summary>  
 public void SpawnParticleExplosionSquare(DotManager square)  
 {  
 poolSystem.SpawnParticle(square.transform.position, Quaternion.identity);  
 poolSystem.SpawnWave(square.transform.position, Quaternion.identity);  
  
 soundManager.PlaySoundBeep();  
  
 if (poolSystem.gameObject.activeInHierarchy)  
 {  
 poolSystem.DespawnSquare(square);  
 }  
  
 CheckIfSuccess();  
 }  
  
 /// <summary>  
 /// Check all conditions. If it's ok, launch success logic  
 /// </summary>  
 void CheckIfSuccess()  
 {  
 if (success)  
 return;  
  
  
 var dmtotal = poolSystem.squares;  
  
 var dmFiltered = dmtotal.FindAll(dot => dot.isEnable == true && dot.isBlack == false);  
  
 int numberDotsToDestroy = dmFiltered.Count;  
  
 if (!isGameOver && numberDotsToDestroy <= 0)  
 {  
 success = true;  
  
 Util.SetCountGameOver(0);  
  
 canvasManager.ButtonLogic();  
 }  
  
 if (success && !isGameOver && numberDotsToDestroy <= 0)  
 {  
 guyAnim.MakeItBlink();  
 soundManager.PlaySoundSuccess();  
 DoWalk();  
 isGameOver = true;  
  
  
 CircleBorder.DOScale(Vector3.zero, 1)  
 .SetEase(Ease.InBack)  
 .OnComplete(() => { canvasManager.AnimationCameraSuccess(); });  
 }  
 }  
  
 public void OnApplicationPause(bool pause)  
 {  
 if (!pause)  
 {  
 Resources.UnloadUnusedAssets();  
 Time.timeScale = 1.0f;  
 }  
 else  
 {  
 Resources.UnloadUnusedAssets();  
 Time.timeScale = 0.0f;  
 }  
 }  
  
 void OnApplicationQuit()  
 {  
 PlayerPrefs.Save();  
 }  
}

## Level manager

to manage the level of game

public class LevelManager : MonoBehaviourHelper  
{  
 void Awake()  
 {  
 int level = Util.GetLastLevelPlayed();  
 if (level > 1200)  
 Util.SetLastLevelPlayed(1200);  
 }  
  
  
 public Level GetLevel(int level)  
 {  
 Level l = new Level (level);  
 return l;  
 }  
}  
  
/// <summary>  
/// Level class with all the informations for to create the level in the GameManager  
/// </summary>  
[Serializable]  
public class Level  
{  
 /// <summary>  
 /// Don't create more than 1200 levels  
 /// </summary>  
 static int maxLevel = 1200;  
 /// <summary>  
 /// Level number. Max = 1200  
 /// </summary>  
 public int levelNumber = 0;  
 /// <summary>  
 /// Number of obstacles in the level  
 /// </summary>  
 public int numberDotsOnCircle = 0;  
 /// <summary>  
 /// Position of obstacles   
 /// </summary>  
 public float sizeRayonRation = 0.25f;  
 /// <summary>  
 /// Delay of one rotation  
 /// </summary>  
 public float rotateDelay = 8f;  
 /// <summary>  
 /// Ease type of the rotation  
 /// </summary>  
 public Ease rotateEaseType = Ease.Linear;  
 /// <summary>  
 /// Loop type of the rotation  
 /// </summary>  
 public LoopType rotateLoopType = LoopType.Incremental;  
 /// <summary>  
 /// Level constructor  
 /// </summary>  
 public Level (int level)  
 {  
 levelNumber = level;  
  
 rotateEaseType = Ease.Linear;  
  
 rotateLoopType = LoopType.Incremental;  
  
 sizeRayonRation = 0.25f;  
  
 rotateDelay = 20f - (level % 10);  
  
 if (level%2 <1)   
 {  
 rotateLoopType = LoopType.Incremental;  
 }  
 else   
 {  
 rotateLoopType = LoopType.Yoyo;  
 }  
  
 int numOfEnum = (System.Enum.GetValues (typeof(Ease)).Length);  
  
 int enumNumber = level % numOfEnum;  
 rotateEaseType = (Ease)(enumNumber);   
  
 while(rotateEaseType.ToString ().Contains ("Elastic") || rotateEaseType.ToString ().Contains ("INTERNAL\_Zero") || rotateEaseType.ToString ().Contains ("INTERNAL\_Custom"))  
 {  
 enumNumber++;  
 if (enumNumber >= numOfEnum)  
 enumNumber = 0;  
  
 rotateEaseType = (Ease)(enumNumber);  
 }  
  
 numberDotsOnCircle = (int)((10 + level % 35));  
   
 if (level < 12)  
 numberDotsOnCircle = 5;  
  
 if (level == 1)  
 numberDotsOnCircle = 1;  
  
 if (level == 2)  
 numberDotsOnCircle = 2;  
   
 if (level > maxLevel)   
 {  
  
 Util.SetLastLevelPlayed(1200);  
  
 level = 1200;  
  
 Util.SetMaxLevelUnlock(1200);  
  
 Application.OpenURL ("http://barouch.fr/moregames.php");  
  
 }  
 }  
}

# MonoBehaviourHelper.cs

it’s a class to create the instance of every class, component. A class to avoid some duplicate code.

public class MonoBehaviourHelper : MonoBehaviour   
{  
 PoolSystem \_poolSystem;  
 /// <summary>  
 /// Reference to the pool system  
 /// </summary>  
 public PoolSystem poolSystem  
 {  
 get  
 {  
 if (\_poolSystem == null)  
 \_poolSystem = FindObjectOfType<PoolSystem> ();  
  
 return \_poolSystem;  
 }  
 }  
  
 SoundManager \_soundManager;  
 /// <summary>  
 /// Reference to the sound manager  
 /// </summary>  
 public SoundManager soundManager  
 {  
 get  
 {  
 if (\_soundManager == null)  
 \_soundManager = FindObjectOfType<SoundManager> ();  
  
 return \_soundManager;  
 }  
 }  
  
  
 GameManager \_gameManager;  
 /// <summary>  
 /// Reference to the game manager  
 /// </summary>  
 public GameManager gameManager  
 {  
 get  
 {  
 if (\_gameManager == null)  
 \_gameManager = FindObjectOfType<GameManager> ();  
  
 return \_gameManager;  
 }  
 }  
  
 CanvasManager \_canvasManager;  
 /// <summary>  
 /// Reference to the canas manager  
 /// </summary>  
 public CanvasManager canvasManager  
 {  
 get  
 {  
 if (\_canvasManager == null)  
 \_canvasManager = FindObjectOfType<CanvasManager> ();  
  
 return \_canvasManager;  
 }  
 }  
  
 Constant \_constant;  
 /// <summary>  
 /// Reference to the Constant GameObject  
 /// </summary>  
 public Constant constant  
 {  
 get  
 {  
 if (\_constant == null)  
 \_constant = FindObjectOfType<Constant> ();  
  
 return \_constant;  
 }  
 }  
  
 LevelManager \_levelManager;  
 /// <summary>  
 /// Reference to the LevelManager GameObject  
 /// </summary>  
 public LevelManager levelManager  
 {  
 get  
 {  
 if (\_levelManager == null)  
 \_levelManager = FindObjectOfType<LevelManager> ();  
  
 return \_levelManager;  
 }  
 }  
  
 GuyAnim \_guyAnim;  
 /// <summary>  
 /// Reference to the player animation script logic  
 /// </summary>  
 public GuyAnim guyAnim  
 {  
 get  
 {  
 if (\_guyAnim == null)  
 \_guyAnim = FindObjectOfType<GuyAnim> ();  
  
 return \_guyAnim;  
 }  
 }  
  
 Transform \_PLAYER;  
 /// <summary>  
 /// Reference to the player transform  
 /// </summary>  
 public Transform PLAYER  
 {  
 get  
 {  
 if (\_PLAYER == null)  
 \_PLAYER = guyAnim.transform.parent;  
  
 return \_PLAYER;  
 }  
 }  
}

# PoolSystem.cs

Add to Game Manager

public class PoolSystem : MonoBehaviour   
{  
 /// <summary>  
 /// Obstacle prefab  
 /// </summary>  
 public GameObject squarePrefab;  
 /// <summary>  
 /// List of obstacles pooled  
 /// </summary>  
 public List<DotManager> squares = new List<DotManager>();  
 /// <summary>  
 /// Particle prefab  
 /// </summary>  
 public GameObject particlePrefab;  
 /// <summary>  
 /// List of particles pooled  
 /// </summary>  
 public List<GameObject> particles = new List<GameObject>();  
 /// <summary>  
 /// Partciel Wave prefab  
 /// </summary>  
 public GameObject waveParticlePrefab;  
 /// <summary>  
 /// List of particle waves pooled  
 /// </summary>  
 public List<GameObject> waves = new List<GameObject>();  
  
  
 void Awake()  
 {  
 PreparePools ();  
 }  
 /// <summary>  
 /// Create the pool for each gameobject  
 /// </summary>  
 void PreparePools()  
 {  
 while (squares.Count < 50)   
 {  
 squares.Add(DOInstantiate (squarePrefab).GetComponent<DotManager>());  
 }  
  
 while (particles.Count < 10)   
 {  
 particles.Add(DOInstantiate (particlePrefab));  
 }  
  
 while (waves.Count < 10)   
 {  
 waves.Add(DOInstantiate (waveParticlePrefab));  
 }  
 }  
 /// <summary>  
 /// Instantiate GameObject, then add it to the pooled list  
 /// </summary>  
 GameObject DOInstantiate(GameObject obj)  
 {  
 var o = Instantiate (obj) as GameObject;  
 o.transform.parent = transform;  
 o.SetActive (false);  
 return o;  
 }  
 /// <summary>  
 /// Despawn all pooled GameObject  
 /// </summary>  
 public void DespawnAll()  
 {  
 DespawnAllSquares ();  
 DespawnAllParticles ();  
 DespawnAllWaves ();  
 }  
 /// <summary>  
 /// Despawn all pooled square GameObjects  
 /// </summary>  
 public void DespawnAllSquares()  
 {  
 var objActive = squares.FindAll (o => o.isEnable == true);  
 foreach (var v in objActive)   
 {  
 DespawnSquare (v);  
 }  
 }  
 /// <summary>  
 /// Despawn all pooled particle GameObjects  
 /// </summary>  
 public void DespawnAllParticles()  
 {  
 var objActive = particles.FindAll (o => o.activeInHierarchy == true);  
 foreach (var v in objActive)   
 {  
 DespawnParticle (v);  
 }  
 }  
 /// <summary>  
 /// Despawn all pooled particle wave GameObjects  
 /// </summary>  
 public void DespawnAllWaves()  
 {  
 var objActive = waves.FindAll (o => o.activeInHierarchy == true);  
 foreach (var v in objActive)   
 {  
 DespawnWave (v);  
 }  
 }  
 /// <summary>  
 /// Spawn a square  
 /// </summary>  
 public DotManager SpawnSquare(Vector3 pos, Quaternion angles, Transform parent)  
 {  
   
 if (squares.Count > 0)   
 {  
 var l = squares.FindAll(o => o.isEnable == false);  
 if (l == null || l.Count == 0)   
 {  
 var obj = DOInstantiate (squarePrefab);  
 squares.Add (obj.GetComponent<DotManager>());  
 return SpawnSquare (pos, angles, parent);  
 }  
  
 l[0].transform.parent = parent;  
 l[0].transform.position = pos;  
 l[0].transform.rotation = angles;  
 l[0].gameObject.SetActive (true);  
 l[0].isEnable = true;  
 return l[0];  
 }  
   
 var ob = DOInstantiate (squarePrefab);  
 squares.Add (ob.GetComponent<DotManager>());  
 return SpawnSquare (pos, angles, parent);  
 }  
 /// <summary>  
 /// Despawn a square  
 /// </summary>  
 public void DespawnSquare(DotManager obj)  
 {  
 obj.transform.parent = transform;  
 obj.isEnable = false;  
 obj.gameObject.SetActive (false);  
 }  
 /// <summary>  
 /// Spawn a particle  
 /// </summary>  
 public GameObject SpawnParticle(Vector3 pos, Quaternion angles)  
 {  
 if (particles.Count > 0)   
 {  
 var l = particles.FindAll(o => o.activeInHierarchy == false);  
 if (l == null || l.Count == 0)   
 {  
 var obj = DOInstantiate (particlePrefab);  
 particles.Add (obj);  
 return SpawnParticle (pos, angles);  
 }  
   
 l[0].transform.position = pos;  
 l[0].transform.rotation = angles;  
 l[0].SetActive (true);  
 return l[0];  
 }  
  
 var ob = DOInstantiate (particlePrefab);  
 particles.Add (ob);  
 return SpawnParticle (pos, angles);  
 }  
 /// <summary>  
 /// Despawn a particle  
 /// </summary>  
 public void DespawnParticle(GameObject obj)  
 {  
 obj.transform.parent = transform;  
 obj.SetActive (false);  
 }  
 /// <summary>  
 /// Spawn a particle wave  
 /// </summary>  
 public GameObject SpawnWave(Vector3 pos, Quaternion angles)  
 {  
 if (waves.Count > 0)   
 {  
 var l = waves.FindAll(o => o.activeInHierarchy == false);  
 if (l == null || l.Count == 0)   
 {  
 var obj = DOInstantiate (waveParticlePrefab);  
 waves.Add (obj);  
 return SpawnWave (pos, angles);  
 }  
  
 l[0].transform.position = pos;  
 l[0].transform.rotation = angles;  
 l[0].SetActive (true);  
 return l[0];  
 }  
  
 var ob = DOInstantiate (waveParticlePrefab);  
 waves.Add (ob);  
 return SpawnWave (pos, angles);  
 }  
 /// <summary>  
 /// Despawn a particle wave  
 /// </summary>  
 public void DespawnWave(GameObject obj)  
 {  
 obj.transform.parent = transform;  
 obj.SetActive (false);  
 }  
}

* Configurator game object with 2 scripts: Constant.cs and LevelManager.cs

# Constant.cs

this class stores the color

public class Constant : MonoBehaviourHelper  
{  
 string shareMessage;  
 public string url;  
 public Texture2D smallIcon;  
  
  
 public Color FailColor;  
  
 public Color SuccessColor;  
  
 public Color BackgroundColor;  
  
 public Color DotColor;  
  
 public Color SquareColor;  
  
 public List<Color> backgroundColors = new List<Color>(); // 18 colors  
  
 public string GetShareMessage()  
 {  
 shareMessage =  
 "I'm on level "  
 + Util.GetMaxLevelUnlock()  
 + "! #"  
 + "Ninja Hero"  
 + " by #appadvisory \n ";  
  
  
 return shareMessage;  
 }  
  
 public Color RandomBrightColor()  
 {  
 if (backgroundColors == null || backgroundColors.Count == 0)  
 return Color.white;  
  
 return backgroundColors[Random.Range(0, backgroundColors.Count)];  
 }  
}

# Util.cs

To save data to PlaeryPrefs

public static class Util  
{  
 public static string COUNTGAMEOVER = "COUNTGAMEOVER";  
 public static string LAST\_LEVEL\_PLAYED = "LEVEL\_PLAYED";  
 public static string LEVEL\_UNLOCKED = "LEVEL";  
 public static string SOUND\_ON = "SOUND\_ON";  
 public static string NUMOFLEVELPLAYED = "NUMOFLEVELPLAYED";  
  
 public static void SetCountGameOver(int count)  
 {  
 PlayerPrefs.SetInt(COUNTGAMEOVER, count);  
 PlayerPrefs.Save();  
 }  
  
 public static int GetCountGameOver()  
 {  
 return PlayerPrefs.GetInt(COUNTGAMEOVER, 0);  
 }  
  
 public static void SetMaxLevelUnlock(int num)  
 {  
 PlayerPrefs.SetInt(LEVEL\_UNLOCKED, num);  
 PlayerPrefs.Save();  
 }  
  
 public static int GetMaxLevelUnlock()  
 {  
 return PlayerPrefs.GetInt(LEVEL\_UNLOCKED, 1);  
 }  
  
 public static void SetLastLevelPlayed(int num)  
 {  
 PlayerPrefs.SetInt(LAST\_LEVEL\_PLAYED, num);  
 PlayerPrefs.Save();  
 }  
  
 public static int GetLastLevelPlayed()  
 {  
 return PlayerPrefs.GetInt(LAST\_LEVEL\_PLAYED, 1);  
 }  
  
 public static void SetSound(bool ON)  
 {  
 if (ON)  
 SetSoundOn();  
 else  
 SetSoundOff();  
 }  
  
 public static void SetSoundOn()  
 {  
 PlayerPrefs.SetInt(SOUND\_ON, 1);  
 PlayerPrefs.Save();  
 }  
  
 public static void SetSoundOff()  
 {  
 PlayerPrefs.SetInt(SOUND\_ON, 0);  
 PlayerPrefs.Save();  
 }  
  
 public static bool SoundIsOn()  
 {  
 return PlayerPrefs.GetInt(SOUND\_ON, 1) == 1;  
 }  
  
 public static void ReloadCurrentLevel()  
 {  
 SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);  
 }  
  
 public static string GetCurrentLevelName()  
 {  
 return SceneManager.GetActiveScene().name;  
 }  
  
 public static void SetNumberOfLevelPLayed(int num)  
 {  
 PlayerPrefs.SetInt(NUMOFLEVELPLAYED, num);  
 PlayerPrefs.Save();  
 }  
  
 public static int GetNumberOfLevelPLayed()  
 {  
 return PlayerPrefs.GetInt(NUMOFLEVELPLAYED, 0);  
 }  
  
 public static bool ActivateButtonNext()  
 {  
 int currentLevel = GetLastLevelPlayed();  
 int max = GetMaxLevelUnlock();  
  
 bool canUnlock = false;  
  
 if (currentLevel < max)  
 canUnlock = true;  
  
// Debug.Log("current = " + currentLevel + " - max = " + max + " ---> canUnlock = " + canUnlock);  
  
 return canUnlock;  
 }  
  
 public static bool ActivateButtonLast()  
 {  
 int currentLevel = GetLastLevelPlayed();  
  
 bool canUnlock = false;  
  
 if (currentLevel > 1)  
 canUnlock = true;  
  
 return canUnlock;  
 }  
}

# UI & BACKGROUND

## Background from Camera background

### AnimCameraColor.cs

to animate the color of camera

public class AnimCameraColor : MonoBehaviourHelper   
{  
 void Start ()  
 {  
 AnimColor ();  
 }  
   
 void AnimColor()  
 {  
 Color c = constant.RandomBrightColor ();  
  
 Camera.main.DOColor (c, Random.Range (3, 10)).OnComplete (AnimColor);  
 }  
}

## 5 Backgrounds

we add 5 square backgrounds (root controls the rotation, plane in child controls material color).

### RotationBackgroundRandom.cs

public class RotationBackgroundRandom : MonoBehaviour  
{  
 void Start()  
 {  
 SequenceLogic();  
 }  
  
 Sequence sequence;  
  
 private void SequenceLogic()  
 {  
 if (sequence != null)  
 sequence.Kill();  
  
 sequence = DOTween.Sequence();  
  
 var rotateVector = new Vector3(0, 0, 1);  
  
 if (Random.Range(0, 2) == 0)  
 rotateVector = new Vector3(0, 0, -1);  
  
 sequence.Append(transform.DORotate(rotateVector \* Random.Range(360, 520), Random.Range(3f, 10f),  
 RotateMode.FastBeyond360));  
 sequence.SetLoops(1, LoopType.Incremental);  
  
 sequence.OnStepComplete(SequenceLogic);  
  
 sequence.Play();  
 }  
}

### ChangeMaterialColor.cs

This will add to a plane has material with Shader Gradient-NoTexutre-Radial-TwoColors…

public class ChangeMaterialColor : MonoBehaviourHelper  
{  
 public Renderer r;  
 public bool isAngle;  
  
 public void Start()  
 {  
 r = GetComponent<Renderer>();  
  
 AnimColor1();  
 AnimColor2();  
 AnimeOffset();  
 AnimeUVOffsetX();  
 AnimeUVOffsetY();  
 }  
  
 private void AnimColor1()  
 {  
 Color c = constant.RandomBrightColor();  
  
 if (Random.Range(0, 2) == 0)  
  
 c = new Color(c.r, c.g, c.b, Random.Range(0.4f, 0.6f));  
  
 r.material.DOColor(c, "\_Color", Random.Range(3, 10)).OnComplete(AnimColor1);  
 }  
  
 private void AnimColor2()  
 {  
 Color c = constant.RandomBrightColor();  
  
 c = new Color(c.r, c.g, c.b, Random.Range(0.4f, 0.6f));  
  
 r.material.DOColor(c, "\_Color2", Random.Range(3, 10)).OnComplete(AnimColor2);  
 }  
  
 private void AnimeOffset()  
 {  
 if (isAngle)  
 r.material.DOFloat(Random.Range(-360.00f, +360.00f), "\_Angle", Random.Range(5, 20)).OnComplete(AnimeOffset);  
 else  
 r.material.DOFloat(Random.Range(-1.00f, +1.00f), "\_Offset", Random.Range(5, 20)).OnComplete(AnimeOffset);  
 }  
  
 void AnimeUVOffsetX()  
 {  
 r.material.DOFloat(Random.Range(-0.50f, +0.50f), "\_UVXOffset", Random.Range(5, 20)).OnComplete(AnimeUVOffsetX);  
 }  
  
 void AnimeUVOffsetY()  
 {  
 r.material.DOFloat(Random.Range(-0.50f, +0.50f), "\_UVYOffset", Random.Range(5, 20)).OnComplete(AnimeUVOffsetY);  
 }  
}

### Shader

Shader "Gradient/No Texture/Radial/Two Colors/Regular UV/Alpha Blend" {  
  
 Properties {  
 \_Color ("First Color", Color) = (1,1,1,1) //Receive input from a fixed Color  
 \_Color2 ("Second Color", Color) = (1,1,1,1) //Receive input from a fixed Color  
 \_UVXOffset ("UV X Offset", float) = 0 //Receive input from UV coordinate X offset  
 \_UVYOffset ("UV Y Offset", float) = 0 //Receive input from UV coordinate Y offset  
 \_UVXScale ("UV X Scale", float) = 1.0 //Receive input from UV X scale  
 \_UVYScale ("UV Y Scale", float) = 1.0 //Receive input from UV Y scale  
 \_Angle ("Angle", float) = 0 //Receive input from rotation Angle (0..360 degrees)  
 \_Offset ("Offset", float) = 0 //Receive input from a float  
 }  
  
 SubShader {  
  
 //Define what queue/order to render this shader in  
 Tags {"Queue" = "Transparent" "RenderType" = "Transparent"} //Background | Geometry | AlphaTest | Transparent | Overlay  
  
 //Define a pass  
 Pass {  
  
 //Set up blending and other operations  
 Cull Off // Back | Front | Off - Do not cull any triangle faces  
 ZTest LEqual //Less | Greater | LEqual | GEqual | Equal | NotEqual | Always - Z-Buffer/Depth testing is off  
 ZWrite On //On | Off - Z coordinates from pixel positions will not be written to the Z/Depth buffer  
 AlphaTest Off //0.0 //Less | Greater | LEqual | GEqual | Equal | NotEqual | Always (also 0.0 (float value) | [\_AlphaTestThreshold]) - All pixels will continue through the graphics pipeline because alpha testing is Off  
 Lighting Off //On | Off - Lighting will not be calculated or applied  
 ColorMask RGBA //RGBA | RGB | A | 0 | any combination of R, G, B, A - Color channels allowed to be modified in the backbuffer are: RGBA  
 //BlendOp //Add // Min | Max | Sub | RevSub - BlendOp is not being used and will default to an Add operation when combining the source and destination parts of the blend mode  
 Blend SrcAlpha OneMinusSrcAlpha //SrcFactor DstFactor (also:, SrcFactorA DstFactorA) = One | Zero | SrcColor | SrcAlpha | DstColor | DstAlpha | OneMinusSrcColor | OneMinusSrcAlpha | OneMinusDstColor | OneMinusDstAlpha - Blending between shader output and the backbuffer will use blend mode 'Alpha Blend'  
 //Blend SrcAlpha OneMinusSrcAlpha = Alpha blending  
 //Blend One One = Additive  
 //Blend OneMinusDstColor One = Soft Additive  
 //Blend DstColor Zero = Multiplicative  
 //Blend DstColor SrcColor = 2x Multiplicative  
  
 CGPROGRAM //Start a program in the CG language  
 #pragma target 2.0 //Run this shader on at least Shader Model 2.0 hardware (e.g. Direct3D 9)  
 #pragma fragment frag //The fragment shader is named 'frag'  
 #pragma vertex vert //The vertex shader is named 'vert'  
 #include "UnityCG.cginc" //Include Unity's predefined inputs and macros  
  
 //Unity variables to be made accessible to Vertex and/or Fragment shader  
 //uniform sampler2D \_MainTex; //Define \_MainTex from Texture Unit 0 to be sampled in 2D  
 //uniform float4 \_MainTex\_ST; //Use the Float \_MainTex\_ST to pass the Offset and Tiling for the texture(s)  
 uniform fixed4 \_Color; //Use the Color \_Color provided by Unity  
 uniform fixed4 \_Color2; //Use the Color \_Color2 provided by Unity  
 uniform float \_UVXOffset;  
 uniform float \_UVYOffset;  
 uniform float \_UVXScale;  
 uniform float \_UVYScale;  
 uniform float \_Angle;  
 uniform float \_Offset;  
   
 //Data structure communication from Unity to the vertex shader  
 //Defines what inputs the vertex shader accepts  
 struct AppData {  
 float4 vertex : POSITION; //Receive vertex position  
 half2 texcoord : TEXCOORD0; //Receive texture coordinates  
 //half2 texcoord1 : TEXCOORD1; //Receive texture coordinates  
 //fixed4 color : COLOR; //Receive vertex colors  
 };  
  
 //Data structure for communication from vertex shader to fragment shader  
 //Defines what inputs the fragment shader accepts  
 struct VertexToFragment {  
 float4 pos : POSITION; //Send fragment position to fragment shader  
 half2 uv : TEXCOORD0; //Send interpolated texture coordinate to fragment shader  
 //half2 uv2 : TEXCOORD1; //Send interpolated texture coordinate to fragment shader  
 //fixed4 color : COLOR; //Send interpolated gouraud-shaded vertex color to fragment shader  
 };  
  
 //Vertex shader  
 VertexToFragment vert(AppData v) {  
 VertexToFragment o; //Create a data structure to pass to fragment shader  
 o.pos = UnityObjectToClipPos(v.vertex); //Include influence of Modelview + Projection matrices  
 //o.uv = TRANSFORM\_TEX(v.texcoord,\_MainTex);//Send texture coords from unit 0 to fragment shader  
 //o.uv = v.texcoord.xy;  
 v.texcoord.xy = half2((v.texcoord.x+\_UVXOffset)\*\_UVXScale,(v.texcoord.y+\_UVYOffset)\*\_UVYScale);  
 float ang = \_Angle\*-0.01745329; //0.01745329 is conversion of 360.0/(2\*PI) as 1.0/(360.0/(2\*PI)) to convert angle to radians  
 float sinX = sin(ang);  
 float cosX = cos(ang);  
 float sinY = sin(ang);  
 float2x2 rotationMatrix = float2x2(cosX,-sinX,sinY,cosX); //Position and rotate  
 o.uv = mul(v.texcoord.xy, rotationMatrix);  
 //o.uv2 = v.texcoord1.xy; //Send texture coords from unit 1 to fragment shader  
 //o.color = v.color; //Send interpolated vertex color to fragment shader  
 //o.color = \_Color; //Send solid color to fragment shader  
 return o; //Transmit data to the fragment shader  
 }  
  
 //Fragment shader  
 fixed4 frag(VertexToFragment i) : COLOR {  
 return fixed4(lerp(\_Color,\_Color2,abs(i.uv.x)+abs(i.uv.y)+\_Offset )); //Output diamond gradient  
 }  
  
 ENDCG //End of CG program  
  
 }  
 }  
}

# INPUT

## InputTouch.cs

public class InputTouch : MonoBehaviour  
{  
 /// <summary>  
 /// Delegate subscribe by the GameManager and trigger when the player make a touch/click  
 /// </summary>  
 public delegate void OnTouchDown(Vector3 pos);  
 public static event OnTouchDown onTouchDown;  
  
 /// <summary>  
 /// Delegate subscribe by the GameManager and trigger when the player release the touch/click  
 /// </summary>  
 public delegate void OnTouchUp(Vector3 pos);  
 public static event OnTouchUp onTouchUp;  
  
 /// <summary>  
 /// To block input when showing the rate us popup  
 /// </summary>  
 public bool BLOCK\_INPUT = false;  
  
 /// <summary>  
 /// Listening for inputs  
 /// </summary>  
 void Update()  
 {  
 if (BLOCK\_INPUT)  
 return;  
  
 if (Application.isMobilePlatform)  
 {  
 var nbTouches = Input.touchCount;  
  
 if (nbTouches > 0)  
 {  
 for (var i = 0; i < nbTouches; i++)  
 {  
 var touch = Input.GetTouch(i);  
  
 var phase = touch.phase;  
  
 if (phase == TouchPhase.Began && onTouchDown != null)  
 {  
 onTouchDown(touch.position);  
 break;  
 }  
  
 if (phase == TouchPhase.Ended && onTouchUp != null)  
 {  
 onTouchUp(touch.position);  
 break;  
 }  
 }  
 }  
 }  
  
 if (!Application.isMobilePlatform)  
 {  
 if (Input.GetMouseButtonDown(0) && onTouchDown != null)  
 onTouchDown(Input.mousePosition);  
  
 if (Input.GetMouseButtonUp(0) && onTouchUp != null)  
 onTouchUp(Input.mousePosition);  
 }  
 }  
}

# SOUND

## SoundManager.cs

public class SoundManager : MonoBehaviourHelper   
{  
 AudioSource a;  
  
 public AudioClip soundBeep;  
 public AudioClip soundFail;  
 public AudioClip soundSuccess;  
  
 bool canPlay  
 {  
 get  
 {  
 return Util.SoundIsOn();  
 }  
 }  
  
 void Awake()  
 {  
 a = FindObjectOfType<AudioSource> ();  
 }  
 /// <summary>  
 /// Play sound beep  
 /// </summary>  
 public void PlaySoundBeep()  
 {  
 if(canPlay)  
 a.PlayOneShot (soundBeep);  
 }  
 /// <summary>  
 /// Play sound fail  
 /// </summary>  
 public void PlaySoundFail()  
 {  
 if(canPlay)  
 a.PlayOneShot (soundFail);  
 }  
 /// <summary>  
 /// Play sound success  
 /// </summary>  
 public void PlaySoundSuccess()  
 {  
 if(canPlay)  
 a.PlayOneShot (soundSuccess);  
 }  
  
 public void MuteAllMusic()  
 {  
   
 }  
  
 public void UnmuteAllMusic()  
 {  
   
 }  
}

# PLAYER

## GuyAnim.cs

public class GuyAnim : MonoBehaviourHelper  
{  
 public Transform legLeft;  
 public Transform legRight;  
  
 public Animator animator;  
  
 public GameObject particle;  
  
 public List<SpriteRenderer> listSpriteRenderer;  
  
 public Collider2D m\_collider;  
  
 readonly Color color = new Color(Color.white.r, Color.white.g, Color.white.b, 1);  
  
 public void MakeItBlink()  
 {  
 transform.localRotation = Quaternion.identity;  
  
 m\_collider.enabled = false;  
  
  
 StopBlink();  
  
 foreach (var s in listSpriteRenderer)  
 {  
 m\_collider.enabled = false;  
  
 s.color = Color.black;  
  
 transform.localRotation = Quaternion.identity;  
  
  
 s.DOColor(color, 0.1f)  
 .SetLoops(-1, LoopType.Yoyo)  
 .OnComplete(() =>  
 {  
 s.color = Color.black;  
  
 m\_collider.enabled = true;  
  
 s.color = Color.black;  
  
 transform.localRotation = Quaternion.identity;  
  
 m\_collider.enabled = true;  
 });  
 }  
 }  
  
 public void StopBlink()  
 {  
 foreach (var s in listSpriteRenderer)  
 {  
 s.color = Color.black;  
  
 m\_collider.enabled = true;  
  
 DOTween.Kill(s);  
  
 m\_collider.enabled = true;  
  
 s.color = Color.black;  
 }  
  
 m\_collider.enabled = true;  
 }  
  
 public void DoJump()  
 {  
 particle.SetActive(true);  
  
 animator.Play("Idle");  
 }  
  
 public void DoWalk()  
 {  
 animator.Play("GuyAnimLegs");  
 particle.SetActive(false);  
 }  
  
 public void StopAll()  
 {  
 animator.Play("Idle");  
 }  
}

# AUTO DESTROY PARTICLE SYSTEM

## AutoDestroyParticle.cs

public class AutoDestroyParticle : MonoBehaviourHelper  
{  
 public bool isParticle;  
 public bool isWave;  
 public bool OnlyDeactivate;  
  
 void OnEnable()  
 {  
 StartCoroutine("CheckIfAlive");  
 }  
  
 IEnumerator CheckIfAlive()  
 {  
 while (true)  
 {  
 yield return new WaitForSeconds(0.5f);  
 if (!GetComponent<ParticleSystem>().IsAlive(true))  
 {  
 if (OnlyDeactivate)  
 {  
 gameObject.SetActive(false);  
 }  
 else  
 {  
 if (isParticle)  
 poolSystem.DespawnParticle(gameObject);  
  
 if (isWave)  
 poolSystem.DespawnWave(gameObject);  
 }  
  
 break;  
 }  
 }  
 }  
}