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

// This enum contains the different phases of a game turn
public enum TurnPhase {
    idle,
    pre,
    waiting,
    post,
    gameOver
}

public class Bartok : MonoBehaviour {
    static public Bartok S;
    // This is static so that there can definitely only be 1 current player
    static public Player CURRENT_PLAYER;

    public TextAsset deckXML;
    public TextAsset layoutXML;
    public Vector3 layoutCenter = Vector3.zero;

    // The number of degrees to fan each card in a hand
    public float handFanDegrees = 10f;
    public int numStartingCards = 7;
    public float drawTimeStagger = 0.1f;

    public bool ____________;
    public Deck deck;
    public List<CardBartok> drawPile;
    public List<CardBartok> discardPile;
    public BartokLayout layout;
    public Transform layoutAnchor;
    public List<Player> players;
    public CardBartok targetCard;
    public TurnPhase phase = TurnPhase.idle;
    public GameObject turnLight;
    public GameObject GTGameOver;
    public GameObject GTRoundResult;

    void Awake() {
        S = this;
        // Find the TurnLight by name
        turnLight = GameObject.Find("TurnLight");
        GTGameOver = GameObject.Find("GTGameOver");
        GTRoundResult = GameObject.Find("GTRoundResult");
        GTGameOver.SetActive(false);
        GTRoundResult.SetActive(false);
    }

    void Start() {
        deck = GetComponent<Deck>(); // Get the Deck
        deck.InitDeck(deckXML.text); // Pass DeckXML to it
        Deck.Shuffle(ref deck.cards); // This shuffles the deck
        // The ref keyword passes a reference to deck.cards, which allows
        // deck.cards to be modified by Deck.Shuffle()
    }
}
layout = GetComponent<BartokLayout>(); // Get the Layout
layout.ReadLayout(layoutXML.text); // Pass LayoutXML to it

drawPile = UpgradeCardsList( deck.cards );
LayoutGame();

// UpgradeCardsList casts the Cards in LCD to be CardBartoks
// Of course, they were all along, but this lets Unity know it
List<CardBartok> UpgradeCardsList(List<Card> lcd) {
    List<CardBartok> lcb = new List<CardBartok>();
    foreach (Card tcd in lcd) {
        lcb.Add( tcd as CardBartok );
    }
    return( lcb );
}

// Position all the cards in the drawPile properly
public void ArrangeDrawPile() {
    CardBartok tCB;
    for (int i=0; i<drawPile.Count; i++) {
        tCB = drawPile[i];
        tCB.transform.parent = layoutAnchor;
        tCB.transform.localPosition = layout.drawPile.pos;
        // Rotation should start at 0
        tCB.faceUp = false;
        tCB.SetSortingLayerName(layout.drawPile.layerName);
        tCB.SetSortOrder(-i*4); // Order them front-to-back
        tCB.state = CBState.drawpile;
    }
}

void LayoutGame() {
    // Create an empty GameObject to serve as an anchor for the tableau //1
    if (layoutAnchor == null) {
        GameObject tGO = new GameObject("_LayoutAnchor");
        // ^ Create an empty GameObject named _LayoutAnchor in the Hierarchy
        layoutAnchor = tGO.transform; // Grab its Transform
        layoutAnchor.transform.position = layoutCenter; // Position it
    }

    // Position the drawPile cards
    ArrangeDrawPile();

    // Set up the players
    Player pl;
    players = new List<Player>();
    foreach (SlotDef tSD in layout.slotDefs) {
        pl = new Player();
        pl.handSlotDef = tSD;
        players.Add(pl);
        pl.playerNum = players.Count;
    }
    players[0].type = PlayerType.human; // Make the 0th player human

    CardBartok tCB;

    // Deal 7 cards to each player
    for (int i=0; i<numStartingCards; i++) {
        for (int j=0; j<4; j++) { // There are always 4 players
            tCB = Draw(); // Draw a card
            // Stagger the draw time a bit. Remember order of operations.
tCB.timeStart = Time.time + drawTimeStagger * ( i*4 + j );
// ^ By setting the timeStart before calling AddCard, we
// override the automatic setting of timeStart by
// CardBartok.MoveTo().

// Add the card to the player's hand. The modulus (%) makes it
// a number from 0 to 3
players[ (j+i)%4 ].AddCard(tCB);
}

// Call Bartok.DrawFirstTarget() when the other cards are done.
Invoke("DrawFirstTarget", drawTimeStagger * (numStartingCards*4+4) );

public void DrawFirstTarget() {
    // Flip up the target card in the middle
    CardBartok tCB = MoveToTarget( Draw() );
    // Set the CardBartok to call CBCallback on this Bartok when it is done
    tCB.reportFinishTo = this.gameObject;
}

// This callback is used by the last card to be dealt at the beginning
// It is only used once per game.
public void CBCallback(CardBartok cb) {
    // You sometimes want to have reporting of method calls like this
    Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.CBCallback()");
    StartGame();
}

public void StartGame() {
    // Pick the player to the left of the human to go first.
    // (players[0] is the human)
    PassTurn(1);
}

public void PassTurn(int num=-1) {
    // If no number was passed in, pick the next player
    if (num == -1) {
        int ndx = players.IndexOf(CURRENT_PLAYER);
        num = (ndx+1)%4;
    }
    int lastPlayerNum = -1;
    if (CURRENT PLAYER != null) {
        lastPlayerNum = CURRENT PLAYER.playerNum;
        // Check for Game Over and need to reshuffle discards
        if ( CheckGameOver() ) {
            return;
        }
        CURRENT PLAYER = players[num];
        phase = TurnPhase.pre;
        CURRENT PLAYER.TakeTurn();
        // Move the TurnLight to shine on the new CURRENT PLAYER
        Vector3 lPos = CURRENT PLAYER.handSlotDef.pos + Vector3.back*5;
        turnLight.transform.position = lPos;
        // Report the turn passing
        Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.PassTurn()","Old: "+lastPlayerNum,"New: "+CURRENT PLAYER.playerNum);
    }
public bool CheckGameOver() {
    // See if we need to reshuffle the discard pile into the draw pile
    if (drawPile.Count == 0) {
        List<Card> cards = new List<Card>();
        foreach (CardBartok cb in discardPile) {
            cards.Add (cb);
        }
        discardPile.Clear();
        Deck.Shuffle( ref cards );
        drawPile = UpgradeCardsList(cards);
        ArrangeDrawPile();
    }
    // Check to see if the current player has won
    if (CURRENT_PLAYER.hand.Count == 0) {
        // The current player has won!
        if (CURRENT_PLAYER.type == PlayerType.human) {
            GGameOver.GetComponent<GUIText>().text = "You Won!";
            GRoundResult.GetComponent<GUIText>().text = "";
        } else {
            GGameOver.GetComponent<GUIText>().text = "Game Over";
            GRoundResult.GetComponent<GUIText>().text = "Player " + CURRENT_PLAYER.playerNum + " won";
        }
        GGameOver.SetActive(true);
        GRoundResult.SetActive(true);
        phase = TurnPhase.gameOver;
        Invoke("RestartGame", 1);
        return(true);
    }
    return(false);
}

public void RestartGame() {
    CURRENT_PLAYER = null;
    Application.LoadLevel("__Bartok_Scene_0");
}

public CardBartok MoveToTarget(CardBartok tCB) {
    tCB.timeStart = 0;
    tCB.MoveTo(layout.discardPile.pos+Vector3.back);
    tCB.state = CBState.toTarget;
    tCB.faceUp = true;
    tCB.SetSortingLayerName("10"); // layout.target.layerName);
    tCB.eventualSortLayer = layout.target.layerName;
    if (targetCard != null) {
        MoveToDiscard(targetCard);
    }
    targetCard = tCB;
    return(tCB);
}

public CardBartok MoveToDiscard(CardBartok tCB) {
    //Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.MoveToDiscard()", tCB.name);
    tCB.state = CBState.discard;
    discardPile.Add ( tCB );
    tCB.SetSortingLayerName(layout.discardPile.layerName);
    tCB.SetSortOrder( discardPile.Count*4 );
    tCB.transform.localPosition = layout.discardPile.pos + Vector3.back/2;
    return(tCB);
}
public CardBartok Draw() {
    CardBartok cd = drawPile[0]; // Pull the 0th CardProspector
    drawPile.RemoveAt(0); // Then remove it from List<> drawPile
    return (cd); // And return it
}

// ValidPlay verifies that the card chosen can be played on the discard pile
public bool ValidPlay(CardBartok cb) {
    // It's a valid play if the rank is the same
    if (cb.rank == targetCard.rank) return (true);

    // It's a valid play if the suit is the same
    if (cb.suit == targetCard.suit) {
        return (true);
    }

    // Otherwise, return false
    return (false);
}

/* Now is a good time to comment out this testing code

// This Update method is used to test passing cards to players
void Update() {
    if (Input.GetKeyDown(KeyCode.Alpha1)) {
        players[0].AddCard(Draw());
    }
    if (Input.GetKeyDown(KeyCode.Alpha2)) {
        players[1].AddCard(Draw());
    }
    if (Input.GetKeyDown(KeyCode.Alpha3)) {
        players[2].AddCard(Draw());
    }
    if (Input.GetKeyDown(KeyCode.Alpha4)) {
        players[3].AddCard(Draw());
    }
}
*/

public void CardClicked(CardBartok tCB) {
    // If it's not the human's turn, don't respond
    if (CURRENT_PLAYER.type != PlayerType.human) return;

    // If the game is waiting on a card to move, don't respond
    if (phase == TurnPhase.waiting) return;

    // Act differently based on whether it was a card in hand or on the drawPile that was clicked
    switch (tCB.state) {
        case CBState.drawpile:
            // Draw the top card, not necessarily the one clicked.
            CardBartok cb = CURRENT_PLAYER.AddCard(Draw());
            cb.callbackPlayer = CURRENT_PLAYER;
            Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.CardClicked()","Draw", cb.name);
            phase = TurnPhase.waiting;
            break;
        case CBState.hand:
            // Check to see whether the card is valid
            if (ValidPlay(tCB)) {
                CURRENT_PLAYER.RemoveCard(tCB);
                MoveToTarget(tCB);
                tCB.callbackPlayer = CURRENT_PLAYER;
                Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.CardClicked()","Play", tCB.name,targetCard.name+" is target");
                phase = TurnPhase.waiting;
            } else {
                // Do something else
            }
        
            // Continue with the rest of the switch
        default:
            break;
    }
}
// Just ignore it
Utils.tr (Utils.RoundToPlaces(Time.time),
  "Bartok.CardClicked()","Attempted to Play",tCB.name,targetCard.name+" is target");
  break;
}
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

// This is actually OUTSIDE of the Utils Class
public enum BoundsTest {
    center, // Is the center of the GameObject on screen
    onScreen, // Are the bounds entirely on screen
    offScreen // Are the bounds entirely off screen
}

public class Utils : MonoBehaviour {

    // Creates bounds that encapsulate of the two Bounds passed in.
    public static Bounds BoundsUnion(Bounds b0, Bounds b1) {
        // If the size of one of the bounds is Vector3.zero, ignore that one
        if (b0.size == Vector3.zero && b1.size != Vector3.zero) {
            return b1;
        }
        else if (b0.size != Vector3.zero && b1.size == Vector3.zero) {
            return b0;
        }
        else if (b0.size == Vector3.zero && b1.size == Vector3.zero) {
            return b0;
        }
        // Stretch b0 to include the b1.min and b1.max
        b0.Encapsulate(b1.min);
        b0.Encapsulate(b1.max);
        return b0;
    }

    public static Bounds CombineBoundsOfChildren(GameObject go) {
        Bounds b = new Bounds(Vector3.zero, Vector3.zero);
        // If this GameObject has a Renderer Component...
        if (go.GetComponent<Renderer>() != null) {
            // Expand b to contain the Renderer's Bounds
            b = BoundsUnion(b, go.GetComponent<Renderer>().bounds);
        }
        // If this GameObject has a Collider Component...
        if (go.GetComponent<Collider>() != null) {
            // Expand b to contain the Collider's Bounds
            b = BoundsUnion(b, go.GetComponent<Collider>().bounds);
        }
        // Iterate through each child of this gameObject.transform
        foreach (Transform t in go.transform) {
            // Expand b to contain their Bounds as well
            b = BoundsUnion(b, CombineBoundsOfChildren(t.gameObject));
        }
        return b;
    }

    // Make a static read-only public property camBounds
    static public Bounds camBounds {
        get {
            // if _camBounds hasn't been set yet
            if (_camBounds.size == Vector3.zero) {
                // SetCameraBounds using the default Camera
                SetCameraBounds();
            }
            return _camBounds;
        }
    }
}
// This is the private static field that camBounds uses

static private Bounds _camBounds;

public static void SetCameraBounds(Camera cam = null) {
    // If no Camera was passed in, use the main Camera
    if (cam == null) cam = Camera.main;
    // This makes a couple important assumptions about the camera:
    // 1. The camera is Orthographic
    // 2. The camera is at a rotation of R:[0,0,0]

    // Make Vector3s at the topLeft and bottomRight of the Screen coords
    Vector3 topLeft = new Vector3(0, 0, 0);
    Vector3 bottomRight = new Vector3(Screen.width, Screen.height, 0);

    // Convert these to world coordinates
    Vector3 boundTLN = cam.ScreenToWorldPoint(topLeft);
    Vector3 boundBRF = cam.ScreenToWorldPoint(bottomRight);

    // Adjust the z to be at the near and far Camera clipping planes
    boundTLN.z += cam.nearClipPlane;
    boundBRF.z += cam.farClipPlane;

    // Find the center of the Bounds
    Vector3 center = (boundTLN + boundBRF) / 2f;
    _camBounds = new Bounds(center, Vector3.zero);
    // Expand _camBounds to encapsulate the extents.
    _camBounds.Encapsulate(boundTLN);
    _camBounds.Encapsulate(boundBRF);
}

// Test to see whether Bounds are on screen.
public static Vector3 ScreenBoundsCheck(Bounds bnd, BoundsTest test = BoundsTest.onScreen) {
    // Call the more generic BoundsInBoundsCheck with camBounds as bigB
    return (BoundsInBoundsCheck(camBounds, bnd, test));
}

// Tests to see whether lilB is inside bigB
public static Vector3 BoundsInBoundsCheck(Bounds bigB, Bounds lilB, BoundsTest test = BoundsTest.onScreen) {
    // Get the center of lilB
    Vector3 pos = lilB.center;

    // Initialize the offset at [0,0,0]
    Vector3 off = Vector3.zero;

    switch (test) {
    // The center test determines what off (offset) would have to be applied to lilB to move
    // its center back inside bigB
    case BoundsTest.center:
        if (bigB.Contains(pos)) {
            return (Vector3.zero);
        }
        // if not contained, find the offset
        if (pos.x > bigB.max.x) {  // set the x off
            off.x = pos.x - bigB.max.x;
        } else if (pos.x < bigB.min.x) {  // set the x off
            off.x = pos.x - bigB.min.x;
        }
        if (pos.y > bigB.max.y) {  // set the y off
            off.y = pos.y - bigB.max.y;
        } else if (pos.y < bigB.min.y) {  // set the y off
            off.y = pos.y - bigB.min.y;
    }
```cpp
if (pos.z > bigB.max.z) {
    off.z = pos.z - bigB.max.z;
} else if (pos.z < bigB.min.z) {
    off.z = pos.z - bigB.min.z;
}
return (off);

// The onScreen test determines what off would have to be applied to keep all of lilB inside bigB

if (pos.z > bigB.max.z) {
    off.z = pos.z - bigB.max.z;
} else if (pos.z < bigB.min.z) {
    off.z = pos.z - bigB.min.z;
}
return (off);

// The offScreen test determines what off would need to be applied to move any tiny part of lilB inside of bigB

return (Vector3.zero);
```
// Transform Functions

public static GameObject FindTaggedParent(GameObject go) {
    // If this gameObject has a tag
    if (go.tag != "Untagged") {
        return (go);
    }
    // If there is no parent of this Transform
    if (go.transform.parent == null) {
        // We've reached the end of the line with no interesting tag
        return (null);
    }
    // Otherwise, recursively climb up the tree
    return (FindTaggedParent(go.transform.gameObject));
}

// This version of the function handles things if a Transform is passed in
public static GameObject FindTaggedParent(Transform t) {
    return (FindTaggedParent(t.gameObject));
}

// Materials Functions

static public Material[] GetAllMaterials(GameObject go) {
    List<Material> mats = new List<Material>();
    if (go.GetComponent<Renderer>() != null) {
        mats.Add(go.GetComponent<Renderer>().material);
    }
    foreach (Transform t in go.transform) {
        mats.AddRange(GetAllMaterials(t.gameObject));
    }
    return (mats.ToArray());
}

// Linear Interpolation

static public Vector3 Lerp(Vector3 vFrom, Vector3 vTo, float u) {
    Vector3 res = (1-u)*vFrom + u*vTo;
    return (res);
}

static public Vector2 Lerp(Vector2 vFrom, Vector2 vTo, float u) {
    Vector2 res = (1-u)*vFrom + u*vTo;
    return (res);
}

static public float Lerp(float vFrom, float vTo, float u) {
    float res = (1-u)*vFrom + u*vTo;
    return (res);
// While most Béier curves are 3 or 4 points, it is possible to have
// any number of points using this recursive function
// This uses the Utilities.Lerp function because it needs to allow extrapolation

static public Vector3 Bezier( float u, List<Vector3> vList ) {
    // If there is only one element in vList, return it
    if (vList.Count == 1) {
        return (vList[0]);
    }
    // Otherwise, create vListR, which is all but the 0th element of vList
    // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
    List<Vector3> vListR = vList.GetRange(1, vList.Count - 1);
    // And create vListL, which is all but the last element of vList
    // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
    List<Vector3> vListL = vList.GetRange(0, vList.Count - 1);
    // The result is the Lerp of these two shorter Lists
    Vector3 res = Lerp(Bezier(u, vListL), Bezier(u, vListR), u);
    return (res);
}

// This version allows an Array or a series of Vector3s as input
static public Vector3 Bezier( float u, params Vector3[] vecs ) {
    return (Bezier(u, new List<Vector3>(vecs)));
}

// The same two functions for Vector2
static public Vector2 Bezier( float u, List<Vector2> vList ) {
    // If there is only one element in vList, return it
    if (vList.Count == 1) {
        return (vList[0]);
    }
    // Otherwise, create vListR, which is all but the 0th element of vList
    // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
    List<Vector2> vListR = vList.GetRange(1, vList.Count - 1);
    // And create vListL, which is all but the last element of vList
    // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
    List<Vector2> vListL = vList.GetRange(0, vList.Count - 1);
    // The result is the Lerp of these two shorter Lists
    Vector2 res = Lerp(Bezier(u, vListL), Bezier(u, vListR), u);
    return (res);
}

// This version allows an Array or a series of Vector2s as input
static public Vector2 Bezier( float u, params Vector2[] vecs ) {
    return (Bezier(u, new List<Vector2>(vecs)));
}

// The same two functions for float
static public float Bezier( float u, List<float> vList ) {
    // If there is only one element in vList, return it
    if (vList.Count == 1) {
        return (vList[0]);
    }
    // Otherwise, create vListR, which is all but the 0th element of vList
    // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
    List<float> vListR = vList.GetRange(1, vList.Count - 1);
    // And create vListL, which is all but the last element of vList
    // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
    List<float> vListL = vList.GetRange(0, vList.Count - 1);
    // The result is the Lerp of these two shorter Lists
    float res = Lerp(Bezier(u, vListL), Bezier(u, vListR), u);
return ( res );

// This version allows an Array or a series of floats as input
static public float Bezier( float u, params float[] vecs ) {
    return ( Bezier( u, new List<float>(vecs) ) );
}

// The same two functions for Quaternion
static public Quaternion Bezier( float u, List<Quaternion> vList ) {
    // If there is only one element in vList, return it
    if ( vList.Count == 1 ) {
        return ( vList[0] );
    }
    // Otherwise, create vListR, which is all but the 0th element of vList
    // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
    List<Quaternion> vListR = vList.GetRange( 1, vList.Count - 1 );
    // And create vListL, which is all but the last element of vList
    // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
    List<Quaternion> vListL = vList.GetRange( 0, vList.Count - 1 );
    // The result is the Slerp of these two shorter Lists
    // It's possible that Quaternion.Slerp may clamp u to [0..1] :(
    Quaternion res = Quaternion.Slerp ( Bezier(u, vListL), Bezier(u, vListR), u );
    return ( res );
}

// This version allows an Array or a series of floats as input
static public Quaternion Bezier( float u, params Quaternion[] vecs ) {
    return ( Bezier( u, new List<Quaternion>(vecs) ) );
}

/******************************* Trace & Logging Functions ******************************/

static public void tr(params object[] objs) {
    string s = objs[0].ToString();
    for ( int i = 1; i < objs.Length; i++ ) {
        s += "\t" + objs[i].ToString();
    }
    print (s);
}

/******************************* Math Functions ***************************************/

static public float RoundToPlaces( float f, int places=2 ) {
    float mult = Mathf.Pow(10, places);
    f *= mult;
    f = Mathf.Round ( f );
    f /= mult;
    return (f);
}

static public string AddCommasToNumber( float f, int places=2 ) {
    int n = Mathf.RoundToInt(f);
    f -= n;
    f = RoundToPlaces(f, places);
    string str = AddCommasToNumber( n );
    str += ","+(f*Mathf.Pow(10, places));
    return ( str );
}
```csharp
static public string AddCommasToNumber(int n) {
    int rem;
    int div;
    string res = "";
    string remstr;
    while (n > 0) {
        rem = n % 1000;
        div = n / 1000;
        remstr = rem.ToString();
        while (div > 0 && remstr.Length < 3) {
            remstr = "0" + remstr;
        }
        // NOTE: It is somewhat faster to use a StringBuilder or a List<String> which
        // is then concatenated using String.Join().
        if (res == "") {
            res = remstr;
        } else {
            res = remstr + "," + res.ToString();
        }
        n = div;
    }
    if (res == "") res = "0";
    return (res);
}

 public class EasingCachedCurve {
    public List<string> curves = new List<string>();
    public List<float> mods = new List<float>();
}

 public class Easing {
    static public string Linear = ",Linear|";
    static public string In = ",In|";
    static public string Out = ",Out|";
    static public string InOut = ",InOut|";
    static public string Sin = ",Sin|";
    static public string SinIn = ",SinIn|";
    static public string SinOut = ",SinOut|";

    static public Dictionary<string, EasingCachedCurve> cache;
    // This is a cache for the information contained in the complex strings
    // that can be passed into the Ease function. The parsing of these
    // strings is most of the effort of the Ease function, so each time one
    // is parsed, the result is stored in the cache to be recalled much
    // faster than a parse would take.
    // Need to be careful of memory leaks, which could be a problem if several
    // million unique easing parameters are called

    static public float Ease(float u, params string[] curveParams) {
        // Set up the cache for curves
        if (cache == null) {
            cache = new Dictionary<string, EasingCachedCurve>();
        }
        float u2 = u;
        foreach (string curve in curveParams) {
            // Check to see if this curve is already cached
            if (!cache.ContainsKey(curve)) {
                // If not, parse and cache it
                EaseParse(curve);
            }
        }
    }
```
// Call the cached curve
    u2 = EaseP( u2, cache[curve] );
}
return( u2 );
}

static private void EaseParse( string curveIn ) {
    EasingCachedCurve ecc = new EasingCachedCurve();
    // It's possible to pass in several comma-separated curves
    string[] curves = curveIn.Split(',');
    foreach (string curve in curves) {
        if (curve == "") continue;
        // Split each curve on | to find curve and mod
        string[] curveA = curve.Split('|');
        ecc.curves.Add( curveA[0] );
        if (curveA.Length == 1 || curveA[1] == "") {
            ecc.mods.Add( float.NaN );
        } else {
            float parseRes;
            if (float.TryParse( curveA[1], out parseRes ) ) {
                ecc.mods.Add( parseRes );
            } else {
                ecc.mods.Add( float.NaN );
            }
        }
    }
    cache.Add(curveIn, ecc);
}

static public float Ease( float u, string curve, float mod ) {
    return( EaseP( u, curve, mod ) );
}

static private float EaseP( float u, EasingCachedCurve ec ) {
    float u2 = u;
    for (int i=0; i<ec.curves.Count; i++) {
        u2 = EaseP( u2, ec.curves[i], ec.mods[i] );
    }
    return( u2 );
}

static private float EaseP( float u, string curve, float mod ) {
    float u2 = u;
    switch (curve) {
        case "In":
            if (float.IsNaN(mod)) mod = 2;
            u2 = Mathf.Pow(u, mod);
            break;
        case "Out":
            if (float.IsNaN(mod)) mod = 2;
            u2 = 1 - Mathf.Pow( 1-u, mod );
            break;
        case "InOut":
            if (float.IsNaN(mod)) mod = 2;
            if ( u <= 0.5f ) {
                u2 = 0.5f * Mathf.Pow( u*2, mod );
            } else {
                u2 = 0.5f + 0.5f * ( 1 - Mathf.Pow( 1-(2*(u-0.5f)), mod ) );
            }
            break;
    }
case "Sin":
    if (float.IsNaN(mod)) mod = 0.15f;
    u2 = u + mod * Mathf.Sin( 2*Mathf.PI*u );
    break;

case "SinIn":
    // mod is ignored for SinIn
    u2 = 1 - Mathf.Cos( u*Mathf.PI * 0.5f );
    break;

case "SinOut":
    // mod is ignored for SinOut
    u2 = Mathf.Sin( u*Mathf.PI * 0.5f );
    break;

case "Linear":
    default:
    // u2 already equals u
    break;

    return( u2 );
}