Bartok.cs

```
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()

```

```
65          layout = GetComponent<BartokLayout>();   // Get the Layout
66          layout.ReadLayout(layoutXML.text); // Pass LayoutXML to it
67
68          drawPile = UpgradeCardsList( deck.cards );
69          LayoutGame();
70      }
71
72      // UpgradeCardsList casts the Cards in lCD to be CardBartoks
73      // Of course, they were all along, but this lets Unity know it
74      List<CardBartok> UpgradeCardsList(List<Card> lCD) {
75          List<CardBartok> lCB = new List<CardBartok>();
76          foreach( Card tCD in lCD ) {
77              lCB.Add ( tCD as CardBartok );
78          }
79          return( lCB );
80      }
81
82      // Position all the cards in the drawPile properly
83      public void ArrangeDrawPile() {
84          CardBartok tCB;
85
86          for (int i=0; i<drawPile.Count; i++) {
87              tCB = drawPile[i];
88              tCB.transform.parent = layoutAnchor;
89              tCB.transform.localPosition = layout.drawPile.pos;
90              // Rotation should start at 0
91              tCB.faceUp = false;
92              tCB.SetSortingLayerName(layout.drawPile.layerName);
93              tCB.SetSortOrder(-i*4); // Order them front-to-back
94              tCB.state = CBState.drawpile;
95          }
96
97      }
98
99      void LayoutGame() {
100         // Create an empty GameObject to serve as an anchor for the tableau //1
101         if (layoutAnchor == null) {
102             GameObject tGO = new GameObject("_LayoutAnchor");
103             // ^ Create an empty GameObject named _LayoutAnchor in the Hierarchy
104             layoutAnchor = tGO.transform;                  // Grab its Transform
105             layoutAnchor.transform.position = layoutCenter;      // Position it
106         }
107
108         // Position the drawPile cards
109         ArrangeDrawPile();
110
111         // Set up the players
112         Player pl;
113         players = new List<Player>();
114         foreach (SlotDef tSD in layout.slotDefs) {
115             pl = new Player();
116             pl.handSlotDef = tSD;
117             players.Add(pl);
118             pl.playerNum = players.Count;
119         }
120         players[0].type = PlayerType.human; // Make the 0th player human
121
122         CardBartok tCB;
123
124         // Deal 7 cards to each player
125         for (int i=0; i<numStartingCards; i++) {
126             for (int j=0; j<4; j++) { // There are always 4 players
127                tCB = Draw (); // Draw a card
128                 // Stagger the draw time a bit. Remember order of operations.
```

```
129                 tCB.timeStart = Time.time + drawTimeStagger * ( i*4 + j );
130                 // ^ By setting the timeStart before calling AddCard, we
131                 //  override the automatic setting of timeStart by
132                 //  CardBartok.MoveTo().
133 
134                 // Add the card to the player's hand. The modulus (%) makes it
135                 //  a number from 0 to 3
136                 players[ (j+1)%4 ].AddCard(tCB);
137             }
138         }
139 
140         // Call Bartok.DrawFirstTarget() when the other cards are done.
141         Invoke("DrawFirstTarget", drawTimeStagger * (numStartingCards*4+4) );
142     }
143 
144     public void DrawFirstTarget() {
145         // Flip up the target card in the middle
146         CardBartok tCB = MoveToTarget( Draw () );
147         // Set the CardBartok to call CBCallback on this Bartok when it is done
148         tCB.reportFinishTo = this.gameObject;
149     }
150 
151     // This callback is used by the last card to be dealt at the beginning
152     // It is only used once per game.
153     public void CBCallback(CardBartok cb) {
154         // You sometimes want to have reporting of method calls liek this
155         Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.CBCallback()",cb.name);
156         StartGame();
157     }
158 
159     public void StartGame() {
160         // Pick the player to the left of the human to go first.
161         // (players[0] is the human)
162         PassTurn(1);
163     }
164 
165     public void PassTurn(int num=-1) {
166         // If no number was passed in, pick the next player
167         if (num == -1) {
168             int ndx = players.IndexOf(CURRENT_PLAYER);
169             num = (ndx+1)%4;
170         }
171         int lastPlayerNum = -1;
172         if (CURRENT_PLAYER != null) {
173             lastPlayerNum = CURRENT_PLAYER.playerNum;
174             // Check for Game Over and need to reshuffle discards
175             if ( CheckGameOver() ) {
176                 return;
177             }
178         }
179         CURRENT_PLAYER = players[num];
180         phase = TurnPhase.pre;
181 
182         CURRENT_PLAYER.TakeTurn();
183 
184         // Move the TurnLight to shine on the new CURRENT_PLAYER
185         Vector3 lPos = CURRENT_PLAYER.handSlotDef.pos + Vector3.back*5;
186         turnLight.transform.position = lPos;
187 
188         // Report the turn passing
189         Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.PassTurn()","Old: "
              ➥+lastPlayerNum,"New: "+CURRENT_PLAYER.playerNum);
190     }
191 
192 
```

```
193     public bool CheckGameOver() {
194         // See if we need to reshuffle the discard pile into the draw pile
195         if (drawPile.Count == 0) {
196             List<Card> cards = new List<Card>();
197             foreach (CardBartok cb in discardPile) {
198                 cards.Add (cb);
199             }
200             discardPile.Clear();
201             Deck.Shuffle( ref cards );
202             drawPile = UpgradeCardsList(cards);
203             ArrangeDrawPile();
204         }
205 
206         // Check to see if the current player has won
207         if (CURRENT_PLAYER.hand.Count == 0) {
208             // The current player has won!
209             if (CURRENT_PLAYER.type == PlayerType.human) {
210                 GTGameOver.GetComponent<GUIText>().text = "You Won!";
211                 GTRoundResult.GetComponent<GUIText>().text = "";
212             }  else {
213                 GTGameOver.GetComponent<GUIText>().text = "Game Over";
214                 GTRoundResult.GetComponent<GUIText>().text = "Player "
                      ➥+CURRENT_PLAYER.playerNum+" won";
215             }
216             GTGameOver.SetActive(true);
217             GTRoundResult.SetActive(true);
218             phase = TurnPhase.gameOver;
219             Invoke("RestartGame", 1);
220             return(true);
221         }
222 
223         return(false);
224     }
225 
226     public void RestartGame() {
227         CURRENT_PLAYER = null;
228         Application.LoadLevel("__Bartok_Scene_0");
229     }
230 
231     public CardBartok MoveToTarget(CardBartok tCB) {
232         tCB.timeStart = 0;
233         tCB.MoveTo(layout.discardPile.pos+Vector3.back);
234         tCB.state = CBState.toTarget;
235         tCB.faceUp = true;
236         tCB.SetSortingLayerName("10");//layout.target.layerName);
237         tCB.eventualSortLayer = layout.target.layerName;
238         if (targetCard != null) {
239             MoveToDiscard(targetCard);
240         }
241         targetCard = tCB;
242 
243         return(tCB);
244     }
245 
246     public CardBartok MoveToDiscard(CardBartok tCB) {
247         //Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.MoveToDiscard()",tCB.name);
248         tCB.state = CBState.discard;
249         discardPile.Add ( tCB );
250         tCB.SetSortingLayerName(layout.discardPile.layerName);
251         tCB.SetSortOrder( discardPile.Count*4 );
252         tCB.transform.localPosition = layout.discardPile.pos + Vector3.back/2;
253 
254         return(tCB);
255     }
256 
```

```
257     // The Draw function will pull a single card from the drawPile and return it
258     public CardBartok Draw() {
259         CardBartok cd = drawPile[0];     // Pull the 0th CardProspector
260         drawPile.RemoveAt(0);            // Then remove it from List<> drawPile
261         return(cd);                        // And return it
262     }
263 
264     // ValidPlay verifies that the card chosen can be played on the discard pile
265     public bool ValidPlay(CardBartok cb) {
266         // It's a valid play if the rank is the same
267         if (cb.rank == targetCard.rank) return(true);
268 
269         // It's a valid play if the suit is the same
270         if (cb.suit == targetCard.suit) {
271             return(true);
272         }
273         // Otherwise, return false
274         return(false);
275     }
276 
277     /* Now is a good time to comment out this testing code
278     // This Update method is used to test passing cards to players
279     void Update() {
280         if (Input.GetKeyDown(KeyCode.Alpha1)) {
281             players[0].AddCard(Draw ());
282         }
283         if (Input.GetKeyDown(KeyCode.Alpha2)) {
284             players[1].AddCard(Draw ());
285         }
286         if (Input.GetKeyDown(KeyCode.Alpha3)) {
287             players[2].AddCard(Draw ());
288         }
289         if (Input.GetKeyDown(KeyCode.Alpha4)) {
290             players[3].AddCard(Draw ());
291         }
292     }
293     */
294 
295 
296     public void CardClicked(CardBartok tCB) {
297         // If it's not the human's turn, don't respond
298         if (CURRENT_PLAYER.type != PlayerType.human) return;
299         // If the game is waiting on a card to move, don't respond
300         if (phase == TurnPhase.waiting) return;
301 
302         // Act differently based on whether it was a card in hand or on the drawPile that
              ➥was clicked
303         switch (tCB.state) {
304         case CBState.drawpile:
305             // Draw the top card, not necessarily the one clicked.
306             CardBartok cb = CURRENT_PLAYER.AddCard( Draw() );
307             cb.callbackPlayer = CURRENT_PLAYER;
308             Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.CardClicked()","Draw",
                  ➥cb.name);
309             phase = TurnPhase.waiting;
310             break;
311         case CBState.hand:
312             // Check to see whether the card is valid
313             if (ValidPlay(tCB)) {
314                 CURRENT_PLAYER.RemoveCard(tCB);
315                 MoveToTarget(tCB);
316                 tCB.callbackPlayer = CURRENT_PLAYER;
317                 Utils.tr (Utils.RoundToPlaces(Time.time), "Bartok.CardClicked()","Play",
                      ➥tCB.name,targetCard.name+" is target");
318                 phase = TurnPhase.waiting;
319             } else {
320 
```

```
321                 // Just ignore it
322                 Utils.tr (Utils.RoundToPlaces(Time.time),
323 "Bartok.CardClicked()","Attempted to Play",tCB.name,targetCard.name+" is target");
324             }
325             break;
326 
327         }
328     }
329 }
```

Utils.cs

```
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 {

//============================== Bounds Functions ============================\

    // 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) {
        // Create an empty Bounds b
        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 );
        }
    }
```

```
65      // This is the private static field that camBounds uses
66      static private Bounds _camBounds;
67
68      public static void SetCameraBounds(Camera cam=null) {
69          // If no Camera was passed in, use the main Camera
70          if (cam == null) cam = Camera.main;
71          // This makes a couple important assumptions about the camera!:
72          //   1. The camera is Orthographic
73          //   2. The camera is at a rotation of R:[0,0,0]
74
75          // Make Vector3s at the topLeft and bottomRight of the Screen coords
76          Vector3 topLeft = new Vector3( 0, 0, 0 );
77          Vector3 bottomRight = new Vector3( Screen.width, Screen.height, 0 );
78
79          // Convert these to world coordinates
80          Vector3 boundTLN = cam.ScreenToWorldPoint( topLeft );
81          Vector3 boundBRF = cam.ScreenToWorldPoint( bottomRight );
82
83          // Adjust the z to be at the near and far Camera clipping planes
84          boundTLN.z += cam.nearClipPlane;
85          boundBRF.z += cam.farClipPlane;
86
87          // Find the center of the Bounds
88          Vector3 center = (boundTLN + boundBRF)/2f;
89          _camBounds = new Bounds( center, Vector3.zero );
90          // Expand _camBounds to encapsulate the extents.
91          _camBounds.Encapsulate( boundTLN );
92          _camBounds.Encapsulate( boundBRF );
93      }
94
95
96
97      // Test to see whether Bounds are on screen.
98      public static Vector3 ScreenBoundsCheck(Bounds bnd, BoundsTest test =
          ➥BoundsTest.center) {
99          // Call the more generic BoundsInBoundsCheck with camBounds as bigB
100         return( BoundsInBoundsCheck( camBounds, bnd, test ) );
101     }
102
103     // Tests to see whether lilB is inside bigB
104     public static Vector3 BoundsInBoundsCheck( Bounds bigB, Bounds lilB, BoundsTest test
          ➥= BoundsTest.onScreen ) {
105         // Get the center of lilB
106         Vector3 pos = lilB.center;
107
108         // Initialize the offset at [0,0,0]
109         Vector3 off = Vector3.zero;
110
111         switch (test) {
112 // The center test determines what off (offset) would have to be applied to lilB to move
      ➥its center back inside bigB
113         case BoundsTest.center:
114             // if the center is contained, return Vector3.zero
115             if ( bigB.Contains( pos ) ) {
116                 return( Vector3.zero );
117             }
118             // if not contained, find the offset
119             if (pos.x > bigB.max.x) {
120                 off.x = pos.x - bigB.max.x;
121             } else if (pos.x < bigB.min.x) {
122                 off.x = pos.x - bigB.min.x;
123             }
124             if (pos.y > bigB.max.y) {
125                 off.y = pos.y - bigB.max.y;
126             } else if (pos.y < bigB.min.y) {
127                 off.y = pos.y - bigB.min.y;
128             }
```

```
            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
        case BoundsTest.onScreen:
            // find whether bigB contains all of lilB
            if ( bigB.Contains( lilB.min ) && bigB.Contains( lilB.max ) ) {
                return( Vector3.zero );
            }
            // if not, find the offset
            if (lilB.max.x > bigB.max.x) {
                off.x = lilB.max.x - bigB.max.x;
            }  else  if (lilB.min.x < bigB.min.x) {
                off.x = lilB.min.x - bigB.min.x;
            }
            if (lilB.max.y > bigB.max.y) {
                off.y = lilB.max.y - bigB.max.y;
            }  else  if (lilB.min.y < bigB.min.y) {
                off.y = lilB.min.y - bigB.min.y;
            }
            if (lilB.max.z > bigB.max.z) {
                off.z = lilB.max.z - bigB.max.z;
            }  else  if (lilB.min.z < bigB.min.z) {
                off.z = lilB.min.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
        case BoundsTest.offScreen:
            // find whether bigB contains any of lilB
            bool cMin = bigB.Contains( lilB.min );
            bool cMax = bigB.Contains( lilB.max );
            if ( cMin || cMax ) {
                return( Vector3.zero );
            }
            // if not, find the offset
            if (lilB.min.x > bigB.max.x) {
                off.x = lilB.min.x - bigB.max.x;
            }  else  if (lilB.max.x < bigB.min.x) {
                off.x = lilB.max.x - bigB.min.x;
            }
            if (lilB.min.y > bigB.max.y) {
                off.y = lilB.min.y - bigB.max.y;
            }  else  if (lilB.max.y < bigB.min.y) {
                off.y = lilB.max.y - bigB.min.y;
            }
            if (lilB.min.z > bigB.max.z) {
                off.z = lilB.min.z - bigB.max.z;
            }  else  if (lilB.max.z < bigB.min.z) {
                off.z = lilB.max.z - bigB.min.z;
            }
            return( off );

        }

        return( Vector3.zero );
    }
```

```
//============================ Transform Functions ============================\

    // This function will iteratively climb up the transform.parent tree
    //   until it either finds a parent with a tag != "Untagged" or no parent
    public static GameObject FindTaggedParent(GameObject go) {
        // If this gameObject has a tag
        if (go.tag != "Untagged") {
            // then return this gameObject
            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
            // So return null
            return( null );
        }
        // Otherwise, recursively climb up the tree
        return( FindTaggedParent( go.transform.parent.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 ============================

    // Returns a list of all Materials in this GameObject or its children
    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 ============================

    // The standard Vector Lerp functions in Unity don't allow for extrapolation
    //   (which is input u values <0 or >1), so we need to write our own functions
    static public Vector3 Lerp (Vector3 vFrom, Vector3 vTo, float u) {
        Vector3 res = (1-u)*vFrom + u*vTo;
        return( res );
    }
    // The same function for Vector2
    static public Vector2 Lerp (Vector2 vFrom, Vector2 vTo, float u) {
        Vector2 res = (1-u)*vFrom + u*vTo;
        return( res );
    }
    // The same function for float
    static public float Lerp (float vFrom, float vTo, float u) {
        float res = (1-u)*vFrom + u*vTo;
        return( res );
    }
```

```
//============================ Béier Curves ============================

    // 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 Utils.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 );
```

```
321         return( res );
322     }
323 
324     // This version allows an Array or a series of floats as input
325     static public float Bezier( float u, params float[] vecs ) {
326         return( Bezier( u, new List<float>(vecs) ) );
327     }
328 
329 
330     // The same two functions for Quaternion
331     static public Quaternion Bezier( float u, List<Quaternion> vList ) {
332         // If there is only one element in vList, return it
333         if (vList.Count == 1) {
334             return( vList[0] );
335         }
336         // Otherwise, create vListR, which is all but the 0th element of vList
337         // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
338         List<Quaternion> vListR =  vList.GetRange(1, vList.Count-1);
339         // And create vListL, which is all but the last element of vList
340         // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
341         List<Quaternion> vListL = vList.GetRange(0, vList.Count-1);
342         // The result is the Slerp of these two shorter Lists
343         // It's possible that Quaternion.Slerp may clamp u to [0..1] :(
344         Quaternion res = Quaternion.Slerp( Bezier(u, vListL), Bezier(u, vListR), u );
345         return( res );
346     }
347 
348     // This version allows an Array or a series of floats as input
349     static public Quaternion Bezier( float u, params Quaternion[] vecs ) {
350         return( Bezier( u, new List<Quaternion>(vecs) ) );
351     }
352 
353 
354 
355     //============================ Trace & Logging Functions ============================
356 
357     static public void tr(params object[] objs) {
358         string s = objs[0].ToString();
359         for (int i=1; i<objs.Length; i++) {
360             s += "\t"+objs[i].ToString();
361         }
362         print (s);
363     }
364 
365 
366 
367     //============================ Math Functions ============================
368 
369     static public float RoundToPlaces(float f, int places=2) {
370         float mult = Mathf.Pow(10,places);
371         f *= mult;
372         f = Mathf.Round (f);
373         f /= mult;
374         return(f);
375     }
376 
377     static public string AddCommasToNumber(float f, int places=2) {
378         int n = Mathf.RoundToInt(f);
379         f -= n;
380         f = RoundToPlaces(f,places);
381         string str = AddCommasToNumber( n );
382         str += "."+(f*Mathf.Pow(10,places));
383         return( str );
384     }
```

```
385     static public string AddCommasToNumber(int n) {
386         int rem;
387         int div;
388         string res = "";
389         string rems;
390         while (n>0) {
391             rem = n % 1000;
392             div = n / 1000;
393             rems = rem.ToString();
394 
395             while (div>0 && rems.Length<3) {
396                 rems = "0"+rems;
397             }
398             // NOTE: It is somewhat faster to use a StringBuilder or a List<String> which
                  is then concatenated using String.Join().
399             if (res == "") {
400                 res = rems;
401             }  else {
402                 res = rems + "," + res.ToString();
403             }
404             n = div;
405         }
406         if (res == "") res = "0";
407         return( res );
408     }
409 }
410 
411 
412 
413 //============================ Easing Classes ============================
414 [System.Serializable]
415 public class EasingCachedCurve {
416     public List<string>        curves =    new List<string>();
417     public List<float>        mods =         new List<float>();
418 }
419 
420 public class Easing {
421     static public string Linear =         ",Linear|";
422     static public string In =             ",In|";
423     static public string Out =            ",Out|";
424     static public string InOut =         ",InOut|";
425     static public string Sin =            ",Sin|";
426     static public string SinIn =        ",SinIn|";
427     static public string SinOut =        ",SinOut|";
428 
429     static public Dictionary<string,EasingCachedCurve> cache;
430     // This is a cache for the information contained in the complex strings
431     //   that can be passed into the Ease function. The parsing of these
432     //   strings is most of the effort of the Ease function, so each time one
433     //   is parsed, the result is stored in the cache to be recalled much
434     //   faster than a parse would take.
435     // Need to be careful of memory leaks, which could be a problem if several
436     //   million unique easing parameters are called
437     static public float Ease( float u, params string[] curveParams ) {
438         // Set up the cache for curves
439         if (cache == null) {
440             cache = new Dictionary<string, EasingCachedCurve>();
441         }
442         float u2 = u;
443         foreach ( string curve in curveParams ) {
444             // Check to see if this curve is already cached
445             if (!cache.ContainsKey(curve)) {
446                 // If not, parse and cache it
447                 EaseParse(curve);
448             }
```

```
449             // Call the cached curve
450             u2 = EaseP( u2, cache[curve] );
451         }
452         return( u2 );
453     }
454 
455     static private void EaseParse( string curveIn ) {
456         EasingCachedCurve ecc = new EasingCachedCurve();
457         // It's possible to pass in several comma-separated curves
458         string[] curves = curveIn.Split(',');
459         foreach (string curve in curves) {
460             if (curve == "") continue;
461             // Split each curve on | to find curve and mod
462             string[] curveA = curve.Split('|');
463             ecc.curves.Add(curveA[0]);
464             if (curveA.Length == 1 || curveA[1] == "") {
465                 ecc.mods.Add(float.NaN);
466             }  else {
467                 float parseRes;
468                 if ( float.TryParse(curveA[1], out parseRes) ) {
469                     ecc.mods.Add( parseRes );
470                 }  else {
471                     ecc.mods.Add( float.NaN );
472                 }
473             }
474         }
475         cache.Add(curveIn, ecc);
476     }
477 
478     static public float Ease( float u, string curve, float mod ) {
479         return( EaseP( u, curve, mod ) );
480     }
481 
482     static private float EaseP( float u, EasingCachedCurve ec ) {
483         float u2 = u;
484         for (int i=0; i<ec.curves.Count; i++) {
485             u2 = EaseP( u2, ec.curves[i], ec.mods[i] );
486         }
487         return( u2 );
488     }
489 
490     static private float EaseP( float u, string curve, float mod ) {
491         float u2 = u;
492 
493         switch (curve) {
494         case "In":
495             if (float.IsNaN(mod)) mod = 2;
496             u2 = Mathf.Pow(u, mod);
497             break;
498 
499         case "Out":
500             if (float.IsNaN(mod)) mod = 2;
501             u2 = 1 - Mathf.Pow( 1-u, mod );
502             break;
503 
504         case "InOut":
505             if (float.IsNaN(mod)) mod = 2;
506             if ( u <= 0.5f ) {
507                 u2 = 0.5f * Mathf.Pow( u*2, mod );
508             }  else {
509                 u2 = 0.5f + 0.5f * (  1 - Mathf.Pow( 1-(2*(u-0.5f)), mod )  );
510             }
511             break;
512 
```

```
513             case "Sin":
514                 if (float.IsNaN(mod)) mod = 0.15f;
515                 u2 = u + mod * Mathf.Sin( 2*Mathf.PI*u );
516                 break;
517 
518             case "SinIn":
519                 // mod is ignored for SinIn
520                 u2 = 1 - Mathf.Cos( u * Mathf.PI * 0.5f );
521                 break;
522 
523             case "SinOut":
524                 // mod is ignored for SinOut
525                 u2 = Mathf.Sin( u * Mathf.PI * 0.5f );
526                 break;
527 
528             case "Linear":
529             default:
530                 // u2 already equals u
531                 break;
532             }
533 
534             return( u2 );
535         }
536 
537 }
```