TargetCamera.cs

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

public class TargetCamera : MonoBehaviour {
    static public TargetCamera S;

    public bool               editMode = true;
    public GameObject        fpCamera; // First-person Camera
    // Maximum deviation in Shot.position allowed
    public float            maxPosDeviation = 1f;
    // Maximum deviation in Shot.target allowed
    public float            maxTarDeviation = 0.5f;
    // Easing for these deviations
    public string            deviationEasing = Easing.Out;
    public float            passingAccuracy = 0.7f;

    public bool               checkToDeletePlayerPrefs = false;

    public bool               _________________;

    public Rect             camRectNormal; // Pulled from camera.rect
    public int                shotNum;
    public GUIText            shotCounter, shotRating;
    public GUITexture        checkMark;
    public Shot               lastShot;
    public int                numShots;
    public Shot[]            playerShots;
    public float[]            playerRatings;
    public GUITexture        whiteOut;

    void Awake() {
        S = this;
    }

    void Start() {
        // Find the GUI Components
        GameObject go = GameObject.Find("ShotCounter");
        shotCounter = go.GetComponent<GUIText>();
        go = GameObject.Find("ShotRating");
        shotRating = go.GetComponent<GUIText>();
        go = GameObject.Find("_Check_64");
        checkMark = go.GetComponent<GUITexture>();
        go = GameObject.Find ("WhiteOut");
        whiteOut = go.GetComponent<GUITexture>();
        // Hide the checkMark and whiteOut
        checkMark.enabled = false;
        whiteOut.enabled = false;

        // Load all the shots from PlayerPrefs
        Shot.LoadShots();
        // If there were shots stored in PlayerPrefs
        if (Shot.shots.Count>0) {
            shotNum = 0;
            ResetPlayerShotsAndRatings();
            ShowShot(Shot.shots[shotNum]);
        }

        // Hide the cursor (Note: this doesn't work in the Unity Editor unless
        //  the Game pane is set to Maximize on Play.)
        Cursor.visible = false;

        camRectNormal = GetComponent<Camera>().rect;
    }
```

```
65      void ResetPlayerShotsAndRatings() {
66          numShots = Shot.shots.Count;
67          // Initialize playerShots & playerRatings with null values
68          playerShots = new Shot[numShots];
69          playerRatings = new float[numShots];
70      }
71
72      void Update () {
73          Shot sh;
74          // Mouse Input
75          // If Left or Right mouse button is pressed this frame...
76          if (Input.GetMouseButtonDown(0) || Input.GetMouseButtonDown(1)) {
77              sh = new Shot();
78              // Grab the position and rotation of fpCamera
79              sh.position = fpCamera.transform.position;
80              sh.rotation = fpCamera.transform.rotation;
81              // Shoot a ray from the camera and see what it hits
82              Ray ray = new Ray(sh.position, fpCamera.transform.forward);
83              RaycastHit hit;
84              if ( Physics.Raycast(ray, out hit) ) {
85                  sh.target = hit.point;
86              }
87
88              if (editMode) {
89                  if (Input.GetMouseButtonDown(0)) {
90                      // Left button records a new shot
91                      Shot.shots.Add(sh);
92                      shotNum = Shot.shots.Count-1;
93                  }  else {
94                      // Right button replaces the current shot
95                      Shot.ReplaceShot(shotNum, sh);
96                      ShowShot(Shot.shots[shotNum]);
97                  }
98                  // Reset information about the player when editing shots
99                  ResetPlayerShotsAndRatings();
100             }  else {
101                 // Test this shot against the current Shot
102                 float acc = Shot.Compare( Shot.shots[shotNum], sh );
103                 lastShot = sh;
104                 playerShots[shotNum] = sh;
105                 playerRatings[shotNum] = acc;
106                 // Show the shot just taken by the player
107                 ShowShot(sh);
108                 // Return to the current shot after waiting 1 second
109                 Invoke("ShowCurrentShot",1);
110             }
111
112             // Play the shutter sound
113             this.GetComponent<AudioSource>().Play();
114         }
115
116         // Keyboard Input
117         // Use Q and E to cycle Shots
118         // Note: Either of these will throw an error if Shot.shots is empty.
119         if (Input.GetKeyDown(KeyCode.Q)) {
120             shotNum--;
121             if (shotNum < 0) shotNum = Shot.shots.Count-1;
122             ShowShot(Shot.shots[shotNum]);
123         }
124         if (Input.GetKeyDown(KeyCode.E)) {
125             shotNum++;
126             if (shotNum >= Shot.shots.Count) shotNum = 0;
127             ShowShot(Shot.shots[shotNum]);
128         }
```

```
129         // If in editMode & Left Shift is held down...
130         if (editMode && Input.GetKey(KeyCode.LeftShift)) {
131             // Use Shift-S to Save
132             if (Input.GetKeyDown(KeyCode.S)) {
133                 Shot.SaveShots();
134             }
135             // Use Shift-X to output XML to Console
136             if (Input.GetKeyDown(KeyCode.X)) {
137                 Utils.tr(Shot.XML);
138             }
139         }
140         // Hold Tab to maximize the Target window
141         if (Input.GetKeyDown(KeyCode.Tab)) {
142             // Maximize when Tab is pressed
143             GetComponent<Camera>().rect = new Rect(0,0,1,1);
144         }
145         if (Input.GetKeyUp(KeyCode.Tab)) {
146             // Return to normal when Tab is released
147             GetComponent<Camera>().rect = camRectNormal;
148         }
149 
150         // Update the GUITexts
151         shotCounter.text = (shotNum+1).ToString()+" of "+Shot.shots.Count;
152         if (Shot.shots.Count == 0) shotCounter.text = "No shots exist";
153         // ^ Shot.shots.Count doesn't require .ToString() because it is assumed
154         //  when the left side of the + operator is a string
155         if (playerRatings.Length > shotNum && playerShots[shotNum] != null) {
156             float rating = Mathf.Round(playerRatings[shotNum]*100f);
157             if (rating < 0) rating = 0;
158             shotRating.text = rating.ToString()+"%";
159             checkMark.enabled = (playerRatings[shotNum] > passingAccuracy);
160             // ^ the > comparison is used to generate true or false
161         }  else {
162             shotRating.text = "";
163             checkMark.enabled = false;
164         }
165 
166     }
167 
168     public void ShowShot(Shot sh) {
169         // Call WhiteOutTargetWindow() and let it handle its own timing
170         StartCoroutine( WhiteOutTargetWindow() );
171         // Position _TargetCamera with the Shot
172         this.transform.position = sh.position;
173         this.transform.rotation = sh.rotation;
174     }
175 
176     public void ShowCurrentShot() {
177         ShowShot(Shot.shots[shotNum]);
178     }
179 
180     // Another use for coroutines is to have a fire-and-forget function with a
181     //  delay in it as we've done here. WhiteOutTargetWindow() will enable
182     //  whiteOut, yield for 0.05 seconds, and then disable it. Compare this
183     //  method of delay to the Invoke("ShowCurrentShot",1f) used above
184     public IEnumerator WhiteOutTargetWindow() {
185         whiteOut.enabled = true;
186         yield return new WaitForSeconds(0.05f);
187         whiteOut.enabled = false;
188     }
189 
190 
191 
192 
```

```
193     // OnDrawGizmos() is called ANY time Gizmos need to be drawn, even when
194     //  Unity isn't playing!
195     public void OnDrawGizmos() {
196         List<Shot> shots = Shot.shots;
197         for (int i=0; i<shots.Count; i++) {
198             Gizmos.color = Color.green;
199             Gizmos.DrawWireSphere(shots[i].position, 0.5f);
200             Gizmos.color = Color.yellow;
201             Gizmos.DrawLine( shots[i].position, shots[i].target );
202             Gizmos.color = Color.red;
203             Gizmos.DrawWireSphere(shots[i].target, 0.25f);
204         }
205 
206         // If checkToDeletePlayerPrefs is checked
207         if (checkToDeletePlayerPrefs) {
208             Shot.DeleteShots(); // Delete all the shots
209             // Uncheck checkToDeletePlayerPrefs
210             checkToDeletePlayerPrefs = false;
211             shotNum = 0; // Set shotNum to 0
212         }
213 
214         // Show the player's last shot attempt
215         if (lastShot != null) {
216             Gizmos.color = Color.green;
217             Gizmos.DrawSphere(lastShot.position, 0.25f);
218             Gizmos.color = Color.white;
219             Gizmos.DrawLine( lastShot.position, lastShot.target );
220             Gizmos.color = Color.red;
221             Gizmos.DrawSphere(lastShot.target, 0.125f);
222         }
223     }
224 }
```

Prototools/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 }
```