

Utils.cs

```
1  using UnityEngine;
2  using System.Collections;
3  using System.Collections.Generic;
4
5  // This is actually OUTSIDE of the Utils Class
6  public enum BoundsTest {
7      center,          // Is the center of the GameObject on screen
8      onScreen,       // Are the bounds entirely on screen
9      offScreen       // Are the bounds entirely off screen
10 }
11
12 public class Utils : MonoBehaviour {
13
14     //===== Bounds Functions =====|
15
16     // Creates bounds that encapsulate of the two Bounds passed in.
17     public static Bounds BoundsUnion( Bounds b0, Bounds b1 ) {
18         // If the size of one of the bounds is Vector3.zero, ignore that one
19         if ( b0.size==Vector3.zero && b1.size!=Vector3.zero ) {
20             return( b1 );
21         } else if ( b0.size!=Vector3.zero && b1.size==Vector3.zero ) {
22             return( b0 );
23         } else if ( b0.size==Vector3.zero && b1.size==Vector3.zero ) {
24             return( b0 );
25         }
26         // Stretch b0 to include the b1.min and b1.max
27         b0.Encapsulate(b1.min);
28         b0.Encapsulate(b1.max);
29         return( b0 );
30     }
31
32     public static Bounds CombineBoundsOfChildren(GameObject go) {
33         // Create an empty Bounds b
34         Bounds b = new Bounds(Vector3.zero, Vector3.zero);
35         // If this GameObject has a Renderer Component...
36         if (go.GetComponent<Renderer>() != null) {
37             // Expand b to contain the Renderer's Bounds
38             b = BoundsUnion(b, go.GetComponent<Renderer>().bounds);
39         }
40         // If this GameObject has a Collider Component...
41         if (go.GetComponent<Collider>() != null) {
42             // Expand b to contain the Collider's Bounds
43             b = BoundsUnion(b, go.GetComponent<Collider>().bounds);
44         }
45         // Iterate through each child of this gameObject.transform
46         foreach( Transform t in go.transform ) {
47             // Expand b to contain their Bounds as well
48             b = BoundsUnion( b, CombineBoundsOfChildren( t.gameObject ) );
49         }
50
51         return( b );
52     }
53
54     // Make a static read-only public property camBounds
55     static public Bounds camBounds {
56         get {
57             // if _camBounds hasn't been set yet
58             if ( _camBounds.size == Vector3.zero ) {
59                 // SetCameraBounds using the default Camera
60                 SetCameraBounds();
61             }
62             return( _camBounds );
63         }
64     }
65 }
```

```

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 =
99     ↳BoundsTest.center) {
100     // Call the more generic BoundsInBoundsCheck with camBounds as bigB
101     return( BoundsInBoundsCheck( camBounds, bnd, test ) );
102 }
103
104 // Tests to see whether lilB is inside bigB
105 public static Vector3 BoundsInBoundsCheck( Bounds bigB, Bounds lilB, BoundsTest test
106     ↳BoundsTest.onScreen ) {
107     // Get the center of lilB
108     Vector3 pos = lilB.center;
109
110     // Initialize the offset at [0,0,0]
111     Vector3 off = Vector3.zero;
112
113     switch (test) {
114 // The center test determines what off (offset) would have to be applied to lilB to move
115 // -its center back inside bigB
116     case BoundsTest.center:
117         // if the center is contained, return Vector3.zero
118         if ( bigB.Contains( pos ) ) {
119             return( Vector3.zero );
120         }
121         // if not contained, find the offset
122         if (pos.x > bigB.max.x) {
123             off.x = pos.x - bigB.max.x;
124         } else if (pos.x < bigB.min.x) {
125             off.x = pos.x - bigB.min.x;
126         }
127         if (pos.y > bigB.max.y) {
128             off.y = pos.y - bigB.max.y;
129         } else if (pos.y < bigB.min.y) {
130             off.y = pos.y - bigB.min.y;
131         }
132     }

```

```

129         if ( pos.z > bigB.max.z ) {
130             off.z = pos.z - bigB.max.z;
131         } else if ( pos.z < bigB.min.z ) {
132             off.z = pos.z - bigB.min.z;
133         }
134         return( off );
135
136 // The onScreen test determines what off would have to be applied to keep all of lilB
// -inside bigB
137         case BoundsTest.onScreen:
138             // find whether bigB contains all of lilB
139             if ( bigB.Contains( lilB.min ) && bigB.Contains( lilB.max ) ) {
140                 return( Vector3.zero );
141             }
142             // if not, find the offset
143             if ( lilB.max.x > bigB.max.x ) {
144                 off.x = lilB.max.x - bigB.max.x;
145             } else if ( lilB.min.x < bigB.min.x ) {
146                 off.x = lilB.min.x - bigB.min.x;
147             }
148             if ( lilB.max.y > bigB.max.y ) {
149                 off.y = lilB.max.y - bigB.max.y;
150             } else if ( lilB.min.y < bigB.min.y ) {
151                 off.y = lilB.min.y - bigB.min.y;
152             }
153             if ( lilB.max.z > bigB.max.z ) {
154                 off.z = lilB.max.z - bigB.max.z;
155             } else if ( lilB.min.z < bigB.min.z ) {
156                 off.z = lilB.min.z - bigB.min.z;
157             }
158             return( off );
159
160 // The offScreen test determines what off would need to be applied to move any tiny part
// -of lilB inside of bigB
161         case BoundsTest.offScreen:
162             // find whether bigB contains any of lilB
163             bool cMin = bigB.Contains( lilB.min );
164             bool cMax = bigB.Contains( lilB.max );
165             if ( cMin || cMax ) {
166                 return( Vector3.zero );
167             }
168             // if not, find the offset
169             if ( lilB.min.x > bigB.max.x ) {
170                 off.x = lilB.min.x - bigB.max.x;
171             } else if ( lilB.max.x < bigB.min.x ) {
172                 off.x = lilB.max.x - bigB.min.x;
173             }
174             if ( lilB.min.y > bigB.max.y ) {
175                 off.y = lilB.min.y - bigB.max.y;
176             } else if ( lilB.max.y < bigB.min.y ) {
177                 off.y = lilB.max.y - bigB.min.y;
178             }
179             if ( lilB.min.z > bigB.max.z ) {
180                 off.z = lilB.min.z - bigB.max.z;
181             } else if ( lilB.max.z < bigB.min.z ) {
182                 off.z = lilB.max.z - bigB.min.z;
183             }
184             return( off );
185
186     }
187
188     return( Vector3.zero );
189 }
190
191
192

```

```

193 //===== Transform Functions =====\
194
195 // This function will iteratively climb up the transform.parent tree
196 // until it either finds a parent with a tag != "Untagged" or no parent
197 public static GameObject FindTaggedParent(GameObject go) {
198     // If this gameObject has a tag
199     if (go.tag != "Untagged") {
200         // then return this gameObject
201         return(go);
202     }
203     // If there is no parent of this Transform
204     if (go.transform.parent == null) {
205         // We've reached the end of the line with no interesting tag
206         // So return null
207         return( null );
208     }
209     // Otherwise, recursively climb up the tree
210     return( FindTaggedParent( go.transform.parent.gameObject ) );
211 }
212 // This version of the function handles things if a Transform is passed in
213 public static GameObject FindTaggedParent(Transform t) {
214     return( FindTaggedParent( t.gameObject ) );
215 }
216
217
218
219
220 //===== Materials Functions =====
221
222 // Returns a List of all Materials in this GameObject or its children
223 static public Material[] GetAllMaterials( GameObject go ) {
224     List<Material> mats = new List<Material>();
225     if (go.GetComponent<Renderer>() != null) {
226         mats.Add(go.GetComponent<Renderer>().material);
227     }
228     foreach( Transform t in go.transform ) {
229         mats.AddRange( GetAllMaterials( t.gameObject ) );
230     }
231     return( mats.ToArray() );
232 }
233
234
235
236
237 //===== Linear Interpolation =====
238
239 // The standard Vector Lerp functions in Unity don't allow for extrapolation
240 // (which is input u values <0 or >1), so we need to write our own functions
241 static public Vector3 Lerp (Vector3 vFrom, Vector3 vTo, float u) {
242     Vector3 res = (1-u)*vFrom + u*vTo;
243     return( res );
244 }
245 // The same function for Vector2
246 static public Vector2 Lerp (Vector2 vFrom, Vector2 vTo, float u) {
247     Vector2 res = (1-u)*vFrom + u*vTo;
248     return( res );
249 }
250 // The same function for float
251 static public float Lerp (float vFrom, float vTo, float u) {
252     float res = (1-u)*vFrom + u*vTo;
253     return( res );
254 }
255
256

```

```

257 //===== Bézier Curves =====
258
259 // While most Bézier curves are 3 or 4 points, it is possible to have
260 // any number of points using this recursive function
261 // This uses the Utils.Lerp function because it needs to allow extrapolation
262 static public Vector3 Bezier( float u, List<Vector3> vList ) {
263     // If there is only one element in vList, return it
264     if (vList.Count == 1) {
265         return( vList[0] );
266     }
267     // Otherwise, create vListR, which is all but the 0th element of vList
268     // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
269     List<Vector3> vListR = vList.GetRange(1, vList.Count-1);
270     // And create vListL, which is all but the last element of vList
271     // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
272     List<Vector3> vListL = vList.GetRange(0, vList.Count-1);
273     // The result is the Lerp of these two shorter Lists
274     Vector3 res = Lerp( Bezier(u, vListL), Bezier(u, vListR), u );
275     return( res );
276 }
277
278 // This version allows an Array or a series of Vector3s as input
279 static public Vector3 Bezier( float u, params Vector3[] vecs ) {
280     return( Bezier( u, new List<Vector3>(vecs) ) );
281 }
282
283
284 // The same two functions for Vector2
285 static public Vector2 Bezier( float u, List<Vector2> vList ) {
286     // If there is only one element in vList, return it
287     if (vList.Count == 1) {
288         return( vList[0] );
289     }
290     // Otherwise, create vListR, which is all but the 0th element of vList
291     // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
292     List<Vector2> vListR = vList.GetRange(1, vList.Count-1);
293     // And create vListL, which is all but the last element of vList
294     // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
295     List<Vector2> vListL = vList.GetRange(0, vList.Count-1);
296     // The result is the Lerp of these two shorter Lists
297     Vector2 res = Lerp( Bezier(u, vListL), Bezier(u, vListR), u );
298     return( res );
299 }
300
301 // This version allows an Array or a series of Vector2s as input
302 static public Vector2 Bezier( float u, params Vector2[] vecs ) {
303     return( Bezier( u, new List<Vector2>(vecs) ) );
304 }
305
306
307 // The same two functions for float
308 static public float Bezier( float u, List<float> vList ) {
309     // If there is only one element in vList, return it
310     if (vList.Count == 1) {
311         return( vList[0] );
312     }
313     // Otherwise, create vListR, which is all but the 0th element of vList
314     // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
315     List<float> vListR = vList.GetRange(1, vList.Count-1);
316     // And create vListL, which is all but the last element of vList
317     // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
318     List<float> vListL = vList.GetRange(0, vList.Count-1);
319     // The result is the Lerp of these two shorter Lists
320     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 //===== Trace & Logging Functions =====
331
332 static public void tr(params object[] objs) {
333     string s = objs[0].ToString();
334     for (int i=1; i<objs.Length; i++) {
335         s += "\t"+objs[i].ToString();
336     }
337     print (s);
338 }
339
340
341 //===== Math Functions =====
342
343 static public float RoundToPlaces(float f, int places=2) {
344     float mult = Mathf.Pow(10,places);
345     f *= mult;
346     f = Mathf.Round (f);
347     f /= mult;
348     return(f);
349 }
350
351 static public string AddCommasToNumber(float f, int places=2) {
352     int n = Mathf.RoundToInt(f);
353     f -= n;
354     f = RoundToPlaces(f,places);
355     string str = AddCommasToNumber( n );
356     str += "."+(f*Mathf.Pow(10,places));
357     return( str );
358 }
359 static public string AddCommasToNumber(int n) {
360     int rem;
361     int div;
362     string res = "";
363     string rems;
364     while (n>0) {
365         rem = n % 1000;
366         div = n / 1000;
367         rems = rem.ToString();
368
369         while (div>0 && rems.Length<3) {
370             rems = "0"+rems;
371         }
372         // NOTE: It is somewhat faster to use a StringBuilder or a List<String> which
373         -is then concatenated using String.Join().
374         if (res == "") {
375             res = rems;
376         } else {
377             res = rems + "," + res.ToString();
378         }
379         n = div;
380     }
381     if (res == "") res = "0";
382     return( res );
383 }
384 }

```

```

385 //===== Easing Classes =====
386 [System.Serializable]
387 public class EasingCachedCurve {
388     public List<string>    curves =    new List<string>();
389     public List<float>    mods =      new List<float>();
390 }
391
392 public class Easing {
393     static public string Linear =      ",Linear|";
394     static public string In =          ",In|";
395     static public string Out =         ",Out|";
396     static public string InOut =      ",InOut|";
397     static public string Sin =        ",Sin|";
398     static public string SinIn =      ",SinIn|";
399     static public string SinOut =     ",SinOut|";
400
401     static public Dictionary<string,EasingCachedCurve> cache;
402     // This is a cache for the information contained in the complex strings
403     // that can be passed into the Ease function. The parsing of these
404     // strings is most of the effort of the Ease function, so each time one
405     // is parsed, the result is stored in the cache to be recalled much
406     // faster than a parse would take.
407     // Need to be careful of memory leaks, which could be a problem if several
408     // million unique easing parameters are called
409     static public float Ease( float u, params string[] curveParams ) {
410         // Set up the cache for curves
411         if (cache == null) {
412             cache = new Dictionary<string, EasingCachedCurve>();
413         }
414         float u2 = u;
415         foreach ( string curve in curveParams ) {
416             // Check to see if this curve is already cached
417             if (!cache.ContainsKey(curve)) {
418                 // If not, parse and cache it
419                 EaseParse(curve);
420             }
421             // Call the cached curve
422             u2 = EaseP( u2, cache[curve] );
423         }
424         return( u2 );
425     }
426
427     static private void EaseParse( string curveIn ) {
428         EasingCachedCurve ecc = new EasingCachedCurve();
429         // It's possible to pass in several comma-separated curves
430         string[] curves = curveIn.Split(',');
431         foreach (string curve in curves) {
432             if (curve == "") continue;
433             // Split each curve on | to find curve and mod
434             string[] curveA = curve.Split('|');
435             ecc.curves.Add(curveA[0]);
436             if (curveA.Length == 1 || curveA[1] == "") {
437                 ecc.mods.Add(float.NaN);
438             } else {
439                 float parseRes;
440                 if ( float.TryParse(curveA[1], out parseRes) ) {
441                     ecc.mods.Add( parseRes );
442                 } else {
443                     ecc.mods.Add( float.NaN );
444                 }
445             }
446         }
447         cache.Add(curveIn, ecc);
448     }

```

```

449
450 static public float Ease( float u, string curve, float mod ) {
451     return( EaseP( u, curve, mod ) );
452 }
453
454 static private float EaseP( float u, EasingCachedCurve ec ) {
455     float u2 = u;
456     for (int i=0; i<ec.curves.Count; i++) {
457         u2 = EaseP( u2, ec.curves[i], ec.mods[i] );
458     }
459     return( u2 );
460 }
461
462 static private float EaseP( float u, string curve, float mod ) {
463     float u2 = u;
464
465     switch (curve) {
466     case "In":
467         if (float.IsNaN(mod)) mod = 2;
468         u2 = Mathf.Pow(u, mod);
469         break;
470
471     case "Out":
472         if (float.IsNaN(mod)) mod = 2;
473         u2 = 1 - Mathf.Pow( 1-u, mod );
474         break;
475
476     case "InOut":
477         if (float.IsNaN(mod)) mod = 2;
478         if ( u <= 0.5f ) {
479             u2 = 0.5f * Mathf.Pow( u*2, mod );
480         } else {
481             u2 = 0.5f + 0.5f * ( 1 - Mathf.Pow( 1-(2*(u-0.5f)), mod ) );
482         }
483         break;
484
485     case "Sin":
486         if (float.IsNaN(mod)) mod = 0.15f;
487         u2 = u + mod * Mathf.Sin( 2*Mathf.PI*u );
488         break;
489
490     case "SinIn":
491         // mod is ignored for SinIn
492         u2 = 1 - Mathf.Cos( u * Mathf.PI * 0.5f );
493         break;
494
495     case "SinOut":
496         // mod is ignored for SinOut
497         u2 = Mathf.Sin( u * Mathf.PI * 0.5f );
498         break;
499
500     case "Linear":
501     default:
502         // u2 already equals u
503         break;
504     }
505
506     return( u2 );
507 }
508
509 }

```