```
1
     using UnityEngine;
     using System.Collections;
     using System.Collections.Generic;
 3
 5
     // This is actually OUTSIDE of the Utils Class
 6
     public enum BoundsTest {
 7
         center,
                         // Is the center of the GameObject on screen
 8
                       // Are the bounds entirely on screen
         onScreen,
                       // Are the bounds entirely off screen
 9
         offScreen
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 ) {
              // If the size of one of the bounds is Vector3.zero, ignore that one
18
19
              if ( b0.size==Vector3.zero && b1.size!=Vector3.zero ) {
                return( b1 );
else if ( b0.size!=Vector3.zero && b1.size==Vector3.zero ) {
20
21
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
              b0.Encapsulate(b1.min);
27
             b0.Encapsulate(b1.max);
28
29
              return( b0 );
30
31
32
         public static Bounds CombineBoundsOfChildren(GameObject go) {
33
              // Create an empty Bounds b
              Bounds b = new Bounds(Vector3.zero, Vector3.zero);
34
35
             // If this GameObject has a Renderer Component...
36
              if (go.GetComponent<Renderer>() != null) {
                  // Expand b to contain the Renderer's Bounds
37
38
                  b = BoundsUnion(b, go.GetComponent<Renderer>().bounds);
39
             // If this GameObject has a Collider Component...
if (go.GetComponent<Collider>() != null) {
40
41
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
                  b = BoundsUnion( b, CombineBoundsOfChildren( t.gameObject ) );
48
49
              }
50
51
              return( b );
52
         }
53
54
         // Make a static read-only public property camBounds
         static public Bounds camBounds {
55
56
              get {
57
                  // if camBounds hasn't been set yet
                  if (_camBounds.size == Vector3.zero) {
58
59
                      \overline{/}/ SetCameraBounds using the default Camera
                      SetCameraBounds();
60
61
                  return( _camBounds );
62
63
             }
         }
64
```

```
65
          // This is the private static field that camBounds uses
 66
          static private Bounds camBounds;
 67
 68
          public static void SetCameraBounds(Camera cam=null) {
              // If no Camera was passed in, use the main Camera
 69
 70
              if (cam == null) cam = Camera.main;
              // This makes a couple important assumptions about the camera!:
 71
                  1. The camera is Orthographic
 72
 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
              Vector3 boundTLN = cam.ScreenToWorldPoint( topLeft );
 80
              Vector3 boundBRF = cam.ScreenToWorldPoint( bottomRight );
 81
 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
              // Find the center of the Bounds
 87
 88
              Vector3 center = (boundTLN + boundBRF)/2f:
 89
               _camBounds = new Bounds( center, Vector3.zero );
              // Expand _camBounds to encapsulate the extents.
 90
              _camBounds.Encapsulate( boundTLN );
 91
              _camBounds.Encapsulate( boundBRF );
 92
 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
          // Tests to see whether lilB is inside bigB
103
          public static Vector3 BoundsInBoundsCheck( Bounds bigB, Bounds lilB, BoundsTest test
104
            →= BoundsTest.onScreen ) {
105
              // Get the center of lilB
106
              Vector3 pos = lilB.center;
107
108
              // Initialize the offset at [0,0,0]
              Vector3 off = Vector3.zero;
109
110
              switch (test) {
111
      // The center test determines what off (offset) would have to be applied to lilB to move
112
        ⇒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) {</pre>
                      off x = pos \cdot x - big B \cdot min \cdot x;
122
123
124
                  if (pos.y > bigB.max.y) {
125
                      off.y = pos.y - bigB.max.y;
                     else if (pos.y < bigB.min.y) {</pre>
126
127
                      off.y = pos.y - bigB.min.y;
128
```

```
129
                   if (pos.z > bigB.max.z) {
130
                       off.z = pos.z - bigB.max.z;
131
                      else if (pos.z < bigB.min.z) {</pre>
                       off.z = pos.z - bigB.min.z;
132
133
134
                   return( off );
135
      // The onScreen test determines what off would have to be applied to keep all of lilB
136
        ⇒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) {
                       off.x = lilB.max.x - bigB.max.x;
144
                     else if (lilB.min.x < bigB.min.x) {</pre>
145
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;
                      else if (lilB.min.y < bigB.min.y) {</pre>
150
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) {</pre>
                       off.z = lilB.min.z - bigB.min.z;
156
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 );
                   bool cMax = bigB.Contains( lilB.max );
164
                   if ( cMin || cMax ) {
    return( Vector3.zero );
165
166
167
168
                   // if not, find the offset
                   if (lilB.min.x > bigB.max.x) {
169
170
                       off.x = lilB.min.x - bigB.max.x;
                      else if (lilB.max.x < bigB.min.x) {</pre>
171
                       off.x = lilB.max.x - bigB.min.x;
172
173
174
                   if (lilB.min.y > bigB.max.y) {
175
                       off.y = lilB.min.y - bigB.max.y;
                      else if (lilB.max.y < bigB.min.y) {</pre>
176
                       off.y = lilB.max.y - bigB.min.y;
177
                   }
178
179
                   if (lilB.min.z > bigB.max.z) {
180
                       off.z = lilB.min.z - bigB.max.z;
                      else if (lilB.max.z < bigB.min.z) {</pre>
181
182
                       off.z = lilB.max.z - bigB.min.z;
183
184
                   return( off );
185
186
              }
187
               return( Vector3.zero );
188
189
          }
190
191
192
```

```
193
     194
195
         // This function will iteratively climb up the transform.parent tree
            until it either finds a parent with a tag != "Untagged" or no parent
196
         public static GameObject FindTaggedParent(GameObject go) {
197
198
            // If this gameObject has a tag
            if (go.tag != "Untagged") {
199
                // then return this gameObject
200
201
                return(go);
202
            // If there is no parent of this Transform
203
204
            if (go.transform.parent == null) {
205
                // We've reached the end of the line with no interesting tag
                // So return null
206
207
                return( null );
208
            // Otherwise, recursively climb up the tree
209
210
            return( FindTaggedParent( go.transform.parent.gameObject ) );
211
         // This version of the function handles things if a Transform is passed in
212
213
         public static GameObject FindTaggedParent(Transform t) {
214
             return( FindTaggedParent( t.gameObject ) );
215
216
217
218
219
     220
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
     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) {
            Vector3 res = (1-u)*vFrom + u*vTo;
242
            return( res );
243
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;
            return( res );
248
249
250
         // The same function for float
         static public float Lerp (float vFrom, float vTo, float u) {
251
252
            float res = (1-u)*vFrom + u*vTo;
253
             return( res );
254
         }
255
256
```

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

```
385
      386
      [System.Serializable]
387
      public class EasingCachedCurve {
388
                                    curves =
          public List<string>
                                                 new List<string>();
389
          public List<float>
                                    mods =
                                                   new List<float>();
390
391
392
      public class Easing {
                                               ",Linear|";
",In|";
",Out|";
          static public string Linear =
393
394
          static public string In =
395
          static public string Out =
          static public string InOut =
static public string Sin =
                                               ", InOut|";
396
                                                  ,Sin|";
397
                                              ",SinIn|";
          static public string SinIn =
398
                                              ",SinOut|";
399
          static public string SinOut =
400
401
          static public Dictionary<string,EasingCachedCurve> cache;
402
          // This is a cache for the information contained in the complex strings
               that can be passed into the Ease function. The parsing of these
403
               strings is most of the effort of the Ease function, so each time one
404
          //
405
               is parsed, the result is stored in the cache to be recalled much
406
          //
               faster than a parse would take.
          // Need to be careful of memory leaks, which could be a problem if several
407
               million unique easing parameters are called
408
          //
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;
              foreach ( string curve in curveParams ) {
415
                  // Check to see if this curve is already cached
416
417
                  if (!cache.ContainsKey(curve)) {
                      // If not, parse and cache it
418
419
                      EaseParse(curve);
420
421
                  // Call the cached curve
                  u2 = EaseP( u2, cache[curve] );
422
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;
                  // Split each curve on | to find curve and mod
433
                  string[] curveA = curve.Split('|');
434
                  ecc.curves.Add(curveA[0]);
435
                  if (curveA.Length == 1 || curveA[1] == "") {
436
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 {
                          ecc.mods.Add( float.NaN );
443
444
445
                  }
446
447
              cache.Add(curveIn, ecc);
          }
448
```

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