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 );
        }
    }
    // This is the private static field that camBounds uses
    static private Bounds _camBounds;

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

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

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

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

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



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

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

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

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

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


    //============================ Trace & Logging Functions ============================

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


    //============================ Math Functions ============================

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

    static public string AddCommasToNumber(float f, int places=2) {
        int n = Mathf.RoundToInt(f);
        f -= n;
        f = RoundToPlaces(f,places);
        string str = AddCommasToNumber( n );
        str += "."+(f*Mathf.Pow(10,places));
        return( str );
    }
    static public string AddCommasToNumber(int n) {
        int rem;
        int div;
        string res = "";
        string rems;
        while (n>0) {
            rem = n % 1000;
            div = n / 1000;
            rems = rem.ToString();

            while (div>0 && rems.Length<3) {
                rems = "0"+rems;
            }
            // NOTE: It is somewhat faster to use a StringBuilder or a List<String> which is then concatenated using String.Join().
            if (res == "") {
                res = rems;
            }  else {
                res = rems + "," + res.ToString();
            }
            n = div;
        }
        if (res == "") res = "0";
        return( res );
    }
}
```

```
//============================ Easing Classes ============================
[System.Serializable]
public class EasingCachedCurve {
    public List<string>        curves =    new List<string>();
    public List<float>        mods =         new List<float>();
}

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

    static public Dictionary<string,EasingCachedCurve> cache;
    // This is a cache for the information contained in the complex strings
    //   that can be passed into the Ease function. The parsing of these
    //   strings is most of the effort of the Ease function, so each time one
    //   is parsed, the result is stored in the cache to be recalled much
    //   faster than a parse would take.
    // Need to be careful of memory leaks, which could be a problem if several
    //   million unique easing parameters are called
    static public float Ease( float u, params string[] curveParams ) {
        // Set up the cache for curves
        if (cache == null) {
            cache = new Dictionary<string, EasingCachedCurve>();
        }
        float u2 = u;
        foreach ( string curve in curveParams ) {
            // Check to see if this curve is already cached
            if (!cache.ContainsKey(curve)) {
                // If not, parse and cache it
                EaseParse(curve);
            }
            // Call the cached curve
            u2 = EaseP( u2, cache[curve] );
        }
        return( u2 );
    }

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

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

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

    static private float EaseP( float u, string curve, float mod ) {
        float u2 = u;

        switch (curve) {
        case "In":
            if (float.IsNaN(mod)) mod = 2;
            u2 = Mathf.Pow(u, mod);
            break;

        case "Out":
            if (float.IsNaN(mod)) mod = 2;
            u2 = 1 - Mathf.Pow( 1-u, mod );
            break;

        case "InOut":
            if (float.IsNaN(mod)) mod = 2;
            if ( u <= 0.5f ) {
                u2 = 0.5f * Mathf.Pow( u*2, mod );
            }  else {
                u2 = 0.5f + 0.5f * (  1 - Mathf.Pow( 1-(2*(u-0.5f)), mod )  );
            }
            break;

        case "Sin":
            if (float.IsNaN(mod)) mod = 0.15f;
            u2 = u + mod * Mathf.Sin( 2*Mathf.PI*u );
            break;

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

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

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

        return( u2 );
    }

}
```