Letter.cs

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

public class Letter : MonoBehaviour {

    private char            _c;    // The char shown on this Letter
    public TextMesh         tMesh; // The TextMesh shows the char
    public Renderer         tRend; // The Renderer of 3D Text. This will
    //  determine whether the char is visible
    public bool             big = false; // Big letters are a little different
    // Linear interpolation fields
    public List<Vector3>    pts = null;
    public float            timeDuration = 0.5f;
    public float            timeStart = -1;
    public string           easingCuve = Easing.InOut; // Easing from Utils.cs

    void Awake() {
        tMesh = GetComponentInChildren<TextMesh>();
        tRend = tMesh.GetComponent<Renderer>();
        visible = false;
    }

    // Used to get or set _c and the letter shown by 3D Text
    public char            c {
        get {
            return( _c );
        }
        set {
            _c = value;
            tMesh.text = _c.ToString();
        }
    }

    // Gets or sets _c as a string
    public string str {
        get {
            return( _c.ToString() );
        }
        set {
            c = value[0];
        }
    }

    // Enables or disables the renderer for 3D Text, which causes the char to be
    //  visible or invisible respectively.
    public bool visible {
        get {
            return( tRend.enabled );
        }
        set {
            tRend.enabled = value;
        }
    }

    // Gets or sets the color of the rounded rectangle
    public Color color {
        get {
            return(GetComponent<Renderer>().material.color);
        }
        set {
            GetComponent<Renderer>().material.color = value;
        }
    }

    // Now sets-up a Bezier curve to move to the new position
    public Vector3 pos {
        set {
            // Find a midpoint that is a random distance from the actual
            //  midpoint between the current position and the value passed in
            Vector3 mid = (transform.position + value)/2f;
            // The random distance will be within 1/4 of the magnitude of the
            //  line from the actual midpoint
            float mag = (transform.position - value).magnitude;
            mid += Random.insideUnitSphere * mag*0.25f;
            // Create a List<Vector3> of Bezier points
            pts = new List<Vector3>() { transform.position, mid, value };
            // If timeStart is at the default -1, then set it
            if (timeStart == -1 ) timeStart = Time.time;
        }
    }

    // Moves immediately to the new position
    public Vector3 position {
        set {
            transform.position = value;
        }
    }

    // Interpolation code
    void Update() {
        if (timeStart == -1) return;

        // Standard linear interpolation code
        float u = (Time.time-timeStart)/timeDuration;
        u = Mathf.Clamp01(u);
        float u1 = Easing.Ease(u,easingCuve);
        Vector3 v = Utils.Bezier(u1, pts);
        transform.position = v;

        // If the interpolation is done, set timeStart back to -1
        if (u == 1) timeStart = -1;
    }
}
```

Prototools/Utils.cs

```csharp
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) ) );
    }


    // The same two functions for Quaternion
    static public Quaternion Bezier( float u, List<Quaternion> 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<Quaternion> 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<Quaternion> vListL = vList.GetRange(0, vList.Count-1);
        // The result is the Slerp of these two shorter Lists
        // It's possible that Quaternion.Slerp may clamp u to [0..1] :(
        Quaternion res = Quaternion.Slerp( Bezier(u, vListL), Bezier(u, vListR), u );
        return( res );
    }

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

    }
```

WordGame.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;   // We'll be using List<> & Dictionary<>
using System.Linq;                  // We'll be using LINQ

public enum GameMode {
    preGame,    // Before the game starts
    loading,    // The word list is loading and being parsed
    makeLevel,  // The individual WordLevel is being created
    levelPrep,  // The level visuals are Instantiated
    inLevel     // The level is in progress
}

public class WordGame : MonoBehaviour {
    static public WordGame   S; // Singleton

    public GameObject        prefabLetter;
    public bool              showAllWyrds = true;
    public Rect              wordArea = new Rect(-24,19,48,28);
    public float             letterSize = 1.5f;
    public float             bigLetterSize = 4f;
    public Color             bigColorDim = new Color(0.8f, 0.8f, 0.8f);
    public Color             bigColorSelected = Color.white;
    public Vector3           bigLetterCenter = new Vector3(0, -16, 0);
    public List<float>       scoreFontSizes = new List<float> { 24, 36, 36, 1 };
    public Vector3           scoreMidPoint = new Vector3(1,1,0);
    public float             scoreComboDelay = 0.5f;
    public Color[]           wyrdPalette;

    public bool ________________;

    public GameMode          mode = GameMode.preGame;
    public WordLevel         currLevel;
    public List<Wyrd>        wyrds;
    public List<Letter>      bigLetters;
    public List<Letter>      bigLettersActive;
    public string            testWord;
    private string           upperCase = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    void Awake() {
        S = this; // Assign the singleton
    }

    void Start () {
        mode = GameMode.loading;
        // Tells WordList.S to start parsing all the words
        WordList.S.Init();
    }

    // Called by the SendMessage() command from WordList
    public void WordListParseComplete() {
        mode = GameMode.makeLevel;
        // Make a level and assign it to currLevel, the current WordLevel
        currLevel = MakeWordLevel();
    }

    // With the default value of -1, this method will generate a level from
    //  a random word.
    public WordLevel MakeWordLevel(int levelNum = -1) {
        WordLevel level = new WordLevel();
        if (levelNum == -1) {
            // Pick a random level
            level.longWordIndex = Random.Range(0,WordList.S.longWordCount);
        }  else {
            // This can be added later
        }
        level.levelNum = levelNum;
        level.word = WordList.S.GetLongWord(level.longWordIndex);
        level.charDict = WordLevel.MakeCharDict(level.word);

        // Call a coroutine to check all the words in the WordList and see
        // whether each word can be spelled by the chars in level.charDict
        StartCoroutine( FindSubWordsCoroutine(level) );

        // This returns the level before the couroutine finishes, so
        //  SubWordSearchComplete() is called when the coroutine is done
        return( level );
    }

    // A coroutine that finds words that can be spelled in this level
    public IEnumerator FindSubWordsCoroutine(WordLevel level) {
        level.subWords = new List<string>();
        string str;

        List<string> words = WordList.S.GetWords();
        // ^ This is very fast because List<string> is passed by reference

        // Iterate through all the words in the WordList
        for (int i=0; i<WordList.S.wordCount; i++) {
            str = words[i];
            // Check whether each one can be spelled using level.charDict
            if (WordLevel.CheckWordInLevel(str, level)) {
                level.subWords.Add(str);
            }
            // Yield if we've parsed a lot of words this frame
            if (i%WordList.S.numToParseBeforeYield == 0) {
                // yield until the next frame
                yield return null;
            }
        }

        // List<string>.Sort() sorts alphabetically by default
        level.subWords.Sort ();
        // Now sort by length to have words grouped by number of letters
        level.subWords = SortWordsByLength(level.subWords).ToList();

        // The coroutine is complete, so call SubWordSearchComplete()
        SubWordSearchComplete();
    }

    public static IEnumerable<string> SortWordsByLength(IEnumerable<string> e)
    {
        // Use LINQ to sort the array received and return a copy
        // The LINQ syntax is different from regular C# and is beyond
        //  the scope of this book
        var sorted = from s in e
            orderby s.Length ascending
                select s;
        return sorted;
    }

    public void SubWordSearchComplete() {
        mode = GameMode.levelPrep;
        Layout();
    }
    void Layout() {
        // Place the letters for each subword of currLevel on screen
        wyrds = new List<Wyrd>();

        // Declare a lot of variables that will be used in this method
        GameObject go;
        Letter lett;
        string word;
        Vector3 pos;
        float left = 0;
        float columnWidth = 3;
        char c;
        Color col;
        Wyrd wyrd;

        // Determine how many rows of Letters will fit on screen
        int numRows = Mathf.RoundToInt(wordArea.height/letterSize);

        // Make a Wyrd of each level.subWord
        for (int i=0; i<currLevel.subWords.Count; i++) {
            wyrd = new Wyrd();
            word = currLevel.subWords[i];

            // if the word is longer than columnWidth, expand it
            columnWidth = Mathf.Max( columnWidth, word.Length );

            // Instantiate a PrefabLetter for each letter of the word
            for (int j=0; j<word.Length; j++) {
                c = word[j]; // Grab the jth char of the word
                go = Instantiate(prefabLetter) as GameObject;
                lett = go.GetComponent<Letter>();
                lett.c = c; // Set the c of the Letter
                // Position the Letter
                pos = new Vector3(wordArea.x+left+j*letterSize, wordArea.y, 0);
                lett.timeStart = Time.time + i*0.05f;
                // The % here makes multiple columns line up
                pos.y -= (i%numRows)*letterSize;

                // Move the lett immediately to a position above the screen
                lett.position = pos+Vector3.up*(20+i%numRows);
                // Then set the pos for it to interpolate to
                lett.pos = pos;
                // Increment lett.timeStart to move wyrds at different times
                lett.timeStart = Time.time + i*0.05f;

                go.transform.localScale = Vector3.one*letterSize;
                wyrd.Add(lett);
            }

            if (showAllWyrds) wyrd.visible = true; // This line is for testing

            // Color the wyrd based on length
            wyrd.color = wyrdPalette[word.Length-WordList.S.wordLengthMin];

            wyrds.Add(wyrd);

            // If we've gotten to the numRows(th) row, start a new column
            if (i%numRows == numRows-1) {
                left += (columnWidth+0.5f)*letterSize;
            }
        }

       // Place the big letters
        // Initialize the List<>s for big Letters
        bigLetters = new List<Letter>();
        bigLettersActive = new List<Letter>();
        // Create a big Letter for each letter in the target word
        for (int i=0; i<currLevel.word.Length; i++) {
            // This is similar to the process for a normal Letter
            c = currLevel.word[i];
            go = Instantiate(prefabLetter) as GameObject;
            lett = go.GetComponent<Letter>();
            lett.c = c;
            go.transform.localScale = Vector3.one*bigLetterSize;

            // Set the initial position of the big Letters below screen
            pos = new Vector3( 0, -100, 0 );
            lett.position = pos;
            // Increment lett.timeStart to have big Letters come in last
            lett.timeStart = Time.time + currLevel.subWords.Count*0.05f;
            lett.easingCuve = Easing.Sin+"-0.18"; // Bouncy easing

            col = bigColorDim;
            lett.color = col;
            lett.visible = true; // This is always true for big letters
            lett.big = true;
            bigLetters.Add(lett);
        }
        // Shuffle the big letters
        bigLetters = ShuffleLetters(bigLetters);
        // Arrange them on screen
        ArrangeBigLetters();

        // Set the mode to be in-game
        mode = GameMode.inLevel;
    }

    // This shuffles a List<Letter> randomly and returns the result
    List<Letter> ShuffleLetters(List<Letter> letts) {
        List<Letter> newL = new List<Letter>();
        int ndx;
        while(letts.Count > 0) {
            ndx = Random.Range(0,letts.Count);
            newL.Add(letts[ndx]);
            letts.RemoveAt(ndx);
        }
        return(newL);
    }

    // This arranges the big Letters on screen
    void ArrangeBigLetters() {
        // The halfWidth allows the big Letters to be centered
        float halfWidth = ( (float) bigLetters.Count )/2f-0.5f;
        Vector3 pos;
        for (int i=0; i<bigLetters.Count; i++) {
            pos = bigLetterCenter;
            pos.x += (i-halfWidth)*bigLetterSize;
            bigLetters[i].pos = pos;
        }
        // bigLettersActive
        halfWidth = ( (float) bigLettersActive.Count )/2f-0.5f;
        for (int i=0; i<bigLettersActive.Count; i++) {
            pos = bigLetterCenter;
            pos.x += (i-halfWidth)*bigLetterSize;
            pos.y += bigLetterSize*1.25f;
            bigLettersActive[i].pos = pos;
        }
    }

    void Update() {
        // Declare a couple useful local variables
        Letter l;
        char c;

        switch (mode) {
        case GameMode.inLevel:
            // Iterate through each char input by the player this frame
            foreach (char cIt in Input.inputString) {
                // Shift cIt to UPPERCASE
                c = System.Char.ToUpperInvariant(cIt);

                // Check to see if it's an uppercase letter
                if (upperCase.Contains(c)) { // Any uppercase letter
                    // Find an available Letter in bigLetters with this char
                    l = FindNextLetterByChar(c);
                    // If a Letter was returned
                    if (l != null) {
                        // ... then add this char to the testWord and move the
                        //  returned big Letter to bigLettersActive
                        testWord += c.ToString();
                        // Move it from the inactive to the active List<>
                        bigLettersActive.Add(l);
                        bigLetters.Remove(l);
                        l.color = bigColorSelected; // Make it the active color
                        ArrangeBigLetters();        // Rearrange the big Letters
                    }
                }

                if (c == '\b') { // Backspace
                    // Remove the last Letter in bigLettersActive
                    if (bigLettersActive.Count == 0) return;
                    if (testWord.Length > 1) {
                        // Clear the last char of testWord
                        testWord = testWord.Substring(0,testWord.Length-1);
                    }  else {
                        testWord = "";
                    }

                    l = bigLettersActive[bigLettersActive.Count-1];
                    // Move it from the active to the inactive List<>
                    bigLettersActive.Remove(l);
                    bigLetters.Add (l);
                    l.color = bigColorDim;         // Make it the inactive color
                    ArrangeBigLetters();           // Rearrange the big Letters
                }

                if (c == '\n' || c == '\r') { // Return/Enter
                    // Test the testWord against the words in WordLevel
                    StartCoroutine( CheckWord() );
                }

                if (c == ' ') { // Space
                    // Shuffle the bigLetters
                    bigLetters = ShuffleLetters(bigLetters);
                    ArrangeBigLetters();
                }
            }

            break;
        }

    }
    // This finds an available Letter with the char c in bigLetters.
    // If there isn't one available, it returns null.
    Letter FindNextLetterByChar(char c) {
        // Search through each Letter in bigLetters
        foreach (Letter l in bigLetters) {
            // If one has the same char as c
            if (l.c == c) {
                // ...then return it
                return(l);
            }
        }
        // Otherwise, return null
        return( null );
    }

    public IEnumerator CheckWord() {
        // Test testWord against the level.subWords
        string subWord;
        bool foundTestWord = false;

        // Create a List<int> to hold the indices of other subWords that are
        //  contained within testWord
        List<int> containedWords = new List<int>();

        // Iterate through each word in currLevel.subWords
        for (int i=0; i<currLevel.subWords.Count; i++) {

            // If the ith Wyrd on screen has already been found
            if (wyrds[i].found) {
                // ...then continue & skip the rest of this iteration
                continue;
                // This works because the Wyrds on screen and the words in the
                //  subWords List<> are in the same order
            }

            subWord = currLevel.subWords[i];
            // if this subWord is the testWord
            if (string.Equals(testWord, subWord)) {
                // ...then highlight the subWord
                HighlightWyrd(i);
                Score( wyrds[i], 1 ); // Score the testWord
                foundTestWord = true;
            }  else if (testWord.Contains(subWord)) {
                // ^ else if testWord contains this subWord (e.g. SAND contains AND)
                // ...then add it to the list of containedWords
                containedWords.Add(i);
            }
        }

        // If the test word was found in subWords
        if (foundTestWord) {
            // ...then highlight the other words contained in testWord
            int numContained = containedWords.Count;
            int ndx;
            // Highlight the words in reverse order
            for (int i=0; i<containedWords.Count; i++) {

                // yield for a bit before highlighting each word
                yield return( new WaitForSeconds(scoreComboDelay) );

                ndx = numContained-i-1;
                HighlightWyrd( containedWords[ndx] );
                Score( wyrds[ containedWords[ndx] ], i+2 ); // Score additional words
                // The second parameter (i+2) is the number of this word in the combo
            }
        }
        // Clear the active big Letters regardless of whether testWord was valid
        ClearBigLettersActive();

    }

    // Highlight a Wyrd
    void HighlightWyrd(int ndx) {
        // Activate the subWord
        wyrds[ndx].found = true;   // Let it know it's been found
        // Lighten its color
        wyrds[ndx].color = (wyrds[ndx].color+Color.white)/2f;
        wyrds[ndx].visible = true; // Make its 3D Text visible
    }

    // Remove all the Letters from bigLettersActive
    void ClearBigLettersActive() {
        testWord = "";             // Clear the testWord
        foreach (Letter l in bigLettersActive) {
            bigLetters.Add(l);     // Add each Letter to bigLetters
            l.color = bigColorDim; // Set it to the inactive color
        }
        bigLettersActive.Clear();  // Clear the List<>
        ArrangeBigLetters();       // Rearrange the Letters on screen
    }

    // Add to the score for this word
    // int combo is the number of this word in a combo
    void Score(Wyrd wyrd, int combo) {
        // Create a List<> of Bezier points for the FloatingScore
        List<Vector3> pts = new List<Vector3>();

        // Get the position of the first Letter in the wyrd
        Vector3 pt = wyrd.letters[wyrd.letters.Count-1].transform.position;
        // Convert the pt to a ViewportPoint. VPs range from 0 to 1 across the screen and
          ➥are used for GUIText coordinates
        pt = Camera.main.WorldToViewportPoint(pt);
        pt.z = 0;

        // Make pt the first Bezier point
        pts.Add(pt);
        // Add a second Bezier point
        pts.Add( scoreMidPoint );
        // Make the Scoreboard the last Bezier point
        pts.Add(Scoreboard.S.transform.position);

        // Set the value of the Floating Score
        int value = wyrd.letters.Count * combo;
        FloatingScore fs = Scoreboard.S.CreateFloatingScore(value, pts);

        fs.timeDuration = 2f;
        fs.fontSizes = scoreFontSizes;

        // Double the InOut Easing effect
        fs.easingCurve = Easing.InOut+Easing.InOut;

        // Make the text of the FloatingScore something like "3 x 2"
        string txt = wyrd.letters.Count.ToString();
        if (combo > 1) {
            txt += " x "+combo;
        }
        fs.GetComponent<GUIText>().text = txt;
    }

}
```