Enemy_4.cs

```
using UnityEngine;
using System.Collections;

// Part is another serializable data storage class just like WeaponDefinition
[System.Serializable]
public class Part {
    // These three fields need to be defined in the Inspector pane
    public string           name;          // The name of this part
    public float            health;          // The amount of health this part has
    public string[]         protectedBy;  // The other parts that protect this

    // These two fields are set automatically in Start().
    // Caching like this makes it faster and easier to find these later
    public GameObject       go;              // The GameObject of this part
    public Material         mat;          // The Material to show damage
}

public class Enemy_4 : Enemy {
    // Enemy_4 will start offscreen and then pick a random point on screen to
    //   move to. Once it has arrived, it will pick another random point and
    //   continue until the player has shot it down.

    public Vector3[]        points;  // Stores the p0 & p1 for interpolation
    public float            timeStart;  // Birth time for this Enemy_4
    public float            duration = 4;  // Duration of movement

    public Part[]           parts;   // The array of ship Parts

    void Start () {
        points = new Vector3[2];
        // There is already an initial position chosen by Main.SpawnEnemy()
        //   so add it to points as the initial p0 & p1
        points[0] = pos;
        points[1] = pos;

        InitMovement();

        // Cache GameObject & Material of each Part in parts
        Transform t;
        foreach(Part prt in parts) {
            t = transform.Find(prt.name);
            if (t != null) {
                prt.go = t.gameObject;
                prt.mat = prt.go.GetComponent<Renderer>().material;
            }
        }
    }

    void InitMovement() {
        // Pick a new point to move to that is on screen
        Vector3 p1 = Vector3.zero;
        float esp = Main.S.enemySpawnPadding;
        Bounds cBounds = Utils.camBounds;
        p1.x = Random.Range(cBounds.min.x + esp, cBounds.max.x - esp);
        p1.y = Random.Range(cBounds.min.y + esp, cBounds.max.y - esp);

        points[0] = points[1];  // Shift points[1] to points[0]
        points[1] = p1;         // Add p1 as points[1]

        // Reset the time
        timeStart = Time.time;
    }


```

```
    public override void Move () {
        // This completely overrides Enemy.Move() with a linear interpolation
        float u = (Time.time-timeStart)/duration;
        if (u>=1) {  // if u >=1...
            InitMovement();  // ...then initialize movement to a new point
            u=0;
        }

        u = 1 - Mathf.Pow( 1-u, 2 );  // Apply Ease Out easing to u

        // This line is the same as: pos = (1-u)*points[0] + u*points[1];
        pos = (1-u)*points[0] + u*points[1];
    }

    // This will override the OnCollisionEnter that is part of Enemy.cs
    // Because of the way that MonoBehaviour declares common Unity functions
    //   like OnCollisionEnter(), the override keyword is not necessary.
    void OnCollisionEnter( Collision coll ) {
        GameObject other = coll.gameObject;
        switch (other.tag) {
        case "ProjectileHero":
            Projectile p = other.GetComponent<Projectile>();
            // Enemies don't take damage unless they're on screen
            // This stops the player from shooting them before they are visible
            bounds.center = transform.position + boundsCenterOffset;
            if (bounds.extents == Vector3.zero || Utils.ScreenBoundsCheck(bounds,
              ➥BoundsTest.offScreen) != Vector3.zero) {
                Destroy(other);
                break;
            }

            // Hurt this Enemy
            // Find the GameObject that was hit
            // The Collision coll has contacts[], an array of ContactPoints
            // Because there was a collision, we're guaranteed that there is at
            //   least a contacts[0], and ContactPoints have a reference to
            //   thisCollider, which will be the collider for the part of the
            //   Enemy_5 that was hit.
            GameObject goHit = coll.contacts[0].thisCollider.gameObject;
            Part prtHit = FindPart(goHit);
            if (prtHit == null) { // If prtHit wasn't found
                // ...then it's usually because, very rarely, thisCollider on
                //   contacts[0] will be the ProjectileHero instead of the ship
                //   part. If so, just look for otherCollider instead
                goHit = coll.contacts[0].otherCollider.gameObject;
                prtHit = FindPart(goHit);
            }
            // Check whether this part is still protected
            if (prtHit.protectedBy != null) {
                foreach( string s in prtHit.protectedBy ) {
                    // If one of the protecting parts hasn't been destroyed...
                    if (!Destroyed(s)) {
                        // ...then don't damage this part yet
                        Destroy(other);  // Destroy the ProjectileHero
                        return; // return before causing damage
                    }
                }
            }
            // It's not protected, so make it take damage
            // Get the damage amount from the Projectile.type & Main.W_DEFS
            prtHit.health -= Main.W_DEFS[p.type].damageOnHit;
            // Show damage on the part
            ShowLocalizedDamage(prtHit.mat);
            if (prtHit.health <= 0) {
                // Instead of Destroying this enemy, disable the damaged part
```

```
                prtHit.go.SetActive(false);
            }
            // Check to see if the whole ship is destroyed
            bool allDestroyed = true; // Assume it is destroyed
            foreach( Part prt in parts ) {
                if (!Destroyed(prt)) { // If a part still exists
                    allDestroyed = false; // ...change allDestroyed to false
                    break;                // and break out of the foreach loop
                }
            }
            if (allDestroyed) { // If it IS completely destroyed
                // Tell the Main singleton that this ship has been destroyed
                Main.S.ShipDestroyed( this );
                // Destroy this Enemy
                Destroy(this.gameObject);
            }
            Destroy(other);  // Destroy the ProjectileHero
            break;
        }
    }

    // These two functions find a Part in parts based on name or GameObject
    Part FindPart(string n) {
        foreach( Part prt in parts ) {
            if (prt.name == n) {
                return( prt );
            }
        }
        return( null );
    }
    Part FindPart(GameObject go) {
        foreach( Part prt in parts ) {
            if (prt.go == go) {
                return( prt );
            }
        }
        return( null );
    }

    // These functions return true if the Part has been destroyed
    bool Destroyed(GameObject go) {
        return( Destroyed( FindPart(go) ) );
    }
    bool Destroyed(string n) {
        return( Destroyed( FindPart(n) ) );
    }
    bool Destroyed(Part prt) {
        if (prt == null) { // If no real ph was passed in
            return(true); // Return true (meaning yes, it was destroyed)
        }
        // Returns the result of the comparison: prt.health <= 0
        // If prt.health is 0 or less, returns true (yes, it was destroyed)
        return (prt.health <= 0);
    }

    // This changes the color of just one Part to red instead of the whole ship
    void ShowLocalizedDamage(Material m) {
        m.color = Color.red;
        remainingDamageFrames = showDamageForFrames;
    }
}
```

Main.cs

```
using UnityEngine;                 // Required for Unity
using System.Collections;          // Required for Arrays & other Collections
using System.Collections.Generic;  // Required to use Lists or Dictionaries

public class Main : MonoBehaviour {
    static public Main S;
    static public Dictionary<WeaponType, WeaponDefinition> W_DEFS;

    public GameObject[]            prefabEnemies;
    public float                   enemySpawnPerSecond = 0.5f; // # Enemies/second
    public float                   enemySpawnPadding = 1.5f; // Padding for position
    public WeaponDefinition[]      weaponDefinitions;

    public GameObject              prefabPowerUp;

    public WeaponType[]            powerUpFrequency = new WeaponType[] {
                                     WeaponType.blaster, WeaponType.blaster,
                                     WeaponType.spread,
                                     WeaponType.shield                    } ;

    public bool ________________;

    public WeaponType[]            activeWeaponTypes;
    private float                  enemySpawnRate; // Delay between Enemies


    void Awake() {
        S = this;
        // Set Utils.camBounds
        Utils.SetCameraBounds(this.GetComponent<Camera>());
        // 0.5 enemies/second = enemySpawnRate of 2
        enemySpawnRate = 1f/enemySpawnPerSecond;
        // Invoke 1 call to SpawnEnemy() in 2 seconds
        Invoke( "SpawnEnemy", enemySpawnRate );

        // A generic Dictionary with WeaponType as the key
        W_DEFS = new Dictionary<WeaponType, WeaponDefinition>();
        foreach( WeaponDefinition def in weaponDefinitions ) {
            W_DEFS[def.type] = def;
        }
    }

    static public WeaponDefinition GetWeaponDefinition( WeaponType wt ) {
        // Check to make sure that the key exists in the Dictionary
        if (W_DEFS.ContainsKey(wt)) {
            // Attempting to retrieve a key that doesn't exist, throws an error
            return( W_DEFS[wt]);
        }
        // This will return a definition for WeaponType.none,
        //   which means it has failed to find the WeaponDefinition
        return( new WeaponDefinition() );
    }

    void Start() {
        activeWeaponTypes = new WeaponType[weaponDefinitions.Length];
        for ( int i=0; i<weaponDefinitions.Length; i++ ) {
            activeWeaponTypes[i] = weaponDefinitions[i].type;
        }
    }

    public void SpawnEnemy() {
        // Pick a random Enemy prefab to instantiate
        int ndx = Random.Range(0, prefabEnemies.Length);
        GameObject go = Instantiate( prefabEnemies[ ndx ] ) as GameObject;
```

```
65          // Position the Enemy above the screen with a random x position
66          Vector3 pos = Vector3.zero;
67          float xMin = Utils.camBounds.min.x+enemySpawnPadding;
68          float xMax = Utils.camBounds.max.x-enemySpawnPadding;
69          pos.x = Random.Range( xMin, xMax );
70          pos.y = Utils.camBounds.max.y + enemySpawnPadding;
71          go.transform.position = pos;
72          // Call SpawnEnemy() again in a couple seconds
73          Invoke( "SpawnEnemy", enemySpawnRate );
74      }
75
76      public void DelayedRestart( float delay ) {
77          // Invoke the Restart() method in delay seconds
78          Invoke("Restart", delay);
79      }
80
81      public void Restart() {
82          // Reload _Scene_0 to restart the game
83          Application.LoadLevel("_Scene_0");
84      }
85
86      public void ShipDestroyed( Enemy e ) {
87          // Potentially generate a PowerUp
88          if (Random.value <= e.powerUpDropChance) {
89              // Random.value generates a value between 0 & 1 (though never == 1)
90              // If the e.powerUpDropChance is 0.50f, a PowerUp will be generated
91              //   50% of the time. For testing, it's now set to 1f.
92
93              // Choose which PowerUp to pick
94              // Pick one from the possibilities in powerUpFrequency
95              int ndx = Random.Range(0,powerUpFrequency.Length);
96              WeaponType puType = powerUpFrequency[ndx];
97
98              // Spawn a PowerUp
99              GameObject go = Instantiate( prefabPowerUp ) as GameObject;
100             PowerUp pu = go.GetComponent<PowerUp>();
101             // Set it to the proper WeaponType
102             pu.SetType( puType );
103
104             // Set it to the position of the destroyed ship
105             pu.transform.position = e.transform.position;
106         }
107     }
108
109 }
```

PowerUp.cs

```
using UnityEngine;
using System.Collections;

public class PowerUp : MonoBehaviour {
    // An unusual but useful use of Vector2s, x is a min value
    //   and y is a max for a Random.Range() to be called later
    public Vector2            rotMinMax = new Vector2(15,90);
    public Vector2            driftMinMax = new Vector2(.25f,2);
    public float              lifeTime = 6f; // Seconds the PowerUp exist
    public float              fadeTime = 4f; // Seconds it will then fade
    public bool _______________;
    public WeaponType         type; // The type of the PowerUp
    public GameObject         cube; // Reference to the Cube child
    public TextMesh           letter; // Reference to the TextMesh
    public Vector3            rotPerSecond; // Euler rotation speed
    public float              birthTime;

    void Awake() {
        // Find the Cube reference
        cube = transform.Find("Cube").gameObject;
        // Find the TextMesh
        letter = GetComponent<TextMesh>();

        // Set a random velocity
        Vector3 vel = Random.onUnitSphere; // Get Random XYZ velocity
        // Random.onUnitSphere gives you a vector point that is somewhere on
        //   the surface of the sphere with a radius of 1m around the origin
        vel.z = 0; // Flatten the vel to the XY plane
        vel.Normalize(); // Make the length of the vel 1
        // Normalizing a Vector3 makes it length 1m
        vel *= Random.Range(driftMinMax.x, driftMinMax.y);
        // Above sets the velocity length to something between the x and y
        //   values of driftMinMax
        GetComponent<Rigidbody>().velocity = vel;

        // Set the rotation of this GameObject to R:[0,0,0]
        transform.rotation = Quaternion.identity;
        // Quaternion.identity is equal to no rotation.

        // Set up the rotPerSecond for the Cube child using rotMinMax x & y
        rotPerSecond = new Vector3( Random.Range(rotMinMax.x,rotMinMax.y),
                                    Random.Range(rotMinMax.x,rotMinMax.y),
                                    Random.Range(rotMinMax.x,rotMinMax.y) );

        // CheckOffscreen() every 2 seconds
        InvokeRepeating( "CheckOffscreen", 2f, 2f );

        birthTime = Time.time;
    }

    void Update () {
        // Manually rotate the Cube child every Update()
        // Multiplying it by Time.time causes the rotation to be time-based
        cube.transform.rotation = Quaternion.Euler( rotPerSecond*Time.time );

        // Fade out the PowerUp over time
        // Given the default values, a PowerUp will exist for 10 seconds
        //   and then fade out over 4 seconds.
        float u = (Time.time - (birthTime+lifeTime)) / fadeTime;
        // For lifeTime seconds, u will be <= 0. Then it will transition to 1
        //   over fadeTime seconds.
        // If u >= 1, destroy this PowerUp
        if (u >= 1) {
            Destroy( this.gameObject );
```

```
            return;
        }
        // Use u to determine the alpha value of the Cube & Letter
        if (u>0) {
            Color c = cube.GetComponent<Renderer>().material.color;
            c.a = 1f-u;
            cube.GetComponent<Renderer>().material.color = c;
            // Fade the Letter too, just not as much
            c = letter.color;
            c.a = 1f - (u*0.5f);
            letter.color = c;
        }
    }

    public void SetType( WeaponType wt ) {
        // Grab the WeaponDefinition from Main
        WeaponDefinition def = Main.GetWeaponDefinition( wt );
        // Set the color of the Cube child
        cube.GetComponent<Renderer>().material.color = def.color;
        //letter.color = def.color; // We could colorize the letter too
        letter.text = def.letter; // Set the letter that is shown
        type = wt; // Finally actually set the type
    }

    public void AbsorbedBy( GameObject target ) {
        // This function is called by the Hero class when a PowerUp is collected
        // We could tween into the target and shrink in size,
        //   but for now, just destroy this.gameObject
        Destroy( this.gameObject );
    }

    void CheckOffscreen() {
        // If the PowerUp has drifted entirely off screen
        if ( Utils.ScreenBoundsCheck( cube.GetComponent<Collider>().bounds,
          ➥BoundsTest.offScreen ) != Vector3.zero ) {
            // ...then destroy this GameObject
            Destroy( this.gameObject );
        }
    }
}
```

Projectile.cs

```
using UnityEngine;
using System.Collections;

public class Projectile : MonoBehaviour {
    [SerializeField]
    private WeaponType    _type;

    // This public property masks the field _type & takes action when it is set
    public WeaponType     type {
        get {
            return( _type );
        }
        set {
            SetType( value );
        }
    }

    void Awake() {
        // Test to see whether this has passed off screen every 2 seconds
        InvokeRepeating( "CheckOffscreen", 2f, 2f );
    }

    public void SetType( WeaponType eType ) {
        // Set the _type
        _type = eType;
        WeaponDefinition def = Main.GetWeaponDefinition( _type );
        GetComponent<Renderer>().material.color = def.projectileColor;
    }

    void CheckOffscreen() {
        if ( Utils.ScreenBoundsCheck( GetComponent<Collider>().bounds,
          ➥BoundsTest.offScreen ) != Vector3.zero ) {
            Destroy( this.gameObject );
        }
    }

}
```

Shield.cs

```
using UnityEngine;
using System.Collections;

public class Shield : MonoBehaviour {
    public float      rotationsPerSecond = 0.1f;
    public bool _________________;
    public int        levelShown = 0;

    void Update () {
        // Read the current shield level from the Hero Singleton
        int currLevel = Mathf.FloorToInt( Hero.S.shieldLevel );
        // If this is different from levelShown...
        if (levelShown != currLevel) {
            levelShown = currLevel;
            Material mat = this.GetComponent<Renderer>().material;
            // Adjust the texture offset to show different shield level
            mat.mainTextureOffset = new Vector2( 0.2f*levelShown, 0 );
        }
        // Rotate the shield a bit every second
        float rZ = (rotationsPerSecond*Time.time*360) % 360f;
        transform.rotation = Quaternion.Euler( 0, 0, rZ );
    }

}
```

Utils.cs

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

// This is actually OUTSIDE of the Utils Class
public enum BoundsTest {
    center,        // Is the center of the GameObject on screen
    onScreen,    // Are the bounds entirely on screen
    offScreen    // Are the bounds entirely off screen
}

public class Utils : MonoBehaviour {


//============================ Bounds Functions ============================\\

    // Creates bounds that encapsulate of the two Bounds passed in.
    public static Bounds BoundsUnion( Bounds b0, Bounds b1 ) {
        // If the size of one of the bounds is Vector3.zero, ignore that one
        if ( b0.size==Vector3.zero && b1.size!=Vector3.zero ) {
            return( b1 );
        }  else if ( b0.size!=Vector3.zero && b1.size==Vector3.zero ) {
            return( b0 );
        }  else if ( b0.size==Vector3.zero && b1.size==Vector3.zero ) {
            return( b0 );
        }
        // Stretch b0 to include the b1.min and b1.max
        b0.Encapsulate(b1.min);
        b0.Encapsulate(b1.max);
        return( b0 );
    }

    public static Bounds CombineBoundsOfChildren(GameObject go) {
        // Create an empty Bounds b
        Bounds b = new Bounds(Vector3.zero, Vector3.zero);
        // If this GameObject has a Renderer Component...
        if (go.GetComponent<Renderer>() != null) {
            // Expand b to contain the Renderer's Bounds
            b = BoundsUnion(b, go.GetComponent<Renderer>().bounds);
        }
        // If this GameObject has a Collider Component...
        if (go.GetComponent<Collider>() != null) {
            // Expand b to contain the Collider's Bounds
            b = BoundsUnion(b, go.GetComponent<Collider>().bounds);
        }
        // Iterate through each child of this gameObject.transform
        foreach( Transform t in go.transform ) {
            // Expand b to contain their Bounds as well
            b = BoundsUnion( b, CombineBoundsOfChildren( t.gameObject ) );
        }

        return( b );
    }

    // Make a static read-only public property camBounds
    static public Bounds camBounds {
        get {
            // if _camBounds hasn't been set yet
            if (_camBounds.size == Vector3.zero) {
                // SetCameraBounds using the default Camera
                SetCameraBounds();
            }
            return( _camBounds );
        }
```

```
65      }
66      // This is the private static field that camBounds uses
67      static private Bounds _camBounds;
68
69      public static void SetCameraBounds(Camera cam=null) {
70          // If no Camera was passed in, use the main Camera
71          if (cam == null) cam = Camera.main;
72          // This makes a couple important assumptions about the camera!:
73          //   1. The camera is Orthographic
74          //   2. The camera is at a rotation of R:[0,0,0]
75
76          // Make Vector3s at the topLeft and bottomRight of the Screen coords
77          Vector3 topLeft = new Vector3( 0, 0, 0 );
78          Vector3 bottomRight = new Vector3( Screen.width, Screen.height, 0 );
79
80          // Convert these to world coordinates
81          Vector3 boundTLN = cam.ScreenToWorldPoint( topLeft );
82          Vector3 boundBRF = cam.ScreenToWorldPoint( bottomRight );
83
84          // Adjust the z to be at the near and far Camera clipping planes
85          boundTLN.z += cam.nearClipPlane;
86          boundBRF.z += cam.farClipPlane;
87
88          // Find the center of the Bounds
89          Vector3 center = (boundTLN + boundBRF)/2f;
90          _camBounds = new Bounds( center, Vector3.zero );
91          // Expand _camBounds to encapsulate the extents.
92          _camBounds.Encapsulate( boundTLN );
93          _camBounds.Encapsulate( boundBRF );
94      }
95
96
97
98      // Test to see whether Bounds are on screen.
99      public static Vector3 ScreenBoundsCheck(Bounds bnd, BoundsTest test =
          ➥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
          ➥= BoundsTest.onScreen ) {
106         // Get the center of lilB
107         Vector3 pos = lilB.center;
108
109         // Initialize the offset at [0,0,0]
110         Vector3 off = Vector3.zero;
111
112         switch (test) {
113 // The center test determines what off (offset) would have to be applied to lilB to move
      ➥its center back inside bigB
114         case BoundsTest.center:
115             // if the center is contained, return Vector3.zero
116             if ( bigB.Contains( pos ) ) {
117                 return( Vector3.zero );
118             }
119            // if not contained, find the offset
120             if (pos.x > bigB.max.x) {
121                 off.x = pos.x - bigB.max.x;
122             }  else  if (pos.x < bigB.min.x) {
123                 off.x = pos.x - bigB.min.x;
124             }
125             if (pos.y > bigB.max.y) {
126                 off.y = pos.y - bigB.max.y;
127             }  else  if (pos.y < bigB.min.y) {
128                 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);
        // Remove the last element of vList, leaving one fewer
        // e.g. if vList = [0,1,2,3,4] then vList = [0,1,2,3]
        vList.RemoveAt(vList.Count-1);
        // The result is the Lerp of these two shorter Lists
        Vector3 res = Lerp( Bezier(u, vList), 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);
        // Remove the last element of vList, leaving one fewer
        // e.g. if vList = [0,1,2,3,4] then vList = [0,1,2,3]
        vList.RemoveAt(vList.Count-1);
        // The result is the Lerp of these two shorter Lists
        Vector2 res = Lerp( Bezier(u, vList), 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) ) );
    }

}




//============================ 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 ) );
    }

```

```
385     static private float EaseP( float u, EasingCachedCurve ec ) {
386         float u2 = u;
387         for (int i=0; i<ec.curves.Count; i++) {
388             u2 = EaseP( u2, ec.curves[i], ec.mods[i] );
389         }
390         return( u2 );
391     }
392 
393     static private float EaseP( float u, string curve, float mod ) {
394         float u2 = u;
395 
396         switch (curve) {
397         case "In":
398             if (float.IsNaN(mod)) mod = 2;
399             u2 = Mathf.Pow(u, mod);
400             break;
401 
402         case "Out":
403             if (float.IsNaN(mod)) mod = 2;
404             u2 = 1 - Mathf.Pow( 1-u, mod );
405             break;
406 
407         case "InOut":
408             if (float.IsNaN(mod)) mod = 2;
409             if ( u <= 0.5f ) {
410                 u2 = 0.5f * Mathf.Pow( u*2, mod );
411             }  else {
412                 u2 = 0.5f + 0.5f * (  1 - Mathf.Pow( 1-(2*(u-0.5f)), mod )  );
413             }
414             break;
415 
416         case "Sin":
417             if (float.IsNaN(mod)) mod = 0.15f;
418             u2 = u + mod * Mathf.Sin( 2*Mathf.PI*u );
419             break;
420 
421         case "SinIn":
422             // mod is ignored for SinIn
423             u2 = 1 - Mathf.Cos( u * Mathf.PI * 0.5f );
424             break;
425 
426         case "SinOut":
427             // mod is ignored for SinOut
428             u2 = Mathf.Sin( u * Mathf.PI * 0.5f );
429             break;
430 
431         case "Linear":
432         default:
433             // u2 already equals u
434             break;
435         }
436 
437         return( u2 );
438     }
439 
440 }
```

Weapon.cs

```
using UnityEngine;
using System.Collections;

// This is an enum of the various possible weapon types
// It also includes a "shield" type to allow a shield power-up
// Items marked [NI] below are Not Implemented in the book
public enum WeaponType {
    none,        // The default / no weapon
    blaster,    // A simple blaster
    spread,        // Two shots simultaneously
    phaser,        // Shots that move in waves [NI]
    missile,    // Homing missiles [NI]
    laser,        // Damage over time [NI]
    shield        // Raise shieldLevel
}

// The WeaponDefinition class allows you to set the properties
//   of a specific weapon in the Inspector. Main has an array
//   of WeaponDefinitions that makes this possible.
[System.Serializable]
// System.Serializable tells Unity to try to view WeaponDefinition
//   in the Inspector pane. It doesn't work for everything, but it
//   will work for simple classes like this!
public class WeaponDefinition {
    public WeaponType   type = WeaponType.none;
    public string       letter;            // The letter to show on the power-up
    public Color        color = Color.white;       // Color of Collar & power-up
    public GameObject   projectilePrefab;          // Prefab for projectiles
    public Color        projectileColor = Color.white;
    public float        damageOnHit = 0;           // Amount of damage caused
    public float        continuousDamage = 0;      // Damage per second (Laser)
    public float        delayBetweenShots = 0;
    public float        velocity = 20;             // Speed of projectiles
}

// Note: Weapon prefabs, colors, etc. are set in the class Main.

public class Weapon : MonoBehaviour {
    static public Transform        PROJECTILE_ANCHOR;

    public bool ____________________;
    [SerializeField]
    private WeaponType            _type = WeaponType.blaster;
    public WeaponDefinition        def;
    public GameObject             collar;
    public float                 lastShot;

    void Awake() {
        collar = transform.Find("Collar").gameObject;
    }

    void Start() {
        // Call SetType() properly for the default _type
        SetType( _type );
        if (PROJECTILE_ANCHOR == null) {
            GameObject go = new GameObject("_Projectile_Anchor");
            PROJECTILE_ANCHOR = go.transform;
        }
        // Find the fireDelegate of the parent
        GameObject parentGO = transform.parent.gameObject;
        if (parentGO.tag == "Hero") {
            Hero.S.fireDelegate += Fire;
       }
    }
```

```
65
66      public WeaponType type {
67          get {    return( _type );    }
68          set {    SetType( value );    }
69      }
70
71      public void SetType( WeaponType wt ) {
72          _type = wt;
73          if (type == WeaponType.none) {
74              this.gameObject.SetActive(false);
75              return;
76          }  else {
77              this.gameObject.SetActive(true);
78          }
79          def = Main.GetWeaponDefinition(_type);
80          collar.GetComponent<Renderer>().material.color = def.color;
81          lastShot = 0; // You can always fire immediately after _type is set.
82      }
83
84      public void Fire() {
85          // If this.gameObject is inactive, return
86          if (!gameObject.activeInHierarchy) return;
87          // If it hasn't been enough time between shots, return
88          if (Time.time - lastShot < def.delayBetweenShots) {
89              return;
90          }
91          Projectile p;
92          switch (type) {
93          case WeaponType.blaster:
94              p = MakeProjectile();
95              p.GetComponent<Rigidbody>().velocity = Vector3.up * def.velocity;
96              break;
97
98          case WeaponType.spread:
99              p = MakeProjectile();
100             p.GetComponent<Rigidbody>().velocity = Vector3.up * def.velocity;
101             p = MakeProjectile();
102             p.GetComponent<Rigidbody>().velocity = new Vector3( -.2f, 0.9f, 0 ) *
                  ➥def.velocity;
103             p = MakeProjectile();
104             p.GetComponent<Rigidbody>().velocity = new Vector3(  .2f, 0.9f, 0 ) *
                  ➥def.velocity;
105             break;
106
107         }
108
109     }
110
111     public Projectile MakeProjectile() {
112             GameObject go = Instantiate( def.projectilePrefab ) as GameObject;
113         if ( transform.parent.gameObject.tag == "Hero" ) {
114             go.tag = "ProjectileHero";
115             go.layer = LayerMask.NameToLayer("ProjectileHero");
116         }  else {
117             go.tag = "ProjectileEnemy";
118             go.layer = LayerMask.NameToLayer("ProjectileEnemy");
119         }
120         go.transform.position = collar.transform.position;
121         go.transform.parent = PROJECTILE_ANCHOR;
122         Projectile p = go.GetComponent<Projectile>();
123         p.type = type;
124         lastShot = Time.time;
125         return( p );
126
127     }
128 }
```