EnemyBug.cs

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


public class EnemyBug : PT_MonoBehaviour, Enemy {
    [SerializeField]
    private float           _touchDamage = 1;
    public float           touchDamage {
        get { return( _touchDamage ); }
        set { _touchDamage = value; }
    }
    // The pos Property is already implemented in PT_MonoBehaviour
    public string           typeString {
        get { return( roomXMLString ); }
        set { roomXMLString = value; }
    }

    public string           roomXMLString;
    public float           speed = 0.5f;
    public float           health = 10;
    public float           damageScale = 0.8f;
    public float           damageScaleDuration = 0.25f;

    public bool _________________;

    private float           damageScaleStartTime;
    private float           _maxHealth;
    public Vector3           walkTarget;
    public bool               walking;
    public Transform        characterTrans;
    // Stores damage for each element each frame
    public Dictionary<ElementType,float>    damageDict;
    // ^ NOTE: Dictionaries do not appear in the Unity Inspector

    void Awake() {
        characterTrans = transform.Find("CharacterTrans");
        _maxHealth = health; // Always starts with max health
        ResetDamageDict();
    }

    // Resets the values for the damageDict
    void ResetDamageDict() {
        if (damageDict == null) {
            damageDict = new Dictionary<ElementType, float>();
        }
        damageDict.Clear();
        damageDict.Add(ElementType.earth, 0);
        damageDict.Add(ElementType.water, 0);
        damageDict.Add(ElementType.air,   0);
        damageDict.Add(ElementType.fire,  0);
        damageDict.Add(ElementType.aether,0);
        damageDict.Add(ElementType.none,  0);
    }

    void Update() {
        WalkTo (Mage.S.pos);
    }

    // All of this walking code is copied directly from Mage


   // ---------------- Walking Code ----------------
    // Walk to a specific position. The position.z is always 0
```

```
65      public void WalkTo(Vector3 xTarget) {
66          walkTarget = xTarget;    // Set the point to walk to
67          walkTarget.z = 0;        // Force z=0
68          walking = true;            // Now the EnemyBug is walking
69          Face(walkTarget);        // Look in the direction of the walkTarget
70      }
71
72      public void Face(Vector3 poi) { // Face towards a point of interest
73          Vector3 delta = poi-pos; // Find vector to the point of interest
74          // Use Atan2 to get the rotation around Z that points the X-axis of
75          //  EnemyBug:CharacterTrans towards poi
76          float rZ = Mathf.Rad2Deg * Mathf.Atan2(delta.y, delta.x);
77          // Set the rotation of characterTrans (doesn't actually rotate _Mage)
78          characterTrans.rotation = Quaternion.Euler(0,0,rZ);
79      }
80
81      public void StopWalking() { // Stops the EnemyBug from walking
82          walking = false;
83          GetComponent<Rigidbody>().velocity = Vector3.zero;
84      }
85
86      void FixedUpdate () { // Happens every physics step (i.e. 60 times/second)
87          if (walking) { // If Mage is walking
88              if ( (walkTarget-pos).magnitude < speed*Time.fixedDeltaTime ) {
89                  // If EnemyBug is very close to walkTarget, just stop there
90                  pos = walkTarget;
91                  StopWalking();
92              }  else {
93                  // Otherwise, move towards walkTarget
94                  GetComponent<Rigidbody>().velocity = (walkTarget-pos).normalized * speed;
95              }
96          }  else {
97              // If not walking, velocity should be zero
98              GetComponent<Rigidbody>().velocity = Vector3.zero;
99          }
100     }
101
102     // Damage this instance. By default, the damage is instant, but it can also
103     //  be treated as damage over time, where the amt value would be the amount
104     //  of damage done every second.
105     // NOTE: This same code can be used to heal the instance
106     public void Damage(float amt, ElementType eT, bool damageOverTime=false) {
107         // If it's DOT, then only damage the fractional amount for this frame
108         if (damageOverTime) {
109             amt *= Time.deltaTime;
110         }
111
112         // Treat different damage types differently (most are default)
113         switch (eT) {
114         case ElementType.fire:
115             // Only the max damage from one fire source affects this instance
116             damageDict[eT] = Mathf.Max ( amt, damageDict[eT] );
117             break;
118
119         case ElementType.air:
120             // air doesn't damage EnemyBugs, so do nothing
121             break;
122
123         default:
124             // By default, damage is added to the other damage by same element
125             damageDict[eT] += amt;
126             break;
127         }
128     }
```

```
129
130     // LateUpdate() is automatically called by Unity every frame. Once all the
131     //  Updates() on all instances have been called, then LateUpdate() is called
132     //  on all instances.
133     void LateUpdate() {
134         // Apply damage from the different element types
135
136         // Iteration through a Dictionary uses a KeyValuePair
137         // entry.Key is the ElementType, while entry.Value is the float
138         float dmg = 0;
139         foreach ( KeyValuePair<ElementType,float> entry in damageDict ) {
140             dmg += entry.Value;
141         }
142
143         if (dmg > 0) { // If this took damage...
144             // and if it is at full scale now (& not already damage scaling)...
145             if (characterTrans.localScale == Vector3.one) {
146                 // start the damage scale animation
147                 damageScaleStartTime = Time.time;
148             }
149         }
150
151         // The damage scale animation
152         float damU = (Time.time - damageScaleStartTime)/damageScaleDuration;
153         damU = Mathf.Min(1, damU); // Limit the max localScale to 1
154         float scl = (1-damU)*damageScale + damU*1;
155         characterTrans.localScale = scl * Vector3.one;
156
157         health -= dmg;
158         health = Mathf.Min(_maxHealth, health); // Limit health if healing
159
160         ResetDamageDict(); // Prepare for next frame
161
162         if (health <= 0) {
163             Die();
164         }
165     }
166
167     // Making Die() a separate function allows us to add things later like
168     //  different death animations, dropping something for the player, etc.
169     public void Die() {
170         Destroy(gameObject);
171     }
172
173 }
```

EnemySpiker.cs

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

public class EnemySpiker : PT_MonoBehaviour, Enemy {
    [SerializeField]
    private float           _touchDamage = 0.5f;
    public float            touchDamage {
        get { return( _touchDamage ); }
        set { _touchDamage = value; }
    }
    // The pos Property is already implemented in PT_MonoBehaviour
    public string            typeString {
        get { return( roomXMLString ); }
        set { roomXMLString = value; }
    }
    public float            speed = 5f;
    public string            roomXMLString = "{";
    public bool _________________;
    public Vector3            moveDir;
    public Transform        characterTrans;
    void Awake() {
        characterTrans = transform.Find("CharacterTrans");
    }
    void Start() {
        // Set the move direction based on the character in Rooms.xml
        switch (roomXMLString) {
        case "^":
            moveDir = Vector3.up;
            break;
        case "v":
            moveDir = Vector3.down;
            break;
        case "{":
            moveDir = Vector3.left;
            break;
        case "}":
            moveDir = Vector3.right;
            break;
        }
    }
    void FixedUpdate () { // Happens every physics step (i.e. 60 times/second)
        GetComponent<Rigidbody>().velocity = moveDir * speed;
    }
    // This has the same structure as the Damage Method in EnemyBug
    public void Damage(float amt, ElementType eT, bool damageOverTime=false) {
        // Nothing damages the EnemySpiker
    }
    void OnTriggerEnter(Collider other) {
        // Check to see if a wall was hit
        GameObject go = Utils.FindTaggedParent(other.gameObject);
        if (go == null) return; // In case nothing is tagged

        if (go.tag == "Ground") {
            // Make sure that the ground tile is in the direction we're moving.
            //  A dot product will help us with this (see the Useful Concepts
            //  Reference).
            float dot = Vector3.Dot(moveDir, go.transform.position - pos);
            if (dot > 0) { // If Spiker is moving towards the block it hit
                moveDir *= -1; // Reverse direction
            }
        }
    }
}
```

Mage.cs

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;    // Enables List<>s
using System.Linq;                    // Enables LINQ queries

// The MPhase enum is used to track the phase of Mouse interaction
public enum MPhase {
    idle,
    down,
    drag
}

// The ElementType enum
public enum ElementType {
    earth,
    water,
    air,
    fire,
    aether,
    none
}


// MouseInfo stores information about the Mouse in each frame of interaction
[System.Serializable]
public class MouseInfo {
    public Vector3          loc;         // 3D loc of the mouse near z=0
    public Vector3          screenLoc;   // Screen position of the mouse
    public Ray              ray;         // Ray from the mouse into 3D space
    public float            time;        // Time this mouseInfo was recorded
    public RaycastHit       hitInfo;     // Info about what was hit by the ray
    public bool             hit;         // Whether the mouse was over any collider

    // These methods see if the mouseRay hits anything
    public RaycastHit Raycast() {
        hit = Physics.Raycast(ray, out hitInfo);
        return(hitInfo);
    }

    public RaycastHit Raycast(int mask) {
        hit = Physics.Raycast(ray, out hitInfo, mask);
        return(hitInfo);
    }
}

// Mage is a subclass of PT_MonoBehaviour
public class Mage : PT_MonoBehaviour {
    static public Mage S;
    static public bool DEBUG = false;

    public float            mTapTime = 0.1f;         // How long is considered a tap
    public float            mDragDist = 5;           // Min dist in pixels to be a drag
    public GameObject        tapIndicatorPrefab;     // Prefab of the tap indicator

    public float            activeScreenWidth = 1;   // % of the screen to use

    public float            speed = 2;               // The speed at which _Mage walks

    public GameObject[]     elementPrefabs;          // The Element_Sphere Prefabs
    public float            elementRotDist = 0.5f;   // Radius of rotation
    public float            elementRotSpeed = 0.5f;  // Period of rotation
    public int              maxNumSelectedElements = 1;
    public Color[]          elementColors;

```

```
65      // These set the min and max distance between two line points
66      public float            lineMinDelta = 0.1f;
67      public float            lineMaxDelta = 0.5f;
68      public float            lineMaxLength = 8f;
69
70      public GameObject       fireGroundSpellPrefab;
71
72      public float            health = 4;             // Total mage health
73      public float            damageTime = -100;
74      // ^ Time that damage occurred. It's set to -100 so that the Mage doesn't
75      //    act damaged immediately when the Scene starts
76      public float            knockbackDist = 1;      // Distance to move backward
77      public float            knockbackDur = 0.5f;    // Seconds to move backward
78      public float            invincibleDur = 0.5f;   // Seconds to be invincible
79      public int              invTimesToBlink = 4;    // # blinks while invincible
80
81      public bool _________________;
82
83      private bool            invincibleBool = false; // Is Mage invincible?
84      private bool            knockbackBool = false;  // Mage being knocked back?
85      private Vector3         knockbackDir;           // Direction of knockback
86      private Transform       viewCharacterTrans;
87
88      protected Transform     spellAnchor;            // The parent transform for spells
89
90      protected float         totalLineLength;
91      public List<Vector3>    linePts;                // Points to be shown in the line
92      protected LineRenderer  liner;                  // Ref to the LineRenderer Component
93      protected float         lineZ = -0.1f;          // Z depth of the line
94      // ^ protected variables are between public and private.
95      //    public variables can be seen by everyone
96      //    private variables can only be seen by this class
97      //    protected variables can be seen by this class or any subclasses
98      //    only public variables appear in the Inspector
99
100     public MPhase           mPhase = MPhase.idle;
101     public List<MouseInfo>  mouseInfos = new List<MouseInfo>();
102     public string           actionStartTag; // ["Mage", "Ground", "Enemy"]
103
104     public bool             walking = false;
105     public Vector3          walkTarget;
106     public Transform        characterTrans;
107
108     public List<Element>    selectedElements = new List<Element>();
109
110
111     void Awake() {
112         S = this; // Set the Mage Singleton
113         mPhase = MPhase.idle;
114
115         // Find the characterTrans to rotate with Face()
116         characterTrans = transform.Find("CharacterTrans");
117         viewCharacterTrans = characterTrans.Find("View_Character");
118
119         // Get the LineRenderer component and disable it
120         liner = GetComponent<LineRenderer>();
121         liner.enabled = false;
122
123         GameObject saGO = new GameObject("Spell Anchor");
124         // ^ Create an empty GameObject named "Spell Anchor". When you create a
125         //    new GameObject this way, it's at P:[0,0,0] R:[0,0,0] S:[1,1,1]
126         spellAnchor = saGO.transform; // Get it's transform
127     }
128
```

```
129
130     void Update() {
131
132         // Find whether the mouse button 0 was pressed or released this frame
133         bool b0Down = Input.GetMouseButtonDown(0);
134         bool b0Up = Input.GetMouseButtonUp(0);
135
136         // Handle all input here (except for Inventory buttons)
137         /*
138         There are only a few possible actions:
139         1. Tap on the ground to move to that point
140         2. Drag on the ground with no spell selected to move to the
141             continuously-updating point
142         3. Drag on the ground with spell to cast along the ground
143         4. Tap on an enemy to attack (with or without a spell; no spell is a
144             force push)
145         */
146
147         // An example of using < to return a bool value
148         bool inActiveArea = (float) Input.mousePosition.x / Screen.width <
149             activeScreenWidth;
150
151         // This is handled as an if statement instead of switch because a tap
152         //  can sometimes happen within a single frame
153         if (mPhase == MPhase.idle) { // If the mouse is idle
154             if (b0Down && inActiveArea) {
155                 mouseInfos.Clear(); // Clear the mouseInfos
156                 AddMouseInfo();     // And add a first one
157
158                 // If the mouse was clicked on something, it's a valid MouseDown
159                 if (mouseInfos[0].hit) {    // Something was hit!
160                     MouseDown();            // Call MouseDown()
161                     mPhase = MPhase.down;    // and set the mPhase
162                 }
163
164             }
165         }
166
167         if (mPhase == MPhase.down) { // if the mouse is down
168             AddMouseInfo(); // Add a MouseInfo for this frame
169             if (b0Up) {
170                 // The mouse button was released
171                 MouseTap(); // This was a tap
172                 mPhase = MPhase.idle;
173             }  else if (Time.time - mouseInfos[0].time > mTapTime) {
174                 // If it's been down longer than a tap, this may be a drag, but
175                 //  to be a drag, it must also have moved a certain number of
176                 //  pixels on screen.
177                 float dragDist = (lastMouseInfo.screenLoc -
178                                   mouseInfos[0].screenLoc).magnitude;
179                 if (dragDist >= mDragDist) {
180                     mPhase = MPhase.drag;
181                     //MouseDragStart();
182                 }
183
184                 // However, drag will immediately start after mTapTime if there
185                 //  are no elements selected.
186                 if (selectedElements.Count == 0) {
187                     mPhase = MPhase.drag;
188                     //MouseDragStart();
189                 }
190             }
191         }
192
```

```
193         if (mPhase == MPhase.drag) {
194             AddMouseInfo();
195             if (b0Up) {
196                 // The mouse button was released
197                 MouseDragUp();
198                 mPhase = MPhase.idle;
199             }  else {
200                 MouseDrag(); // Still dragging
201             }
202         }
203         OrbitSelectedElements();
204     }
205 
206     // Pulls info about the Mouse, adds it to mouseInfos, and returns it
207     MouseInfo AddMouseInfo() {
208         MouseInfo mInfo = new MouseInfo();
209         mInfo.screenLoc = Input.mousePosition;
210         mInfo.loc = Utils.mouseLoc; // Gets the position of the mouse at z=0
211         mInfo.ray = Utils.mouseRay; // Gets the ray from the main camera through
212         //  the mouse pointer
213         mInfo.time = Time.time;
214         mInfo.Raycast(); // Default is to raycast with no mask
215 
216         if (mouseInfos.Count == 0) {
217             // If this is the first mouseInfo
218             mouseInfos.Add(mInfo); // Add mInfo to mouseInfos
219         }  else {
220             float lastTime = mouseInfos[mouseInfos.Count-1].time;
221             if (mInfo.time != lastTime) {
222                 // if time has passed since the last mouseInfo
223                 mouseInfos.Add(mInfo); // Add mInfo to mouseInfos
224             }
225             // This time test is necessary because AddMouseInfo() could be
226             //  called twice in one frame
227         }
228         return(mInfo); // Return mInfo as well
229     }
230 
231     public MouseInfo lastMouseInfo {
232         // Access to the latest MouseInfo
233         get {
234             if (mouseInfos.Count == 0) return( null );
235             return( mouseInfos[mouseInfos.Count-1] );
236         }
237     }
238 
239     void MouseDown() {
240         // The mouse was pressed on something (it could be a drag or tap)
241         if (DEBUG) print("Mage.MouseDown()");
242 
243         GameObject clickedGO = mouseInfos[0].hitInfo.collider.gameObject;
244         // ^ If the mouse wasn't clicked on anything, this would throw an error
245         //    because hitInfo would be null. However, we know that MouseDown()
246         //    is only called when the mouse WAS clicking on something, so
247         //    hitInfo is guaranteed to be defined.
248 
249         GameObject taggedParent = Utils.FindTaggedParent(clickedGO);
250         if (taggedParent == null) {
251             actionStartTag = "";
252         }  else {
253             actionStartTag = taggedParent.tag;
254             // ^ this should be either "Ground", "Mage", or "Enemy"
255         }
256     }
```

```
257     void MouseTap() {
258         // Something was tapped like a button
259         if (DEBUG) print("Mage.MouseTap()");
260 
261         // Now this cares what was tapped
262         switch (actionStartTag) {
263         case "Mage":
264             // Do nothing
265             break;
266         case "Ground":
267             // Move to tapped point @ z=0 whether or not an element is selected
268             WalkTo(lastMouseInfo.loc);  // Walk to the latest mouseInfo pos
269             ShowTap(lastMouseInfo.loc); // Show where the player tapped
270             break;
271         }
272     }
273     /*
274     void MouseDragStart() {
275         // The conversion from the wait for a tap to realizing it's a drag
276         if (selectedElements.Count > 0) { // If we're drawing a spell line
277             foreach( MouseInfo mi in mouseInfos ) {
278                 // Add the already-existing points to the line
279                 AddPointToLiner( mi.loc );
280             }
281         }
282     }
283     */
284     void MouseDrag() {
285         // The mouse is being drug across something
286         if (DEBUG) print("Mage.MouseDrag()");
287 
288         // Drag is meaningless unless the mouse started on the ground
289         if (actionStartTag != "Ground") return;
290 
291         // If there is no element selected, the player should follow the mouse
292         if (selectedElements.Count == 0) {
293             // Continuously walk towards the current mouseInfo pos
294             WalkTo(mouseInfos[mouseInfos.Count-1].loc);
295         }  else {
296             // This is a ground spell, so we need to draw a line
297             AddPointToLiner( mouseInfos[mouseInfos.Count-1].loc );
298             // ^ add the most recent MouseInfo.loc to liner
299         }
300     }
301 
302     void MouseDragUp() {
303         // The mouse is released after being drug
304         if (DEBUG) print("Mage.MouseDragUp()");
305 
306         // Drag is meaningless unless the mouse started on the ground
307         if (actionStartTag != "Ground") return;
308 
309         // If there is no element selected, stop walking now
310         if (selectedElements.Count == 0) {
311             // Stop walking when the drag is stopped
312             StopWalking();
313         }  else {
314             CastGroundSpell();
315 
316             // Clear the liner
317             ClearLiner();
318         }
319     }
320 
```

```
321
322     void CastGroundSpell() {
323         // There is not a no-element ground spell, so return
324         if (selectedElements.Count == 0) return;
325
326         // Because this version of the prototype only allows a single element to
327         //  be selected, we can use that 0th element to pick the spell.
328         switch (selectedElements[0].type) {
329         case ElementType.fire:
330             GameObject fireGO;
331             foreach( Vector3 pt in linePts ) { // For each Vector3 in linePts...
332                 // ...create an instance of fireGroundSpellPrefab
333                 fireGO = Instantiate(fireGroundSpellPrefab) as GameObject;
334                 fireGO.transform.parent = spellAnchor;
335                 fireGO.transform.position = pt;
336             }
337             break;
338             //TODO: Add other elements types later
339         }
340         // Clear the selectedElements; they're used by the spell
341         ClearElements();
342     }
343
344     // Walk to a specific position. The position.z is always 0
345     public void WalkTo(Vector3 xTarget) {
346         walkTarget = xTarget;     // Set the point to walk to
347         walkTarget.z = 0;         // Force z=0
348         walking = true;             // Now the Mage is walking
349         Face(walkTarget);         // Look in the direction of the walkTarget
350     }
351
352     public void Face(Vector3 poi) { // Face towards a point of interest
353         Vector3 delta = poi-pos; // Find vector to the point of interest
354         // Use Atan2 to get the rotation around Z that points the X-axis of
355         //  _Mage:CharacterTrans towards poi
356         float rZ = Mathf.Rad2Deg * Mathf.Atan2(delta.y, delta.x);
357         // Set the rotation of characterTrans (doesn't actually rotate _Mage)
358         characterTrans.rotation = Quaternion.Euler(0,0,rZ);
359     }
360
361     public void StopWalking() { // Stops the _Mage from walking
362         walking = false;
363         GetComponent<Rigidbody>().velocity = Vector3.zero;
364     }
365
366     void FixedUpdate () { // Happens every physics step (i.e. 60 times/second)
367         if (invincibleBool) {
368             // Get number [0..1]
369             float blinkU = (Time.time - damageTime)/invincibleDur;
370             blinkU *= invTimesToBlink; // Multiply by times to blink
371             blinkU %= 1.0f;
372             // ^ Modulo 1.0 gives us the whole number left when dividing blinkU
373             //    by 1.0. For example: 3.85f % 1.0f is 0.85f
374             bool visible = (blinkU > 0.5f);
375             if (Time.time - damageTime > invincibleDur) {
376                 invincibleBool = false;
377                 visible = true; // Just to be sure
378             }
379             // Making the GameObject inactive makes it invisible
380             viewCharacterTrans.gameObject.SetActive(visible);
381         }
382
383
384
```

```
385         if (knockbackBool) {
386             if (Time.time - damageTime > knockbackDur) {
387                 knockbackBool = false;
388             }
389             float knockbackSpeed = knockbackDist/knockbackDur;
390             vel = knockbackDir * knockbackSpeed;
391             return; // Returns to avoid walking code below
392         }
393 
394         if (walking) { // If Mage is walking
395             if ( (walkTarget-pos).magnitude < speed*Time.fixedDeltaTime ) {
396                 // If Mage is very close to walkTarget, just stop there
397                 pos = walkTarget;
398                 StopWalking();
399             }  else {
400                 // Otherwise, move towards walkTarget
401                 GetComponent<Rigidbody>().velocity = (walkTarget-pos).normalized * speed;
402             }
403         }  else {
404             // If not walking, velocity should be zero
405             GetComponent<Rigidbody>().velocity = Vector3.zero;
406         }
407     }
408 
409     void OnCollisionEnter( Collision coll ) {
410         GameObject otherGO = coll.gameObject;
411 
412         // Colliding with a wall can also stop walking
413         Tile ti = otherGO.GetComponent<Tile>();
414         if (ti != null) {
415             if (ti.height > 0) { // If ti.height is > 0
416                 // Then this ti is a wall, and Mage should stop
417                 StopWalking();
418             }
419         }
420 
421         // See if it's an EnemyBug
422         EnemyBug bug = coll.gameObject.GetComponent<EnemyBug>();
423         // If otherGO is an EnemyBug, pass bug to CollisionDamage(), which will
424         //  interpret it as an Enemy
425         if (bug != null) CollisionDamage(bug);
426     }
427 
428     void OnTriggerEnter(Collider other) {
429         EnemySpiker spiker = other.GetComponent<EnemySpiker>();
430         if (spiker != null) {
431             // CollisionDamage() will see spiker as an Enemy
432             CollisionDamage(spiker);
433         }
434     }
435 
436     void CollisionDamage(Enemy enemy) {
437         // Don't take damage if you're already invincible
438         if (invincibleBool) return;
439 
440         // The Mage has been hit by an enemy
441         StopWalking();
442         ClearInput();
443 
444         health -= enemy.touchDamage; // Take a variable amount of damage
445         if (health <= 0) {
446             Die();
447             return;
448         }
```

```
449
450         damageTime = Time.time;
451         knockbackBool = true;
452         knockbackDir = (pos - enemy.pos).normalized;
453         invincibleBool = true;
454     }
455
456     // The Mage dies
457     void Die() {
458         Application.LoadLevel(0); // Reload the level
459         // ^ Eventually, you'll want to do something more elegant
460     }
461
462     // Show where the player tapped
463     public void ShowTap(Vector3 loc) {
464         GameObject go = Instantiate(tapIndicatorPrefab) as GameObject;
465         go.transform.position = loc;
466     }
467
468     // Chooses an Element_Sphere of elType and adds it to selectedElements
469     public void SelectElement(ElementType elType) {
470         if (elType == ElementType.none) {    // If it's the none element...
471             ClearElements();                 //  then clear all Elements
472             return;                          //  and return
473         }
474
475         if (maxNumSelectedElements == 1) {
476             // If only one can be selected, clear the existing one...
477             ClearElements(); // ...so it can be replaced
478         }
479
480         // Can't select more than maxNumSelectedElements simultaneously
481         if (selectedElements.Count >= maxNumSelectedElements) return;
482
483         // It's okay to add this element
484         GameObject go = Instantiate(elementPrefabs[(int) elType]) as GameObject;
485         // ^ Note the typecast from ElementType to int in the line above
486         Element el = go.GetComponent<Element>();
487         el.transform.parent = this.transform;
488
489         selectedElements.Add(el); // Add el to the list of selectedElements
490     }
491
492     // Clears all elements from selectedElements and destroys their GameObjects
493     public void ClearElements() {
494         foreach (Element el in selectedElements) {
495             // Destroy each GameObject in the list
496             Destroy(el.gameObject);
497         }
498         selectedElements.Clear(); // and clear the list
499     }
500
501     // Called every Update() to orbit the elements around
502     void OrbitSelectedElements() {
503         // If there are none selected, just return
504         if (selectedElements.Count == 0) return;
505
506         Element el;
507         Vector3 vec;
508         float theta0, theta;
509         float tau = Mathf.PI*2; // tau is 360°n radians (i.e. 6.283...)
510
511        // Divide the circle into the number of elements that are orbiting
512         float rotPerElement = tau / selectedElements.Count;
```

```
513         // The base rotation angle (theta0) is set based on time
514         theta0 = elementRotSpeed * Time.time * tau;
515         for (int i=0; i<selectedElements.Count; i++) {
516             // Determine the rotation angle for each element
517             theta = theta0 + i*rotPerElement;
518             el = selectedElements[i];
519             // Use simple trigonometry to turn the angle into a unit vector
520             vec = new Vector3(Mathf.Cos(theta),Mathf.Sin(theta),0);
521             // Multiply that unit vector by the elementRotDist
522             vec *= elementRotDist;
523             // Raise the element to waist height.
524             vec.z = -0.5f;
525             el.lPos = vec;    // Set the position of the Element_Sphere
526         }
527     }
528 
529 
530     //---------------- LineRenderer Code ----------------//
531 
532     // Add a new point to the line. This ignores the point if it's too close to
533     //  existing ones and adds extra points if it's too far away
534     void AddPointToLiner(Vector3 pt) {
535         pt.z = lineZ; // Set the z of the pt to lineZ to elevate it slightly
536         //  above the ground
537 
538         //linePts.Add(pt); // Comment out or delete these two lines
539         //UpdateLiner();
540 
541         // Always add the point if linePts is empty
542         if (linePts.Count == 0) {
543             linePts.Add (pt);
544             totalLineLength = 0;
545             return; // ...but wait for a second point to enable the LineRenderer
546         }
547 
548         // If the line is too long already, return
549         if (totalLineLength > lineMaxLength) return;
550 
551         // If there is a previous point (pt0), then find how far pt is from it
552         Vector3 pt0 = linePts[linePts.Count-1]; // Get the last point in linePts
553         Vector3 dir = pt-pt0;
554         float delta = dir.magnitude;
555         dir.Normalize();
556 
557         totalLineLength += delta;
558 
559         // If it's less than the min distance
560         if ( delta < lineMinDelta ) {
561             // ...then it's too close; don't add it
562             return;
563         }
564 
565         // If it's further than the max distance then extra points...
566         if (delta > lineMaxDelta) {
567             // ...then add extra points in between
568             float numToAdd = Mathf.Ceil(delta/lineMaxDelta);
569             float midDelta = delta/numToAdd;
570             Vector3 ptMid;
571             for (int i=1; i<numToAdd; i++) {
572                 ptMid = pt0+(dir*midDelta*i);
573                 linePts.Add(ptMid);
574             }
575         }
576 
```

```
577
578         linePts.Add(pt); // Add the point
579         UpdateLiner();   // And finally update the line
580     }
581
582     // Update the LineRenderer with the new points
583     public void UpdateLiner() {
584         // Get the type of the selectedElement
585         int el = (int) selectedElements[0].type;
586
587         // Set the line color based on that type
588         liner.SetColors(elementColors[el],elementColors[el]);
589
590         // Update the representation of the ground spell about to be cast
591         liner.SetVertexCount(linePts.Count);  // Set the number of vertices
592         for (int i=0; i<linePts.Count; i++) {
593             liner.SetPosition(i, linePts[i]); // Set each vertex
594         }
595         liner.enabled = true;                  // Enable the LineRenderer
596     }
597
598     public void ClearLiner() {
599         liner.enabled = false;    // Disable the LineRenderer
600         linePts.Clear();        // and clear all linePts
601     }
602
603     // Stop any active drag or other mouse input
604     public void ClearInput() {
605         mPhase = MPhase.idle;
606     }
607
608
609
610 }
```

Prototools/PT_MonoBehaviour.cs

```
using UnityEngine;
using System.Collections;

// This class includes several properties to enable easier access to common fields
public class PT_MonoBehaviour : MonoBehaviour {

    public Vector3 pos {
        get { return( transform.position ); }
        set { transform.position = value; }
    }

    public Vector3 lPos {
        get { return( transform.localPosition ); }
        set { transform.localPosition = value; }
    }

    public Vector3 rot {
        get { return( transform.eulerAngles ); }
        set { transform.rotation = Quaternion.Euler(value); }
    }

    public Color color {
        get { return( this.GetComponent<Renderer>().material.color ); }
        set { this.GetComponent<Renderer>().material.color = value; }
    }

    public Material mat {
        get { return( this.GetComponent<Renderer>().material); }
        set { this.GetComponent<Renderer>().material = value; }
    }

    public Vector3 scale {
        get { return( transform.localScale ); }
        set { transform.localScale = value; }
    }

    public float scaleF {
        get { return( Mathf.Max( scale.x, scale.y, scale.z ) ); }
        set { scale = Vector3.one * value; }
    }

    public Vector3 vel {
        get {
            if (GetComponent<Rigidbody>() != null) {
                return( GetComponent<Rigidbody>().velocity );
            }  else {
                return( Vector3.zero );
            }
        }
        set {
            if (GetComponent<Rigidbody>() != null) {
                GetComponent<Rigidbody>().velocity = value;
            }
        }
    }

}
```

Tile.cs

```
using UnityEngine;
using System.Collections;

public class Tile : PT_MonoBehaviour {
// public fields
    public string        type;

// Hidden private fields
    private string        _tex;
    private int           _height = 0;
    private Vector3       _pos;

// Properties with get{} and set{}

    // height moves the Tile up or down. Walls have height=1
    public int            height {
        get { return( _height ); }
        set {
            _height = value;
            AdjustHeight();
        }
    }

    // Sets the texture of the Tile based on a string
    // It requires LayoutTiles, so it's commented out for now

    public string        tex {
        get {
            return( _tex );
        }
        set {
            _tex = value;
            name = "TilePrefab_"+_tex; // Sets the name of this GameObject
            Texture2D t2D = LayoutTiles.S.GetTileTex(_tex);
            if (t2D == null) {
                Utils.tr("ERROR","Tile.type{set}=",value,
                  "No matching Texture2D in LayoutTiles.S.tileTextures!");
            }  else {
                GetComponent<Renderer>().material.mainTexture = t2D;
            }
        }
    }

    // Uses the "new" keyword to replace the pos inherited from PT_MonoBehaviour
    // Without the "new" keyword, the two properties would conflict
    new public Vector3 pos {
        get { return( _pos ); }
        set {
            _pos = value;
            AdjustHeight();
        }
    }

// Methods
    public void AdjustHeight() {
        // Moves the block up or down based on _height
        Vector3 vertOffset = Vector3.back*(_height-0.5f);
        // The -0.5f shifts the Tile down 0.5 units so that it's top surface is
        //  at z=0 when pos.z=0 and height=0
        transform.position = _pos+vertOffset;
    }

}
```

Prototools/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 {
    static public bool DEBUG = true;

    // Returns the maximum value for a Vector3, which can be used to return a unique, identifiable Vector3 value
    static public Vector3 maxVector3 {
        get { return( new Vector3(float.MaxValue, float.MaxValue, float.MaxValue) ); }
    }

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

    // Creates bounds that encapsulate 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 );
    }

    // Get the location of the mouse in World coordinates (at z=0)
    static public Vector3 mouseLoc {
        get {
            Vector3 loc = Input.mousePosition;
            loc.z = -Camera.main.transform.position.z;
            loc = Camera.main.ScreenToWorldPoint(loc);
            return(loc);
        }
    }
    static public Vector3 MouseLoc {
        get {
            return(MouseLoc);
        }
    }

    static public Ray mouseRay {
        get {
            Vector3 loc = Input.mousePosition;
            Ray r = Camera.main.ScreenPointToRay(loc);
            return( r );
        }
    }
    static public Ray MouseRay {
        get { return( mouseRay ); }
    }

```

```
129     // Test to see whether Bounds are on screen.
130     public static Vector3 ScreenBoundsCheck(Bounds bnd, BoundsTest test =
          ➥BoundsTest.center) {
131         // Call the more generic BoundsInBoundsCheck with camBounds as bigB
132         return( BoundsInBoundsCheck( camBounds, bnd, test ) );
133     }
134
135     // Tests to see whether lilB is inside bigB
136     public static Vector3 BoundsInBoundsCheck( Bounds bigB, Bounds lilB, BoundsTest test
          ➥= BoundsTest.onScreen ) {
137         // Get the center of lilB
138         Vector3 pos = lilB.center;
139
140         // Initialize the offset at [0,0,0]
141         Vector3 off = Vector3.zero;
142
143         switch (test) {
144 // The center test determines what off (offset) would have to be applied to lilB to move
      ➥its center back inside bigB
145         case BoundsTest.center:
146             // if the center is contained, return Vector3.zero
147             if ( bigB.Contains( pos ) ) {
148                 return( Vector3.zero );
149             }
150             // if not contained, find the offset
151             if (pos.x > bigB.max.x) {
152                 off.x = pos.x - bigB.max.x;
153             } else if (pos.x < bigB.min.x) {
154                 off.x = pos.x - bigB.min.x;
155             }
156             if (pos.y > bigB.max.y) {
157                 off.y = pos.y - bigB.max.y;
158             } else if (pos.y < bigB.min.y) {
159                 off.y = pos.y - bigB.min.y;
160             }
161             if (pos.z > bigB.max.z) {
162                 off.z = pos.z - bigB.max.z;
163             } else if (pos.z < bigB.min.z) {
164                 off.z = pos.z - bigB.min.z;
165             }
166             return( off );
167
168 // The onScreen test determines what off would have to be applied to keep all of lilB
      ➥inside bigB
169         case BoundsTest.onScreen:
170             // find whether bigB contains all of lilB
171             if ( bigB.Contains( lilB.min ) && bigB.Contains( lilB.max ) ) {
172                 return( Vector3.zero );
173             }
174             // if not, find the offset
175             if (lilB.max.x > bigB.max.x) {
176                 off.x = lilB.max.x - bigB.max.x;
177             } else if (lilB.min.x < bigB.min.x) {
178                 off.x = lilB.min.x - bigB.min.x;
179             }
180             if (lilB.max.y > bigB.max.y) {
181                 off.y = lilB.max.y - bigB.max.y;
182             } else if (lilB.min.y < bigB.min.y) {
183                 off.y = lilB.min.y - bigB.min.y;
184             }
185             if (lilB.max.z > bigB.max.z) {
186                 off.z = lilB.max.z - bigB.max.z;
187             } else if (lilB.min.z < bigB.min.z) {
188                 off.z = lilB.min.z - bigB.min.z;
189             }
190             return( off );
191
```

```
// 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) ) );
    }

```

```
321     // The same two functions for Vector2
322     static public Vector2 Bezier( float u, List<Vector2> vList ) {
323         // If there is only one element in vList, return it
324         if (vList.Count == 1) {
325             return( vList[0] );
326         }
327         // Otherwise, create vListR, which is all but the 0th element of vList
328         // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
329         List<Vector2> vListR =  vList.GetRange(1, vList.Count-1);
330         // And create vListL, which is all but the last element of vList
331         // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
332         List<Vector2> vListL = vList.GetRange(0, vList.Count-1);
333         // The result is the Lerp of these two shorter Lists
334         Vector2 res = Lerp( Bezier(u, vListL), Bezier(u, vListR), u );
335         return( res );
336     }
337 
338     // This version allows an Array or a series of Vector2s as input
339     static public Vector2 Bezier( float u, params Vector2[] vecs ) {
340         return( Bezier( u, new List<Vector2>(vecs) ) );
341     }
342 
343 
344     // The same two functions for float
345     static public float Bezier( float u, List<float> vList ) {
346         // If there is only one element in vList, return it
347         if (vList.Count == 1) {
348             return( vList[0] );
349         }
350         // Otherwise, create vListR, which is all but the 0th element of vList
351         // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
352         List<float> vListR =  vList.GetRange(1, vList.Count-1);
353         // And create vListL, which is all but the last element of vList
354         // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
355         List<float> vListL = vList.GetRange(0, vList.Count-1);
356         // The result is the Lerp of these two shorter Lists
357         float res = Lerp( Bezier(u, vListL), Bezier(u, vListR), u );
358         return( res );
359     }
360 
361     // This version allows an Array or a series of floats as input
362     static public float Bezier( float u, params float[] vecs ) {
363         return( Bezier( u, new List<float>(vecs) ) );
364     }
365 
366 
367     // The same two functions for Quaternion
368     static public Quaternion Bezier( float u, List<Quaternion> vList ) {
369         // If there is only one element in vList, return it
370         if (vList.Count == 1) {
371             return( vList[0] );
372         }
373         // Otherwise, create vListR, which is all but the 0th element of vList
374         // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
375         List<Quaternion> vListR =  vList.GetRange(1, vList.Count-1);
376         // And create vListL, which is all but the last element of vList
377         // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
378         List<Quaternion> vListL = vList.GetRange(0, vList.Count-1);
379         // The result is the Slerp of these two shorter Lists
380         // It's possible that Quaternion.Slerp may clamp u to [0..1] :(
381         Quaternion res = Quaternion.Slerp( Bezier(u, vListL), Bezier(u, vListR), u );
382         return( res );
383     }
384 
```

```
385     // This version allows an Array or a series of floats as input
386     static public Quaternion Bezier( float u, params Quaternion[] vecs ) {
387         return( Bezier( u, new List<Quaternion>(vecs) ) );
388     }
389
390
391     //============================ Trace & Logging Functions ============================
392
393     static public void tr(params object[] objs) {
394         string s = objs[0].ToString();
395         for (int i=1; i<objs.Length; i++) {
396             s += "\t"+objs[i].ToString();
397         }
398         print (s);
399     }
400
401     static public void trd(params object[] objs) {
402         if (DEBUG) {
403             tr (objs);
404         }
405     }
406
407
408
409     //============================ Math Functions ============================
410
411     static public float RoundToPlaces(float f, int places=2) {
412         float mult = Mathf.Pow(10,places);
413         f *= mult;
414         f = Mathf.Round (f);
415         f /= mult;
416         return(f);
417     }
418
419     static public string AddCommasToNumber(float f, int places=2) {
420         int n = Mathf.RoundToInt(f);
421         f -= n;
422         f = RoundToPlaces(f,places);
423         string str = AddCommasToNumber( n );
424         str += "."+(f*Mathf.Pow(10,places));
425         return( str );
426     }
427     static public string AddCommasToNumber(int n) {
428         int rem;
429         int div;
430         string res = "";
431         string rems;
432         while (n>0) {
433             rem = n % 1000;
434             div = n / 1000;
435             rems = rem.ToString();
436
437             while (div>0 && rems.Length<3) {
438                 rems = "0"+rems;
439             }
440             // NOTE: It is somewhat faster to use a StringBuilder or a List<String> which
                ➥is then concatenated using String.Join().
441             if (res == "") {
442                 res = rems;
443             }  else {
444                 res = rems + "," + res.ToString();
445             }
446             n = div;
447         }
448
```

```
449         if (res == "") res = "0";
450         return( res );
451     }
452
453 }
454
455
456
457
458 //============================ Easing Classes ============================
459 [System.Serializable]
460 public class EasingCachedCurve {
461     public List<string>        curves =    new List<string>();
462     public List<float>        mods =         new List<float>();
463 }
464
465
466 public class Easing {
467     static public string Linear =         ",Linear|";
468     static public string In =             ",In|";
469     static public string Out =            ",Out|";
470     static public string InOut =         ",InOut|";
471     static public string Sin =            ",Sin|";
472     static public string SinIn =        ",SinIn|";
473     static public string SinOut =        ",SinOut|";
474
475
476     static public Dictionary<string,EasingCachedCurve> cache;
477     // This is a cache for the information contained in the complex strings
478     //   that can be passed into the Ease function. The parsing of these
479     //   strings is most of the effort of the Ease function, so each time one
480     //   is parsed, the result is stored in the cache to be recalled much
481     //   faster than a parse would take.
482     // Need to be careful of memory leaks, which could be a problem if several
483     //   million unique easing parameters are called
484
485
486     static public float Ease( float u, params string[] curveParams ) {
487         // Set up the cache for curves
488         if (cache == null) {
489             cache = new Dictionary<string, EasingCachedCurve>();
490         }
491
492         float u2 = u;
493         foreach ( string curve in curveParams ) {
494             // Check to see if this curve is already cached
495             if (!cache.ContainsKey(curve)) {
496                 // If not, parse and cache it
497                 EaseParse(curve);
498             }
499             // Call the cached curve
500             u2 = EaseP( u2, cache[curve] );
501         }
502         return( u2 );
503         /*
504
505             // It's possible to pass in several comma-separated curves
506             string[] curvesA = curves.Split(',');
507             foreach (string curve in curvesA) {
508                 if (curve == "") continue;
509                 //string[] curveA =
510             }
511
512         }
```

```
513         //string[] curve = func.Split(',');
514
515         foreach (string curve in curves) {
516
517         }
518
519         string[] funcSplit;
520         foreach (string f in funcs) {
521             funcSplit = f.Split('|');
522
523         }
524         */
525     }
526
527     static private void EaseParse( string curveIn ) {
528         EasingCachedCurve ecc = new EasingCachedCurve();
529         // It's possible to pass in several comma-separated curves
530         string[] curves = curveIn.Split(',');
531         foreach (string curve in curves) {
532             if (curve == "") continue;
533             // Split each curve on | to find curve and mod
534             string[] curveA = curve.Split('|');
535             ecc.curves.Add(curveA[0]);
536             if (curveA.Length == 1 || curveA[1] == "") {
537                 ecc.mods.Add(float.NaN);
538             }  else {
539                 float parseRes;
540                 if ( float.TryParse(curveA[1], out parseRes) ) {
541                     ecc.mods.Add( parseRes );
542                 }  else {
543                     ecc.mods.Add( float.NaN );
544                 }
545             }
546         }
547         cache.Add(curveIn, ecc);
548     }
549
550
551     static public float Ease( float u, string curve, float mod ) {
552         return( EaseP( u, curve, mod ) );
553     }
554
555     static private float EaseP( float u, EasingCachedCurve ec ) {
556         float u2 = u;
557         for (int i=0; i<ec.curves.Count; i++) {
558             u2 = EaseP( u2, ec.curves[i], ec.mods[i] );
559         }
560         return( u2 );
561     }
562
563     static private float EaseP( float u, string curve, float mod ) {
564         float u2 = u;
565
566         switch (curve) {
567         case "In":
568             if (float.IsNaN(mod)) mod = 2;
569             u2 = Mathf.Pow(u, mod);
570             break;
571
572         case "Out":
573             if (float.IsNaN(mod)) mod = 2;
574             u2 = 1 - Mathf.Pow( 1-u, mod );
575             break;
576
```

```
577
578         case "InOut":
579             if (float.IsNaN(mod)) mod = 2;
580             if ( u <= 0.5f ) {
581                 u2 = 0.5f * Mathf.Pow( u*2, mod );
582             }  else {
583                 u2 = 0.5f + 0.5f * (  1 - Mathf.Pow( 1-(2*(u-0.5f)), mod )  );
584             }
585             break;
586
587         case "Sin":
588             if (float.IsNaN(mod)) mod = 0.15f;
589             u2 = u + mod * Mathf.Sin( 2*Mathf.PI*u );
590             break;
591
592         case "SinIn":
593             // mod is ignored for SinIn
594             u2 = 1 - Mathf.Cos( u * Mathf.PI * 0.5f );
595             break;
596
597         case "SinOut":
598             // mod is ignored for SinOut
599             u2 = Mathf.Sin( u * Mathf.PI * 0.5f );
600             break;
601
602         case "Linear":
603         default:
604             // u2 already equals u
605             break;
606         }
607
608         return( u2 );
609     }
610
611 }
```