Bartok

Pedro Antonio González Calero (pedro@fdi.ucm.es)

Contents

El juego

Diseño

Vamos a desarrollar una versión del juego de cartas Bartok (https://en.wikipedia.org/wiki/Bartok_(card_game)), que en su versión más básica se puede jugar aquí: http://gaia.fdi.ucm.es/files/people/pedro/slides/dev/05solitaire/05solitaire.html.

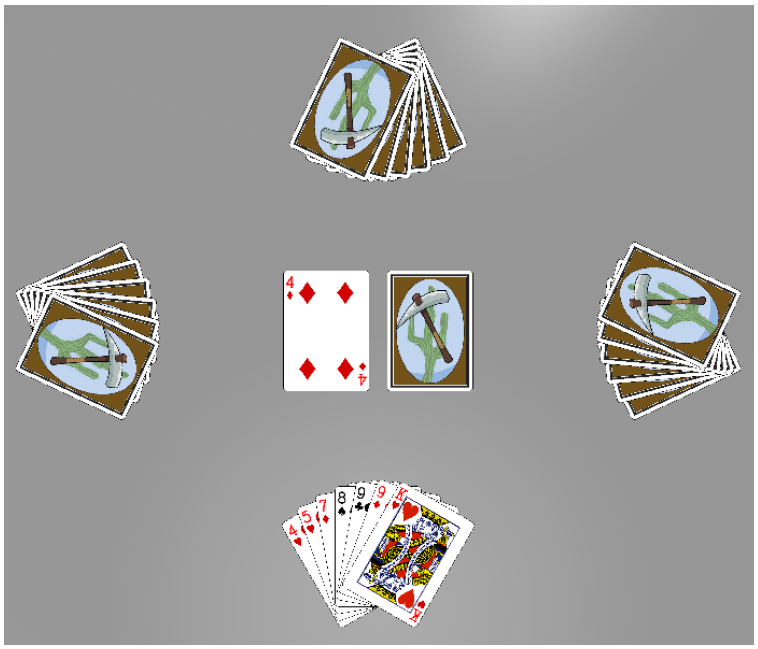

Figure 1: Bartok

Este fragmento de XML define la disposición de las cartas:

```
<xml>
    <!-- This file includes info for laying out the Bartok card game. -->

    <!-- The multiplier is multiplied by the x and y attributes below. -->
    <!-- This determines how loose or tight the layout is. -->
    <multiplier x="1" y="1" />

    <!-- This positions the draw pile and staggers it -->
    <slot type="drawpile" x="1.5" y="0" xstagger="0.05" layer="1"/>

    <!-- This positions the discard pile -->
    <slot type="discardpile" x="-1.5" y="0" layer="2"/>

    <!-- This positions the target card -->
    <slot type="target" x="-1.5" y="0" layer="4"/>

    <!-- These slots are for the four hands held by the four players -->
    <slot type="hand" x="0"   y="-8" rot="0"   player="1" layer="3"/>
    <slot type="hand" x="-10" y="0"  rot="270" player="2" layer="3"/>
    <slot type="hand" x="0"   y="8"  rot="180" player="3" layer="3"/>
```

```
    <slot type="hand" x="10"  y="0"  rot="90"  player="4" layer="3"/>
</xml>
```

Rotación de las cartas

Este es el código encargado de la animación de las cartas:

```
// CBState includes both states for the game and to___ states for movement
public enum CBState {
    drawpile,
    toHand,
    hand,
    toTarget,
    target,
    discard,
    to,
    idle
}

// CardBartok extends Card just as CardProspector did.
public class CardBartok : Card {
    // These static fields are used to set values that will be the same
    //  for all instances of CardBartok
    static public float       MOVE_DURATION = 0.5f;
    static public string      MOVE_EASING = Easing.InOut;
    static public float       CARD_HEIGHT = 3.5f;
    static public float       CARD_WIDTH = 2f;

    public CBState            state = CBState.drawpile;

    // Fields to store info the card will use to move and rotate
    public List<Vector3>      bezierPts;
    public List<Quaternion>   bezierRots;
    public float              timeStart, timeDuration; // declares 2 fields

    // When the card is done moving, it will call reportFinishTo.SendMessage()
    public GameObject         reportFinishTo = null;

    // MoveTo tells the card to interpolate to a new position and rotation
    public void MoveTo(Vector3 ePos, Quaternion eRot) {
        // Make new interpolation lists for the card.
        // Position and Rotation will each have only two points.
        bezierPts = new List<Vector3>();
        bezierPts.Add ( transform.localPosition );  // Current position
        bezierPts.Add ( ePos );                     // New position
        bezierRots = new List<Quaternion>();
        bezierRots.Add ( transform.rotation );      // Current rotation
        bezierRots.Add ( eRot );                    // New rotation

        // If timeStart is 0, then it's set to start immediately,
        //  otherwise, it starts at timeStart. This way, if timeStart is
        //  already set, it won't be overwritten.
```

```
    if (timeStart == 0) {
        timeStart = Time.time;
    }
    // timeDuration always starts the same but can be altered later
    timeDuration = MOVE_DURATION;

    // Setting state to either toHand or toTarget will be handled by the
    //  calling method
    state  = CBState.to;
}
// This overload of MoveTo doesn't require a rotation argument
public void MoveTo(Vector3 ePos) {
    MoveTo(ePos, Quaternion.identity);
}

void Update() {
    switch (state) {
    // All the to___ states are ones where the card is interpolating
    case CBState.toHand:
    case CBState.toTarget:
    case CBState.to:
        // Get u from the current time and duration
        // u ranges from 0 to 1 (usually)
        float u = (Time.time - timeStart)/timeDuration;

        // Use Easing class from Utils to curve the u value
        float uC = Easing.Ease (u, MOVE_EASING);

        if (u<0) { // If u<0, then we shouldn't move yet.
            // Stay at the initial position
            transform.localPosition = bezierPts[0];
            transform.rotation = bezierRots[0];
            return;
        } else if (u>=1) { // If u>=1, we're finished moving
            uC = 1; // Set uC=1 so we don't overshoot
            // Move from the to___ state to the following state
            if (state == CBState.toHand)   state = CBState.hand;
            if (state == CBState.toTarget) state = CBState.toTarget;
            if (state == CBState.to)       state = CBState.idle;
            // Move to the final position
            transform.localPosition = bezierPts[bezierPts.Count-1];
            transform.rotation = bezierRots[bezierPts.Count-1];
            // Reset timeStart to 0 so it gets overwritten next time
            timeStart = 0;

            if (reportFinishTo != null) { //If there's a callback GameObject
                // ... then use SendMessage to call the CBCallback method
                //  with this as the parameter.
                reportFinishTo.SendMessage("CBCallback", this);
                // After calling SendMessage(), reportFinishTo must be set
                //  to null so that it the card doesn't continue to report
                //  to the same GameObject every subsequent time it moves.
                reportFinishTo = null;
            } else { // If there is nothing to callback
```

```
                // Do nothing
            }
        } else { // 0<=u<1, which means that this is interpolating now
            // Use Bezier curve to move this to the right point
            Vector3 pos = Utils.Bezier(uC, bezierPts);
            transform.localPosition = pos;
            Quaternion rotQ = Utils.Bezier(uC, bezierRots);
            transform.rotation = rotQ;

        }
        break;

    }
}
```

y para que esto funcione es necesario añadir un par de funciones a *Utils.cs*:

```
// The same two functions for Quaternion
static public Quaternion Bezier( float u, List<Quaternion> vList ) {
    // If there is only one element in vList, return it
    if (vList.Count == 1) {
        return( vList[0] );
    }
    // Otherwise, create vListR, which is all but the 0th element of vList
    // e.g. if vList = [0,1,2,3,4] then vListR = [1,2,3,4]
    List<Quaternion> vListR =  vList.GetRange(1, vList.Count-1);
    // And create vListL, which is all but the last element of vList
    // e.g. if vList = [0,1,2,3,4] then vListL = [0,1,2,3]
    List<Quaternion> vListL = vList.GetRange(0, vList.Count-1);
    // The result is the Slerp of these two shorter Lists
    // It's possible that Quaternion.Slerp may clamp u to [0..1] :(
    Quaternion res = Quaternion.Slerp(Bezier(u,vListL), Bezier(u,vListR), u);
    return( res );
}

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

Referencias

- Jeremy Gibson. Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#. Addison-Wesley Professional, 2014