Prospector Solitaire

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

Contents

El juego

Diseño

Vamos a desarrollar una versión del solitario de cartas Tri-Peaks y además dejaremos preparada la infraestructura para desarrollar otros juegos de cartas.

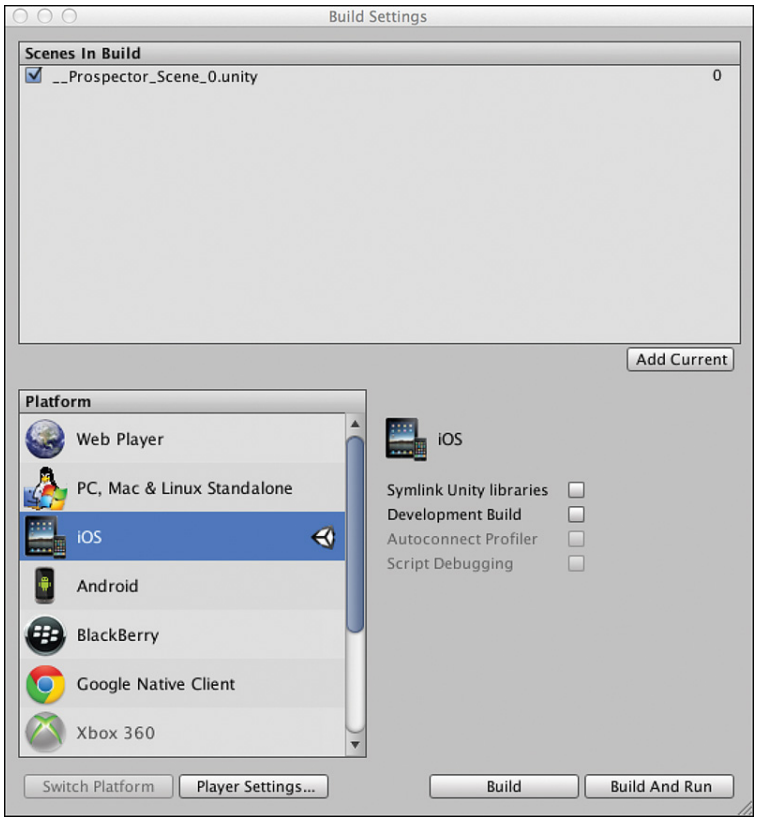

Figure 1: Añadimos la escena actual a la configuración de Build

- Empezamos importando el paquete `04solitaire.unitypackage`
- en la configuración de Build añadimos la escena actual, seleccionamos iOS en la lista de plataformas y pulsamos *Switch Platform*
- importamos los sprites, seleccionando en el inspector el valor *Texture Type* para el atributo *Sprite* y *Truecolor* en *Format*
- *Letters* se importa con el valor *Multiple* en *Sprite Mode* y se entra en el editor de sprites para dividir el atlas manualmente en celdas de 32x32 píxeles

- Creamos tres scripts: Card, Deck y Prospector

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

public class Card : MonoBehaviour {
```

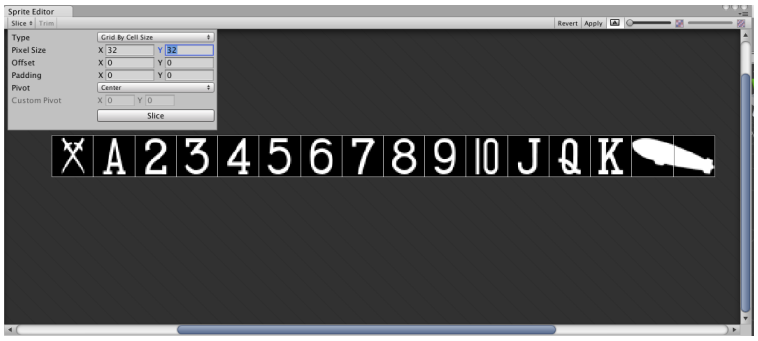

Figure 2: Generación de los sprites de las letras

```
}

[System.Serializable]
public class Decorator {
    // This class stores information about each decorator or pip from DeckXML
    public string    type; // For card pips, type = "pip"
    public Vector3   loc;  // The location of the Sprite on the Card
    public bool      flip = false;  // Whether to flip the Sprite vertically
    public float     scale = 1f;    // The scale of the Sprite
}

[System.Serializable]
public class CardDefinition {
    // This class stores information for each rank of card
    public string             face; // Sprite to use for each face card
    public int                rank; // The rank (1-13) of this card
    public List<Decorator>    pips = new List<Decorator>();  // Pips used
    // Because decorators (from the XML) are used the same way on every card in
    //   the deck, pips only stores information about the pips on numbered cards
}

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

public class Deck : MonoBehaviour {

    public bool _________________;

    public PT_XMLReader           xmlr;

    // InitDeck is called by Prospector when it is ready
    public void InitDeck(string deckXMLText) {
        ReadDeck(deckXMLText);
    }

    // ReadDeck parses the XML file passed to it into CardDefinitions
    public void ReadDeck(string deckXMLText) {
        xmlr = new PT_XMLReader(); // Create a new PT_XMLReader
        xmlr.Parse(deckXMLText);   // Use that PT_XMLReader to parse DeckXML
```

```
        // This prints a test line to show you how xmlr can be used.
        string s = "xml[0] decorator[0] ";
        s += "type="+xmlr.xml["xml"][0]["decorator"][0].att("type");
        s += " x="+xmlr.xml["xml"][0]["decorator"][0].att("x");
        s += " y="+xmlr.xml["xml"][0]["decorator"][0].att("y");
        s += " scale="+xmlr.xml["xml"][0]["decorator"][0].att("scale");
        print(s);

    }
}
```

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

public class Prospector : MonoBehaviour {
    static public Prospector    S;

    public Deck                 deck;
    public TextAsset            deckXML;

    void Awake() {
        S = this; // Set up a Singleton for Prospector
    }

    void Start () {
        deck = GetComponent<Deck>(); // Get the Deck
        deck.InitDeck(deckXML.text); // Pass DeckXML to it
    }
}
```

- Las cartas se definen a partir de un XML que establece dónde se ubican cada uno de los sprites para cada una de las 13 cartas que componen un palo

```
<xml>
    <!-- decorators appear on every card. They are the suit and rank in the corners. -->
    <decorator type="letter" x="-1.05" y="1.42" z="0" flip="0" scale="1.25"/>
    <decorator type="suit" x="-1.05" y="1.03" z="0" flip="0" scale="0.4"/>
    <decorator type="suit" x="1.05" y="-1.03" z="0" flip="1" scale="0.4"/>
    <decorator type="letter" x="1.05" y="-1.42" z="0" flip="1" scale="1.25"/>
    <!-- A list of all cards that defines where pips are placed. -->
    <card rank="1">
        <pip x="0" y="0" z="0" flip="0" scale="2"/>
    </card>
    <card rank="2">
        <pip x="0" y="1.1" z="0" flip="0"/>
        <pip x="0" y="-1.1" z="0" flip="1"/>
    </card>
    <card rank="3">
        <pip x="0" y="1.1" z="0" flip="0"/>
        <pip x="0" y="0" z="0" flip="0"/>
        <pip x="0" y="-1.1" z="0" flip="1"/>
```

```
    </card>

...

    <card rank="11" face="FaceCard_11"/>
    <card rank="12" face="FaceCard_12"/>
    <card rank="13" face="FaceCard_13"/>
</xml>
```

- *Prospector* carga el archivo *deckXML* y se lo pasa a *Deck* que va generando las cartas a partir de su contenido

```
public class Deck : MonoBehaviour {

    public bool _________________;

    public PT_XMLReader           xmlr;
    public List<string>               cardNames;
    public List<Card>                 cards;
    public List<Decorator>            decorators;
    public List<CardDefinition>       cardDefs;
    public Transform                  deckAnchor;
    public Dictionary<string,Sprite>  dictSuits;


    // ReadDeck parses the XML file passed to it into CardDefinitions
    public void ReadDeck(string deckXMLText) {
        xmlr = new PT_XMLReader(); // Create a new PT_XMLReader
        xmlr.Parse(deckXMLText);   // Use that PT_XMLReader to parse DeckXML


        // Read decorators for all Cards
        decorators = new List<Decorator>(); // Init the List of Decorators
        // Grab a PT_XMLHashList of all <decorator>s in the XML file
        PT_XMLHashList xDecos = xmlr.xml["xml"][0]["decorator"];
        Decorator deco;
        for (int i=0; i<xDecos.Count; i++) {
            // For each <decorator> in the XML
            deco = new Decorator(); // Make a new Decorator
            // Copy the attributes of the <decorator> to the Decorator
            deco.type = xDecos[i].att("type");
            // Set the bool flip based on whether the text of the attribute is
            //   "1" or something else. This is an atypical but perfectly fine
            //   use of the == comparison operator. It will return a true or
            //   false, which will be assigned to deco.flip.
            deco.flip = ( xDecos[i].att ("flip") == "1" );
            // floats need to be parsed from the attribute strings
            deco.scale = float.Parse( xDecos[i].att ("scale") );
            // Vector3 loc initializes to [0,0,0], so we just need to modify it
            deco.loc.x = float.Parse( xDecos[i].att ("x") );
            deco.loc.y = float.Parse( xDecos[i].att ("y") );
            deco.loc.z = float.Parse( xDecos[i].att ("z") );
            // Add the temporary deco to the List decorators
            decorators.Add (deco);
```

```
        }

        // Read pip locations for each card number
        cardDefs = new List<CardDefinition>(); // Init the List of Cards
        // ...
```

- Configuramos *Deck* para que tenga acceso a los sprites a partir de los cuales construiremos las cartas. Añadiendo las variables que los almacenarán

```
public class Deck : MonoBehaviour {
    // Suits
    public Sprite               suitClub;
    public Sprite               suitDiamond;
    public Sprite               suitHeart;
    public Sprite               suitSpade;

    public Sprite[]             faceSprites;
    public Sprite[]             rankSprites;

    public Sprite               cardBack;
    public Sprite               cardBackGold;
    public Sprite               cardFront;
    public Sprite               cardFrontGold;

    // Prefabs
    public GameObject           prefabSprite;
    public GameObject           prefabCard;

    public bool _________________;
```

- y asignándoles los recursos en el inspector

- Las cartas se crearán a partir de dos prefabs de tipo Sprite (*2D Object | Sprite*), *PrefabSprite* y *PrefabCard*. *PrefabSprite* es simplemente un game object de tipo sprite, mientras que *PrefabCard* requiere algo más de configuración: es un sprite, al que le asignamos el sprite *Card_Front* en la variable *Sprite* de su componente *Sprite Renderer*, le añadimos el script *Card* y un componente *Box Collider*. Después de crearlos, se transforman en prefabs y se asignan a las variables correspondientes de *Deck*

- Estructura de datos de una carta

```
    public string           suit; // Suit of the Card (C,D,H, or S)
    public int              rank; // Rank of the Card (1-14)
    public Color            color = Color.black; // Color to tint pips
    public string           colS = "Black"; // or "Red". Name of the Color

    // This List holds all of the Decorator GameObjects
    public List<GameObject> decoGOs = new List<GameObject>();
    // This List holds all of the Pip GameObjects
```

```
public List<GameObject> pipGOs  = new List<GameObject>();

public GameObject       back; // The GameObject of the back of the card

public CardDefinition   def; // Parsed from DeckXML.xml
```

- A continuación debes escribir el código que genere las 52 cartas de la baraja y las muestre al ejecutarse el juego. Cada carta es una instancia de *PrefabCard* y cada decoración es un hijo suyo que se obtiene instanciando *PrefabSprite* y asignando el sprite adecuado al atributo *sprite* de su componente *SpriteRenderer*

Referencias

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

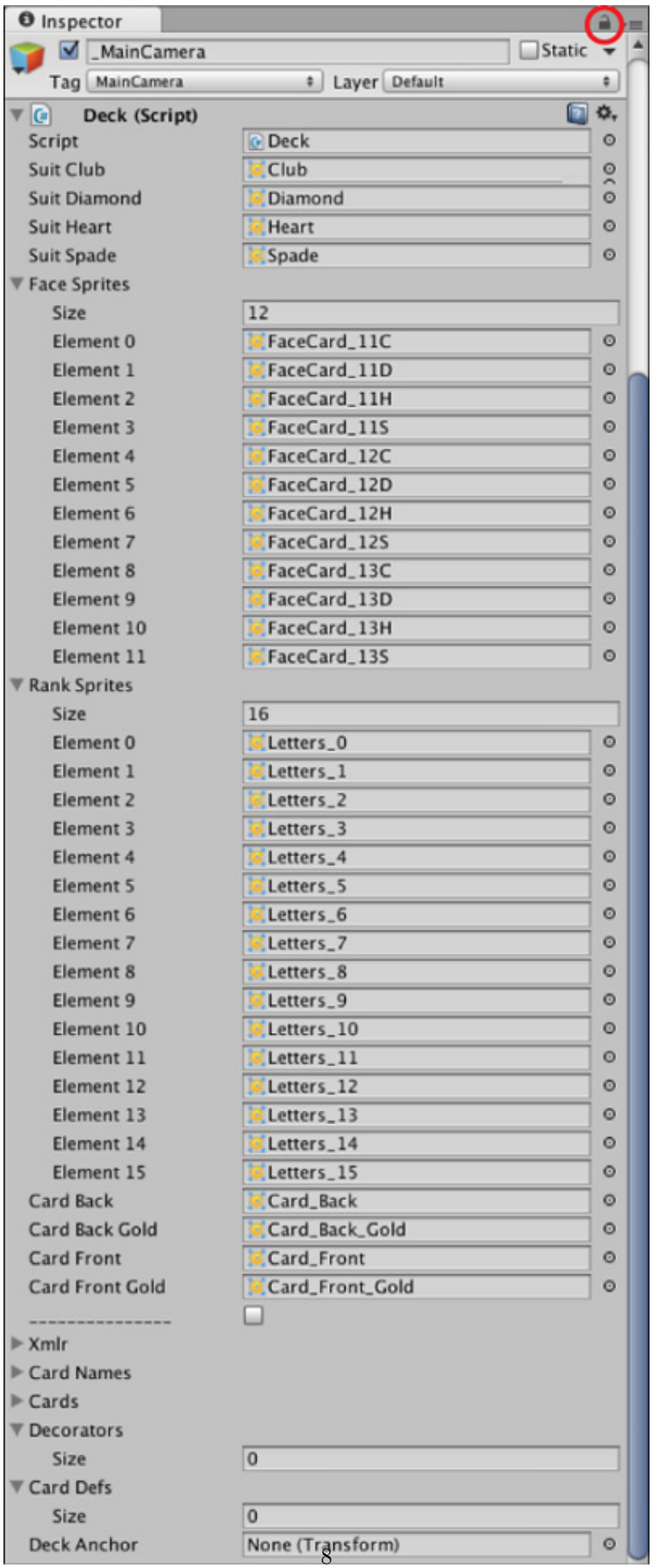

Figure 3: Sprites asignados a la clase Deck

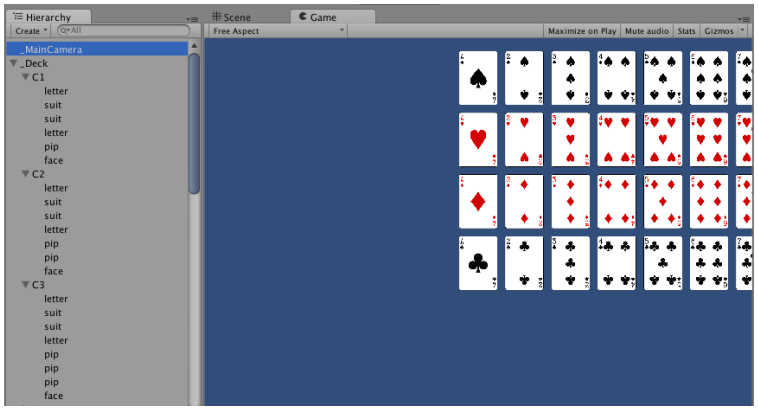

Figure 4: Generación de la baraja