# VIDEO POKER JACKS OR BETTER

Game Documentation and HowTo Guide

Unity Version: 2017.2.0f3

## Contents

## Package Description and Features

Video Poker : Jacks Or Better is a full Unity game project that can be used as a base to create other variations of popular Las Vegas video poker games or simply release your own Jacks Or Better app.

**Features:**

- Game ready for release straight out of the box, just build and play!
- Works on the following platforms: PC, Mac, iOS, Android, and WebGL.
- Supports multiple resolutions and aspect ratios, automatically.
- Supports Mouse and Touch controls.

**Current Version 1.0**

## Setting Up:

In order to quickly get your project set up, follow the steps below:

- Create a new Unity project (Current version of Unity as of this writing is 2017.2.0f3)
- Import the Custom package

## Getting Started

Video Poker : Jacks Or Better is a complete project, and as such is supposed to work as the starting point of your planned game, rather than an addition to an existing project. This project works best as a starter kit which you can customize to fit your project needs.

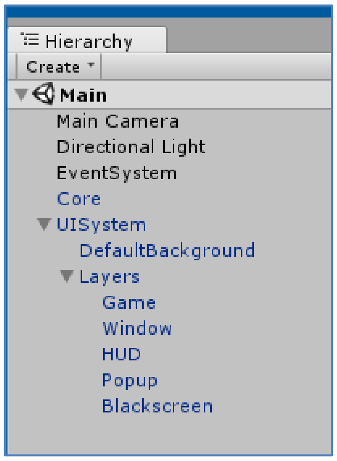
To play the game, simply open up your "Main.scene" file. Pressing the play button within the Unity editor should start the game.

## The Core

The "Core" game object in the project hierarchy is the main object that initializes the game such as the ScreenManager, PlayerPrefHelper, PokerGame, and a few others. After it is initialized, it calls the "LoadTitleScreen" function which, as the function name states, loads the title screen.

### UISystem

The Core script ("Core.cs") has a public GameObject variable ***uiSystem*** that needs to be set to the UISystem game object in the scene hierarchy. If this variable is not properly assigned, it will not initialize properly. The UISystem is a prefab located at ***Assets/Resources/UI/Prefabs*** and contains the different layers which the game will use to display the various game screens. You can refer to the following image to see what layers are currently defined:



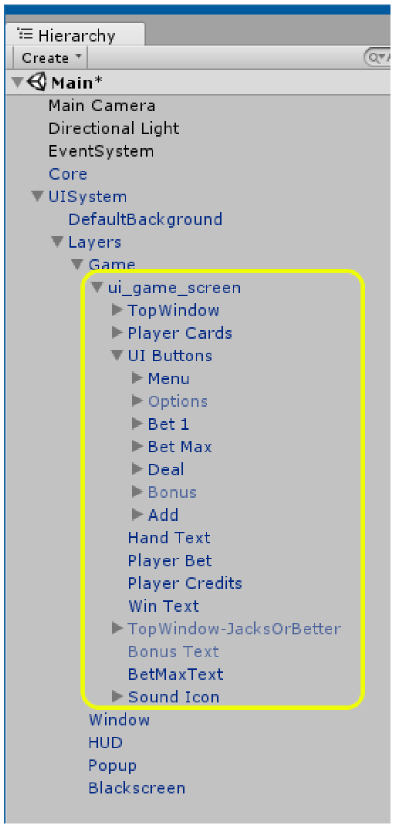
```
Hierarchy
Create
▼ Main
    Main Camera
    Directional Light
    EventSystem
    Core
  ▼ UISystem
      DefaultBackground
    ▼ Layers
        Game
        Window
        HUD
        Popup
        Blackscreen
```

This game object will be passed into the ScreenManager during initialization.

## ScreenManager

The ScreenManager ("ScreenManager.cs") is the class that manages the various game screens that are created through code and are added to the different layers of the UISystem.

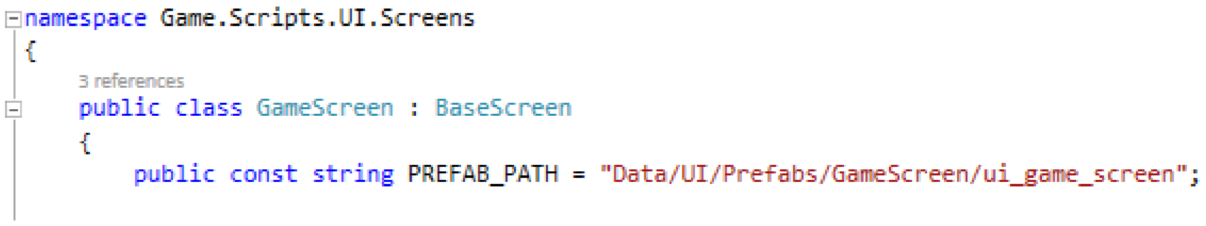
Each UI screen is a prefab located in the "Resources" folder **Assets/Resources/Data/UI/Prefabs**. If you would like to view or edit these prefabs, just drag and drop the prefab onto one of the layers of the UISystem. For example, I've placed the "ui_game_screen" prefab onto the "Game" layer:



```
Hierarchy
Create
▼ Main*
    Main Camera
    Directional Light
    EventSystem
    Core
  ▼ UISystem
      DefaultBackground
    ▼ Layers
      ▼ Game
        ▼ ui_game_screen
          ▶ TopWindow
          ▶ Player Cards
          ▼ UI Buttons
            ▶ Menu
            ▶ Options
            ▶ Bet 1
            ▶ Bet Max
            ▶ Deal
            ▶ Bonus
            ▶ Add
            Hand Text
            Player Bet
            Player Credits
            Win Text
          ▶ TopWindow-JacksOrBetter
            Bonus Text
            BetMaxText
          ▶ Sound Icon
        Window
        HUD
        Popup
        Blackscreen
```

When the UI prefab is added to one of the layers in the UISystem, you should be able to view and edit the screen within the editor.

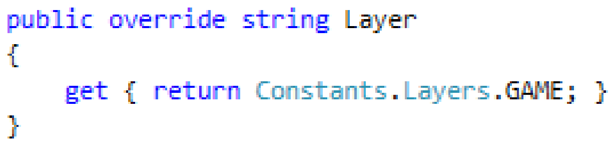
Each UI screen prefab will be associated with a corresponding screen class. Here is an example of how this is defined for the game screen:



```
namespace Game.Scripts.UI.Screens
{
    3 references
    public class GameScreen : BaseScreen
    {
        public const string PREFAB_PATH = "Data/UI/Prefabs/GameScreen/ui_game_screen";
```

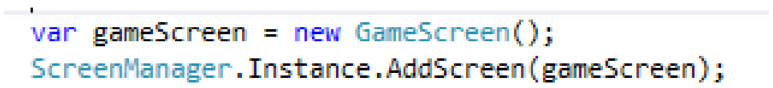
In this example, we've defined the "PREFAB_PATH" for the GameScreen class to the prefab for the game screen. This variable will be used to create the proper game object when the GameScreen class is instantiated.

As you can see, the GameScreen class is derived from the BaseScreen class. This is important because each screen should be derived from the BaseScreen class because the BaseScreen class contains properties and methods that will be required for each UI screen. One of those properties defines which layer the UI screen will be attached to within the UISystem game object. This is done automatically within the ScreenManager class. For example, I've defined the layer property for the game screen to the "Game" layer like so:



```
public override string Layer
{
    get { return Constants.Layers.GAME; }
}
```

Creating and adding the new screen in the game is very simple. For example, I've created a new GameScreen instance and add it to the game like this:



```
var gameScreen = new GameScreen();
ScreenManager.Instance.AddScreen(gameScreen);
```

## PlayerPrefHelper

The PlayerPrefHelper ("PlayerPrefHelper.cs") is used to store various game data like preferences and the amount of credits the player has. For example, we use it to save the player's credits inside of the GameScreen class:

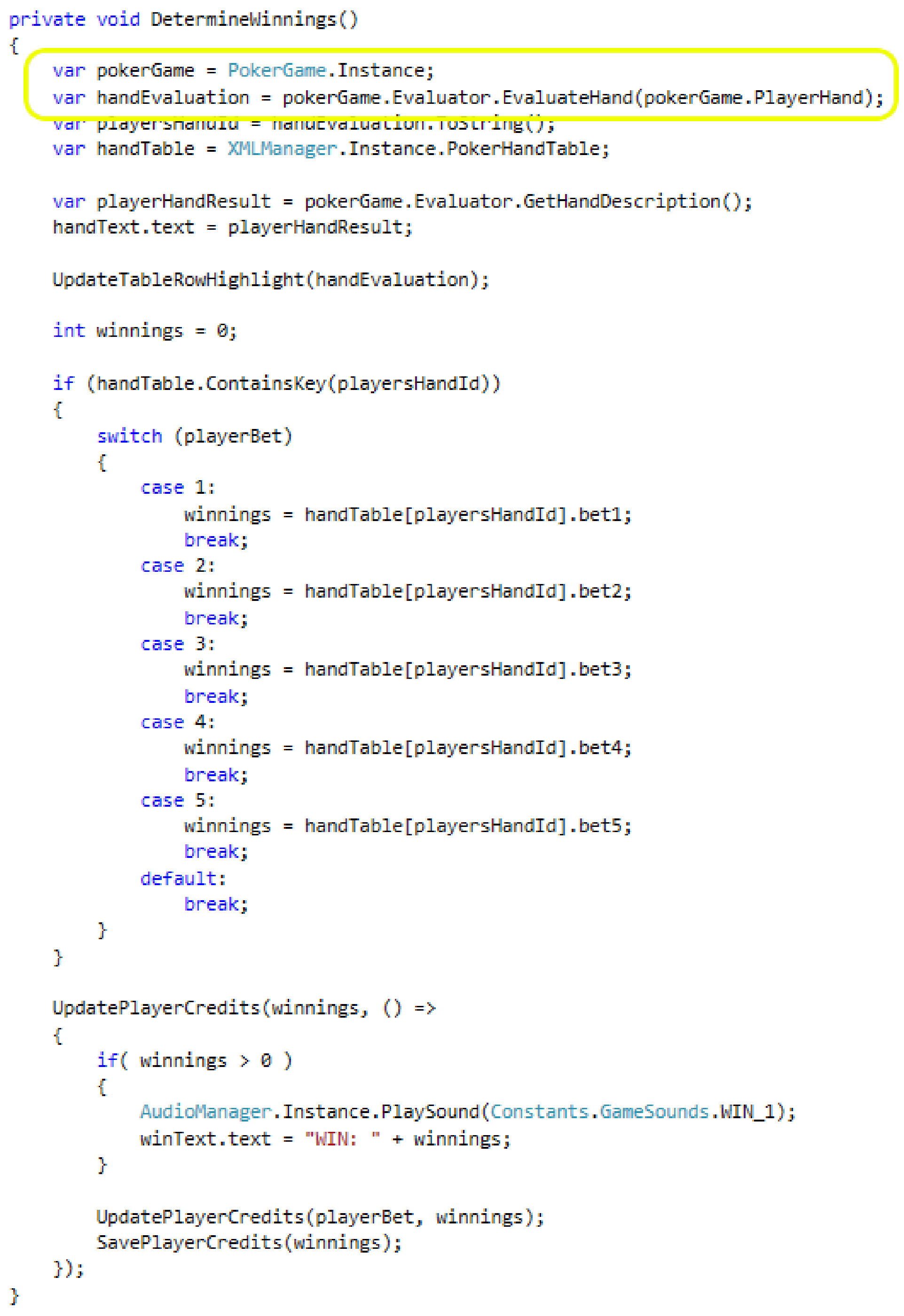
```
public void SavePlayerCredits(int addValue = 0)
{
    var playerPrefHelper = PlayerPrefHelper.Instance;

    if (playerPrefHelper == null)
    {
        Debug.LogError("GameScreen::SavePlayerCredits - Unable to get PlayerPrefHelper instance.  NULL returned.");
        return;
    }

    if (addValue > 0 )
    {
        playerPrefHelper.SetInt(Constants.Game.PLAYER_CREDITS, playerCredits + addValue);
    }
    else
    {
        playerPrefHelper.SetInt(Constants.Game.PLAYER_CREDITS, playerCredits);
    }
}
```

## PokerGame

The PokerGame class is a simple class that contains the array of cards the player holds, the deck of cards, and the HandEvaluator (“HandEvaluator.cs”).

The HandEvaluator class defines the type of card combinations and calculates is used to calculate what type of hand the player currently holds.

In the following example, I use the HandEvaluator to determine the player’s winnings in the GameScreen (“GameScreen.cs”):



```
private void DetermineWinnings()
{
    var pokerGame = PokerGame.Instance;
    var handEvaluation = pokerGame.Evaluator.EvaluateHand(pokerGame.PlayerHand);
    var playersHandId = handEvaluation.ToString();
    var handTable = XMLManager.Instance.PokerHandTable;

    var playerHandResult = pokerGame.Evaluator.GetHandDescription();
    handText.text = playerHandResult;

    UpdateTableRowHighlight(handEvaluation);

    int winnings = 0;

    if (handTable.ContainsKey(playersHandId))
    {
        switch (playerBet)
        {
            case 1:
                winnings = handTable[playersHandId].bet1;
                break;
            case 2:
                winnings = handTable[playersHandId].bet2;
                break;
            case 3:
                winnings = handTable[playersHandId].bet3;
                break;
            case 4:
                winnings = handTable[playersHandId].bet4;
                break;
            case 5:
                winnings = handTable[playersHandId].bet5;
                break;
            default:
                break;
        }
    }

    UpdatePlayerCredits(winnings, () =>
    {
        if( winnings > 0 )
        {
            AudioManager.Instance.PlaySound(Constants.GameSounds.WIN_1);
            winText.text = "WIN:  " + winnings;
        }

        UpdatePlayerCredits(playerBet, winnings);
        SavePlayerCredits(winnings);
    });
}
```

## Support

Website: http://www.kahunastudios.com

Email: support@kahunastudios.com