Creating a Role Playing Game with XNA Game Studio
Part 48
Picking Up Items - Part 2

To follow along with this tutorial you will have to have read the previous tutorials to understand much of what it going on. You can find a list of tutorials here: XNA Role Playing Game Tutorials You will also find the latest version of the project on the web site on that page. If you want to follow along and type in the code from this PDF as you go, you can find the previous project at this link: http://www.jtmbooks.com/rpgtutorials/New2DRPG47.zip You can download the graphics from this link: Graphics.zip

I this tutorial I will be continuing on with having the player being able to pick up items on the map. To do this I will be adding in a new game screen to display what the player has found and add what they have found to the character. You will need a background image for the pop up screen that will display what the player has found. You can download the image that I used from the following file: treasurebackground.zip. After you have downloaded and extracted the image, add it to the **GUI** folder in the **Content** folder. The name of the image is **treasurebackground.png**. If you use your own image, to avoid confusion, try and give it the same name.

The first thing I want to do is create the new screen. Right click the GameScreens folder and add a new class called TreasureScreen. This is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Content;
using New2DRPG.CoreComponents;
using New2DRPG.ItemClasses;

namespace New2DRPG
{
    class TreasureScreen : GameScreen
    {
        ButtonMenu menu;
        Texture2D image;
        Texture2D buttonImage;
        SpriteFont spriteFont;
        Vector2 imagePosition;
        Chest chest = null;

        public TreasureScreen(Game game)
            : base(game)
        {
            LoadContent();

            Components.Add(new BackgroundComponent(game, image, false));
```

```
        imagePosition = new Vector2();
        imagePosition.X = (game.Window.ClientBounds.Width - image.Width) / 2;
        imagePosition.Y = (game.Window.ClientBounds.Height - image.Height) / 2;

        string[] items = { "OK" };
        menu = new ButtonMenu(game, spriteFont, buttonImage);
        menu.SetMenuItems(items);
        Components.Add(menu);
    }

    public int SelectedIndex
    {
        get { return menu.SelectedIndex; }
    }

    public Chest Chest
    {
        get { return chest; }
    }

    protected override void LoadContent()
    {
        image = Content.Load<Texture2D>(@"GUI\treasurebackground");
        buttonImage = Content.Load<Texture2D>(@"GUI\buttonbackgroundshort");
        spriteFont = Content.Load<SpriteFont>(@"normal");
        base.LoadContent();
    }

    public override void Draw(GameTime gameTime)
    {
        base.Draw(gameTime);
        if (chest.Gold != 0)
        {
            Vector2 position = new Vector2(
                15 + imagePosition.X,
                130 + imagePosition.Y);
            string gold = "You found " + chest.Gold.ToString() + " gold!";
            spriteBatch.DrawString(spriteFont,
                gold,
                position,
                Color.Yellow);
        }
    }

    public override void Show()
    {
        chest = null;
        base.Show();
        menu.Position = new Vector2((image.Width -
                          menu.Width) / 2 + imagePosition.X,
                          image.Height - menu.Height - 10 + imagePosition.Y);
    }

    public void Show(Chest chest)
    {
        Show();
        this.chest = chest;
    }
```

```
        public override void Hide()
        {
            base.Hide();
            chest = null;
        }
    }
}
```

This class is like most of the other game screens that I created but there are a few differences. I actually copied and pasted the code from another screen to this class instead of starting from scratch. The first difference is there is a using statement for the **ItemClasses** of the game. I added that so I can pass **Chest** objects to the screen. I will use the **Chest** objects to display the gold the player has found, as well as any items, when I add the ability to pick up actual items. This class inherits from the **GameScreen** class so that it can be used in the screen management system. Because I will be passing **Chest** objects to the class there is a **Chest** field, **chest**.

The constructor is like most of the other pop up screens I created as well. The one difference is there is only a single menu item, OK, because this is just an information screen and there will be no decisions made by the screen. You just need to the player to be able to close the screen. There is a get only property that will be used to return the chest back to the game. The reason I added this is because the chests are stored in the game as a list. When I show the screen I pass the current chest to the screen and remove the chest form the list. When I close the screen, I get the chest and add anything the chest contains to the player's character. In the **LoadContent** method, I load in the new background for the screen.

In the **Draw** method is where I display what is contained in the chest. You need to draw it after the call to **base.Draw** or it will be drawn over. To determine if I will draw what is in the chest I check to make sure that the chest is not null and that there is actually gold in the chest. I do this by checking to see if the **Gold** property of the chest is not 0. To handle the case where a chest isn't available I set the **chest** to null initially. By putting it as the first part of the condition means that the second part of the condition will not be evaluated if **chest** is null. This way if you display a chest that is set to null your program won't generate an exception when you check to see if there is gold in the chest. You should get into the habit of doing things like this, to prevent exceptions from being generated whenever possible.

Inside that if statement I then create a **Vector2** to determine where to draw text. This **Vector2** is set to be the **X** position of the screen plus 15 pixels. I added the 15 pixels to give the text a little padding on the left hand side to make it look a little better on the screen. For the **Y** position of the text I take the **Y** position of the image screen and add 130 pixels. The reason I added this is because of the image that I used to display the contents of the chest. It is another padding value that positions the text nicely with the image I created. If you used your own image, you will want to set these padding values yourself to have things look nice in the pop up screen. I then make a **string** that holds the message: "You found " + **chest.Gold.ToString()** + " gold!" This string will display the amount of the gold the player has found. I then draw the string in yellow. This just made the string stand out more, in my opinion.

There are two **Show** methods in this class. The first one is an override of the **Show** method of the **GameScreen** class. The other takes a **Chest** object as its parameter. In the first **Show** method I set **chest** to null. This way if the default **Show** method is called the **Draw** method will not draw anything if there was previously a **chest** object. It then calls **base.Show** to call the **Show** method of the parent

class, **GameScreen**. Then like in other screens that have menus I set the position of the menu. In the **Show** method that takes a **Chest** object I call the override of the **Show** method that takes no parameters to do everything above and then set the **chest** field to the **chest** parameter.

The override of the **Hide** method calls **base.Hide** to actually hide the screen and then sets **chest** to null. This just makes sure that old chests won't be displayed.

I made a few small changes to the **Chest** class. Since it isn't a long class I will give you the new code and then go over the changes. This is the code for the new **Chest** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using New2DRPG.SpriteClasses;

namespace New2DRPG.ItemClasses
{
    class Chest
    {
        static Random random = new Random();
        ItemSprite sprite;
        static float collisionRadius = 64;
        int goldMinimum;
        int goldMaximum;
        int gold;

        public Chest(Game game, Texture2D texture, Vector2 position)
        {
            sprite = new ItemSprite(game, texture, position);
            gold = 0;
        }

        public Chest(Game game,
                Texture2D texture,
                Vector2 position,
                int goldMinimum,
                int goldMaximum)
        {
            sprite = new ItemSprite(game, texture, position);
            this.goldMinimum = goldMinimum;
            this.goldMaximum = goldMaximum;
            if (goldMinimum == goldMaximum)
                gold = goldMinimum;
            else
                gold = Chest.random.Next(goldMinimum, goldMaximum + 1);
        }

        public static float CollisionRadius
        {
            get { return collisionRadius; }
        }

        public Vector2 Position
        {
```

```
            get { return sprite.Position; }
        }

        public Vector2 Origin
        {
            get { return sprite.Origin; }
        }

        public int Gold
        {
            get { return gold; }
        }

        public void Update(GameTime gameTime)
        {
            sprite.Update(gameTime);
        }

        public void Draw(GameTime gameTime)
        {
            sprite.Draw(gameTime);
        }
    }
}
```

The first change was I change the **collisionRadius** field to 64. This way the player will pick up a chest when the player sprite's origin is with in 64 pixels of the chest's origin. The next change is I added in a new field, **gold**, that will hold the gold that is in the chest. I added in a new constructor for the class that takes as parameters the minimum gold the chest can hold and the maximum gold the chest can hold. In the original constructor I set the **gold** field to 0. In the new constructor I set the **goldMinimum** and **goldMaximum** fields. I then check to see if the **goldMinimum** and **goldMaximum** values are the same. If they are I set the **gold** field to **goldMinimum**. Otherwise I use the **Random** object to generate a random number between **goldMinimum** and **goldMaximum**. The overload of the **Next** method that I used will generate a number between the first parameter and the second parameter that excludes the second parameter. That is why I added 1 to **goldMaximum**. The **Gold** property now just returns the **gold** field.

Before I get to the changes made to the **Game1** class, I want to make a quick change to the **PlayerCharacter** class. I needed a way to add the gold in the chests to the player's character. To do this I added in a new method called **AddGold**. The method takes an integer parameter, the gold to be added to the players character's **gold** field. Since I'm passing in an integer, I needed to cast the value to a ulong to add it to the **gold** field. I added this method just below the **Gold** property. The reason I decided to go with the **AddGold** method is that it is the best way to eventually be able to preform validation on values passed as parameters. This is the new method.

```
public void AddGold(int gold)
{
    this.gold += (ulong)gold;
}
```

The rest of the changes were all in the **Game1** class. The first thing to do is to add in a field for the **TreasureScreen** class. Add this field in with the rest of the screen fields.

```
TreasureScreen treasureScreen;
```

Now you need to create an instance of the **TreasureScreen** class. What I did is in the **LoadGameScreens** method was call a method I wrote called **LoadTreasureScreen**. This method just creates a new instance of the **TreasureScreen** class like the other **Load** methods. This is the code for the **LoadGameScreens** method and the **LoadTreasureScreen** method.

```
private void LoadGameScreens()
{
    LoadCreatePCScreen();
    LoadGenderPopUpScreen();
    LoadClassPopUpScreen();
    LoadDifficultyPopUpScreen();
    LoadNameInputScreen();

    LoadHelpScreen();
    LoadCreditScreen();
    LoadIntroScreen();

    LoadActionScreen();
    LoadViewCharacterScreen();
    LoadCombatScreen();

    LoadStartScreen();
    LoadTreasureScreen();
    LoadQuitPopUpScreen();
    LoadQuitActionScreen();

    creditScreen.Hide();
    startScreen.Hide();
    helpScreen.Hide();
    createPCScreen.Hide();

    activeScreen = introScreen;
    activeScreen.Show();
}

private void LoadTreasureScreen()
{
    treasureScreen = new TreasureScreen(this);
    Components.Add(treasureScreen);
    treasureScreen.Hide();
}
```

What I did next was modify the **CreateChests** method that randomly added chests to the game. I just used the new constructor to pass in some random values for the **goldMinimum** and **goldMaximum** parameters. This is the new code for that method.

```
private void CreateChests()
{
    for (int i = 0; i < 2; i++)
    {
        Chest tempChest = new Chest(
            this,
            Content.Load<Texture2D>(@"Items\chest"),
            new Vector2(random.Next(3, 3 + 5), random.Next(3, 3 + 5)),
            random.Next(100),
```

```
                random.Next(100, 200));
            chests.Add(tempChest);
        }
    }
```

The next thing to change is the **CheckPickupRadius** method that checks to see if there is a collision with a chest. There is a big difference from the previous method so I will explain the code after you have read it. This is the new **CheckPickupRadius** method.

```
private bool CheckPickupRadius(GameTime gameTime)
{
    for (int i = 0; i < chests.Count; i++)
    {
        chests[i].Update(gameTime);

        float distance = Vector2.Distance(
            chests[i].Origin,
            playerSprite.Origin);

        if (distance < Chest.CollisionRadius)
        {
            activeScreen.Enabled = false;
            activeScreen = treasureScreen;
            treasureScreen.Show(chests[i]);
            chests.RemoveAt(i);
            return true;
        }
    }

    return false;
}
```

The method loops through all of the chests in a for loop, not a foreach loop. The reason for this is when you use a foreach loop, you can not modify what you are looping through. I don't mean the individual objects, I mean the collection or array that you are looping through. When I find a collision with a chest I remove that chest from the list of chests. I can't do that in a foreach loop. This is the best way to deal with removing chests from the game.

Inside the loop I first call the **Update** method of the chest. The reason I'm doing this is eventually you could use an animated sprite for the chests and the chest could sparkle for example. Then like with other check radius methods I calculate the distance between the origin of the player's sprite and the chest. If the distance is less than the collision radius of the chest I set the **Enabled** property of **activeScreen** to false so the **activeScreen** will still draw but not update. I then set **activeScreen** to be **treasureScreen**. Unlike when I transfer between other screens, I do not call the **Show** method of **activeScreen**. I call the **Show** method of **treasureScreen** because I want to pass in the current chest as a parameter to the **Show** method. The **List<T>** class has a method **RemoveAt** that takes as an integer parameter that is the index of the element you want to remove from the **List**. So I remove the current element passing in its index and I return true. If there is no collision I just return false.

There is one thing left to do. That is to handle input if **activeScreen** is **treasureScreen**. I created a new method, **HandleTreasureScreenInput**, that I will call from the **Update** method if **activeScreen** is **treasureScreen**. I will give you the new code of the **Update** method and the code for the new

HandleTreasureScreenInput and then explain the new method.

```
protected override void Update(GameTime gameTime)
{
    newState = Keyboard.GetState();
    newPadState = GamePad.GetState(PlayerIndex.One);

    if (inDialog && dialogNPC != null)
    {
        dialog.Show();
        dialogNPC.StartDialog(dialogNPC.DialogName);
        dialogNPC = null;
    }

    if (!dialog.Enabled)
    {
        inDialog = false;
        dialogNPC = null;
    }

    if (activeScreen == startScreen)
    {
        HandleStartScreenInput();
    }
    else if (activeScreen == helpScreen)
    {
        HandleHelpScreenInput();
    }
    else if (activeScreen == createPCScreen)
    {
        HandleCreatePCScreenInput();
    }
    else if (activeScreen == quitPopUpScreen)
    {
        HandleQuitPopUpScreenInput();
    }
    else if (activeScreen == genderPopUpScreen)
    {
        HandleGenderPopUpScreenInput();
    }
    else if (activeScreen == classPopUpScreen)
    {
        HandleClassPopUpScreenInput();
    }
    else if (activeScreen == difficultyPopUpScreen)
    {
        HandleDifficultyPopUpScreenInput();
    }
    else if (activeScreen == nameInputScreen)
    {
        HandleNameInputScreenInput();
    }
    else if (activeScreen == introScreen)
    {
        HandleIntroScreenInput();
    }
    else if (activeScreen == creditScreen)
    {
```

```
            HandleCreditScreenInput();
        }
        else if (activeScreen == actionScreen)
        {
            HandleActionScreeenInput();
            HandlePlayerInput(gameTime);
        }
        else if (activeScreen == viewCharacterScreen)
        {
            HandleViewCharacterScreenInput();
        }
        else if (activeScreen == quitActionScreen)
        {
            HandleQuitActionScreenInput();
        }
        else if (activeScreen == combatScreen)
        {
            HandleCombatScreenInput();
        }
        else if (activeScreen == treasureScreen)
        {
            HandleTreasureScreenInput();
        }
        base.Update(gameTime);

        oldState = newState;
        oldPadState = newPadState;
    }

    private void HandleTreasureScreenInput()
    {
        if (CheckKey(Keys.Enter) || CheckButton(Buttons.B))
        {
            Chest chest = treasureScreen.Chest;
            activeScreen.Hide();
            activeScreen = actionScreen;
            activeScreen.Show();
            playerCharacter.AddGold(chest.Gold);
        }
    }
```

The new method checks to see if the Enter key or the B button on the game pad have been pressed. If they have I get the **Chest** object that was passed to **treasureScreen**. I then call the **Hide** method of **activeScreen** to hide that screen. Set **activeScreen** to be **actionScreen** and call the **Show** method to show **activeScreen**. I now call the **AddGold** method of **playerCharacter** to add the gold in the chest to the player's gold.

In a future tutorial I will cover being able to pick up items in a chest. Before I can do that I will need to add in a few new classes to handle items. You will eventually want to add in the ability to create chests in an editor and add them to the game at compile time.

Well, that is it for this tutorial. I will be adding in more and more functionality to the game that you would expect to find in a role playing game. In the next tutorial I will continue on with picking up items. I encourage you to keep either visiting my site http://xna.jtmbooks.com or my blog, http://xna-rpg.blogspot.com for the latest news on my tutorials.