Creating a Role Playing Game with XNA Game Studio 3.0
Part 36
Collision Layer - Part 1

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 3.0 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/New2DRPG35.zip You can download the graphics from this link: Graphics.zip

In today's tutorial I will get started with adding the collision layer to the map. At the moment the player can wander around the map but they can't really interact with the map. What I mean is in the map I created there are water tiles but the player can walk right through them. To keep the player from doing this I'm going to add a collision layer for the tile map. At the moment the collision layer will be a little simple but as the game progresses I will be updating the collision layer.

To get started the first thing that you will want to do is make sure that the game is the active project. What you will want to do is right click the game project and select **Set As Start Up Project**. Since the collision layer is related to the tile engine I will add it to the **TileEngine** folder. Right click the **TileEngine** folder and select the **Add Class** option and name the new class **CollisionLayer**. This is the code for the **CollisionLayer** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace New2DRPG
{
    public enum CollisionType { Passable, Unpassable };

    class CollisionLayer
    {
        CollisionType[,] map;

        public CollisionLayer(int width, int height)
        {
            map = new CollisionType[height, width];

            for (int y = 0; y < height; y++)
                for (int x = 0; x < width; x++)
                    SetTile(x, y, CollisionType.Passable);
        }

        public CollisionType GetTile(int x, int y)
        {
            return map[y, x];
        }

        public void SetTile(int x, int y, CollisionType value)
        {
            map[y, x] = value;
        }
    }
}
```

The first thing you will have to do is make sure you change the namespace from **New2DRPG.TileEgnine** to just **New2DRPG**. There is an enum in this class called **CollisionType**. This enum will be used to tell how the collision will work with the collision layer. At the moment it is rather simple there are only two items: **Passable** and **Unpassable**. The **Passable** one will be applied to tiles that the player can walk through and **Unpassable** will mean that the player can not walk through the tiles.

Just like in the **TileMapLayer** class there is a two dimensional array to hold the values for the layer called **map**. Instead of being of type int they are of type **CollisionType**. I find that using an enum instead of numbers makes it easier to remember what the values do when you return to the code later on. There will be a slight problem a little later on when I'm making the collision layer in the editor and creating a content pipeline project for the collision layer. We will cross that bridge when we get to it though.

There is just one constructor and two methods in this class. The methods are **GetTile** and **SetTile** as you can guess the first one gets the value of the tile and the other one sets the value of a tile. The constructor takes as parameter the width and the height of the map. It creates a new two dimensional array with the height first and the width second, just like with the layers of the map. The next step in the constructor wasn't entirely necessary because by default the array will have as a default value **CollisionType.Passable** but I did it just to make sure. The **GetTile** method has two parameters, the coordinates of the tile. It returns the tile at the coordinates. The **SetTile** method has three parameters, the coordinates of the tile and the value to set the tile too.

Now that we have the class it needs to be implemented into the game. This will be done in the **TileMap** class. The first thing you will want to do is add a field for the collision layer. Just below the **tileMapLayers** field add the following field.

```
CollisionLayer collisionLayer;
```

The next thing you have to do is actually create the collision layer and process the collision layer.. You will of course create the collision layer in the constructor of the **TileMap** class. To process the collision layer I will create a new method called **ProcessCollisionLayer**. I will also be adding in a method to the **TileMap** class to get what the value for the collision layer is called **GetCollisionValue**.

There is still the two constructors in the **TileMap** class. I will add the collision layer to both of them. What I did in both constructors was after either creating a random map or loading in a map was create the **collisionLayer** field and then in a foreach loop go through all of the layers in the map and call the **ProcessCollisionLayer** method passing in as an argument the current layer. These are the new constructors for the **TileMap** class.

```
public TileMap(int tileMapWidth, int tileMapHeight, Tileset tileset, Game game)
    : base(game)
{
    spriteBatch = Game1.TileSpriteBatch;
    Content =
        (ContentManager)Game.Services.GetService(typeof(ContentManager));
    this.tileset = tileset;

    LoadContent();

    mapWidth = tileMapWidth;
    mapHeight = tileMapHeight;

    TileMapLayer layer = new TileMapLayer(mapWidth, mapHeight);

    for (int x = 0; x < tileMapWidth; x++)
        for (int y = 0; y < tileMapHeight; y++)
            layer.SetTile(x, y, 0);

    tileMapLayers.Add(layer);

    layer = new TileMapLayer(mapWidth, mapHeight);

    for (int x = 0; x < tileMapWidth; x++)
        for (int y = 0; y < tileMapHeight; y++)
        {
            layer.SetTile(x, y, -1);
            if (random.Next(0, 50) < 5)
            {
                layer.SetTile(x, y, tileset.Tiles.Count - 1);
            }
        }

    tileMapLayers.Add(layer);

    collisionLayer = new CollisionLayer(mapWidth, mapHeight);
    foreach (TileMapLayer mapLayer in tileMapLayers)
    {
        ProcessCollisionLayer(mapLayer);
    }

    for (int i = 0; i < 2; i++)
    {
        ItemSprite item =
            new ItemSprite(game,
                itemTexture,
                new Vector2(random.Next(0, 8), random.Next(0, 8)));
        items.Add(item);
    }
}

public TileMap(string mapName, Tileset tileset, Game game)
    : base(game)
{
    spriteBatch = Game1.TileSpriteBatch;
    Content =
        (ContentManager)Game.Services.GetService(typeof(ContentManager));

    this.tileset = tileset;
    LoadContent();
    LoadMap(mapName);

    collisionLayer = new CollisionLayer(mapWidth, mapHeight);
    foreach (TileMapLayer layer in tileMapLayers)
    {
        ProcessCollisionLayer(layer);
    }
}
```

As you can see the **ProcessCollisonLayer** method has one parameter, a **TileMapLayer** object. The other method, **GetCollisionValue** takes two parameters the x and y value for the tile and returns a **CollisionType**. This is the code for the two methods. I will explain the code after you have seen it.

```
private void ProcessCollisionLayer(TileMapLayer layer)
{
    for (int y = 0; y < mapHeight; y++)
        for (int x = 0; x < mapWidth; x++)
        {
            switch (layer.GetTile(x, y))
            {
                case 32:
                case 33:
                case 34:
                case 40:
                case 41:
                case 42:
                case 48:
                case 49:
                case 50:
                case 63:
                    collisionLayer.SetTile(x, y, CollisionType.Unpassable);
                    break;
            }
        }
}

public CollisionType GetCollisionValue(int x, int y)
{
    return collisionLayer.GetTile(x, y);
}
```

All the **ProcessCollisionLayer** method does is loop through all of the tiles in the layer using nested for loops. Then there is a switch statement. This might look a little strange to some of you. Why did I have no code after each of the cases? Well, if you don't have any code after a case when the switch statement executes if it matches a case the code will fall through to the next case and so on. This way if you have a lot of values that have the same result you can just list them one after the other and when they are all listed you have your code. If you try to put code between two of them with out a break you will get an error. Also, if you put code with a break between the cases you will not fall through from that case to the others. At the moment I just handled 10 tiles. Tiles 32, 33, 34, 40, 41, 42, 48, 49, and 50 are all water tiles. Tile 63 is a rock tile. I could have added in more for the trees but I thought this was enough to get started with. If one of these cases is true the value for the collision layer is set to **CollisiontType.Unpassable**, which means the tile can not be entered. All that the **GetCollisionValue** method does is call the **GetTile** method of the **CollisionLayer** class passing in the same arguments.

That is it for this tutorial. I know that it was rather short but I didn't want to throw handling the collision between the player's sprite and the collision layer at the same time because that is rather a complex topic and will take a bit of time to explain. In the next tutorial I will be working on the game again. 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.