Creating a Role Playing Game with XNA Game Studio 3.0
Part 37
Collision Layer - 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 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/New2DRPG36.zip You can download the graphics from this link: Graphics.zip

In today's tutorial I will be continuing on with the collision layer and adding in working with how to preform the collision detection. The first thing I want to do is add a public static method to the **TileEngine** class. This method will take a **Vector2** and return a **Point** that will hold which tile in the map the **Vector2** is in. It is just a copy of the method from the **Camera** class. Add the following method to the **TileEngine** class.

```
public static Point VectorToCell(Vector2 position)
{
    return new Point(
        (int)position.X / tileWidth,
        (int)position.Y / tileHeight);
}
```

The movement of the player takes place in the **HandleActionScreenInput** method. This is where I will check for the collision between the sprite and the tiles. The actual collision detection will take place in the **ActionScreen** class as that is where the map exists. This is the new code for the **HandleActionScreenInput** method. I will explain it after you have read it.

```
private void HandleActionScreeenInput()
{
    if (CheckKey(Keys.Escape))
    {
        playerSprite.IsAnimating = false;
        activeScreen.Enabled = false;
        activeScreen = quitActionScreen;
        activeScreen.Show();
    }
    else if (CheckKey(Keys.V))
    {
        playerSprite.IsAnimating = false;
        activeScreen.Enabled = false;
        activeScreen = viewCharacterScreen;
        viewCharacterScreen.SetPlayerCharacter(playerCharacter);
        activeScreen.Show();
    }
    else
    {
        Vector2 motion = new Vector2();
        playerSprite.IsAnimating = true;
        if (newState.IsKeyDown(Keys.Up) || newState.IsKeyDown(Keys.NumPad8))
        {
```

```
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Up;
    }
    if (newState.IsKeyDown(Keys.Down) || newState.IsKeyDown(Keys.NumPad2))
    {
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Down;
    }
    if (newState.IsKeyDown(Keys.Right) || newState.IsKeyDown(Keys.NumPad6))
    {
        motion.X++;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.Left) || newState.IsKeyDown(Keys.NumPad4))
    {
        motion.X--;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (newState.IsKeyDown(Keys.NumPad9))
    {
        motion.X++;
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.NumPad3))
    {
        motion.X++;
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Right;
    }
    if (newState.IsKeyDown(Keys.NumPad1))
    {
        motion.X--;
        motion.Y++;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (newState.IsKeyDown(Keys.NumPad7))
    {
        motion.X--;
        motion.Y--;
        playerSprite.CurrentAnimation = AnimationKey.Left;
    }
    if (motion != Vector2.Zero)
    {
        motion.Normalize();
        motion *= playerSprite.Speed;

        if (!actionScreen.CheckUnWalkableTile(playerSprite, motion))
            playerSprite.Position += motion;

    }
    else
    {
        playerSprite.IsAnimating = false;
    }


    playerSprite.LockToMap();
```

```
            camera.LockToSprite(playerSprite);
            camera.LockCamera();
        }
    }
```

Where the change was is in the if statement where I check to see if the **motion** vector is not **Vector2.Zero**. The reason is there is no need to check for a collision if the sprite is not moving. After normalizing the vector I then multiply **motion** by the speed of the sprite. I then call a method that I created in the **ActonScreen** class called **CheckForUnWalkableTile** that checks to see if the tile can be walked into passing in the sprite for the character and the motion vector. If the method returns false, meaning the tile can be entered I add the **motion** vector to the sprite's position.

Collision detection can be a very complex topic, especially when dealing with transparency and animated sprites. Collision detection with tiles can be just as complicated. I found a good method for handling the collision between tiles and the player character's sprite. You could eventually add it in so that the transparent parts of the sprites can enter into the transparent parts of tiles you don't want the sprite to walk into. I may eventually add the functionality into the game but that is a very complex topic and at the moment I will be focusing on just making it so that the player is unable to enter at all tiles that can not be entered.

The **CheckForUnWalkableTile** method calls one of eight methods in the **TileMap** class. These method check for collision in the various directions the sprite can move: up and to the left, straight up, up and to the right, straight left, straight right, down and to the left, straight down and down and to the right. This makes the code easier to read and keeps the methods short which is something I like to work toward. The methods are called **CheckUpAndLeft, CheckUp, CheckUpAndRight, CheckLeft, CheckRight, CheckDownAndLeft, CheckDown** and **CheckDownAndRight**. I will start with the code for the **CheckForUnWalkableTile** method and explain the way it works.

```
public bool CheckUnWalkableTile(AnimatedSprite sprite, Vector2 motion)
{
    Vector2 nextLocation = sprite.Position + motion;

    Rectangle nextRectangle = new Rectangle(
        (int)nextLocation.X,
        (int)nextLocation.Y,
        sprite.Width,
        sprite.Height);

    if (motion.Y < 0 && motion.X < 0)
    {
        return tileMap.CheckUpAndLeft(nextRectangle);
    }
    else if (motion.Y < 0 && motion.X == 0)
    {
        return tileMap.CheckUp(nextRectangle);
    }
    else if (motion.Y < 0 && motion.X > 0)
    {
        return tileMap.CheckUpAndRight(nextRectangle);
    }
    else if (motion.Y == 0 && motion.X < 0)
    {
        return tileMap.CheckLeft(nextRectangle);
    }
```

```
    else if (motion.Y == 0 && motion.X > 0)
    {
        return tileMap.CheckRight(nextRectangle);
    }
    else if (motion.Y > 0 && motion.X < 0)
    {
        return tileMap.CheckDownAndLeft(nextRectangle);
    }
    else if (motion.Y > 0 && motion.X == 0)
    {
        return tileMap.CheckDown(nextRectangle);
    }
    return tileMap.CheckDownAndRight(nextRectangle);
}
```

The first thing the method does is create a **Vector2** called **nextLocation** which will hold as a vector the next location the player's sprite will be in measured in pixels. I calculate it by taking the sprite's position and adding the **motion** vector to it. I then, using this vector, create a **Rectangle** called **nextRectangle** that I will pass to the helper methods that uses **nextLocation** and the width and height of the sprite. The **X** and **Y** properties of this rectangle will be the location of the sprite in pixels on the map. When it is converted to tiles it will be which tile the top left hand corner of the sprite is in. By adding the width and height of the sprite to the **X** and **Y** properties of the rectangle you can find out which tile the bottom right corner of the sprite is in. If the sprite is larger than the tile, for example if the sprite was a dragon and is bigger than 1 tile, you will need to check for collision with more than just one tile.

Next there is a set of if statements that check to see what direction the sprite is traveling in. The first one checks to see if the **X** and **Y** properties of **motion** are both less than zero. That means the sprite is moving up and left and the method calls the **CheckUpAndLeft** method passing in the result and returns that result. The second one checks to see if if the **Y** property is less than zero and the **X** property is 0 which means the sprite is moving just up and calls the appropriate method and returns the result. If **Y** is negative and **X** is positive the sprite is moving up and right and the **CheckUpAndRight** method is called. Then I check to see if **Y** is 0 and **X** is negative and call **CheckLeft** followed by **Y** = 0 and **X** is positive which results in calling **CheckRight**. The next two else ifs check for **Y** positive and **X** negative which means down and right so **CheckDownAndRight** is called and if **Y** is positive and **X** is 0 which means straight down and **CheckDown** is called. There is no need for another else if because all of the other possible options have been taken care of and that leaves the sprite moving down and right.

I guess the best way to deal with the helper methods is just show you the code for them all and then explain them after you have read the code. This is the code for the helper methods for the **TileMap** class place them near the **ProcessCollisionLayer** method as they are somewhat related to each other: **CheckUpAndLeft, CheckUp, CheckUpAndRight, CheckLeft, CheckRight, CheckDownAndLeft, CheckDown** and **CheckDownAndRight**.

```
public bool CheckUpAndLeft(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width,
                    nextRectangle.Y + nextRectangle.Height));
```

```
    bool doesCollide = false;

    if (tile1.X < 0 || tile1.Y < 0)
        return !doesCollide;


    for (int y = tile1.Y; y <= tile2.Y; y++)
        for (int x = tile1.X; x <= tile2.X; x++)
            if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                doesCollide = true;

    return doesCollide;
}

public bool CheckUp(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width - 1,
                    nextRectangle.Y + nextRectangle.Height));

    bool doesCollide = false;

    if (tile1.Y < 0)
        return !doesCollide;

    int y = tile1.Y;
    for (int x = tile1.X; x <= tile2.X; x++)
        if (GetCollisionValue(x, y) == CollisionType.Unpassable)
            doesCollide = true;

    return doesCollide;
}

public bool CheckUpAndRight(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width + 1,
                    nextRectangle.Y + nextRectangle.Height));

    bool doesCollide = false;

    if (tile2.X >= mapWidth || tile1.Y < 0)
        return !doesCollide;

    for (int y = tile1.Y; y <= tile2.Y; y++)
        for (int x = tile1.X; x <= tile2.X; x++)
            if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                doesCollide = true;

    return doesCollide;
}

public bool CheckLeft(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
```

```
            new Vector2(nextRectangle.X, nextRectangle.Y));
        Point tile2 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X + nextRectangle.Width,
                        nextRectangle.Y + nextRectangle.Height - 1));

        bool doesCollide = false;

        if (tile1.X < 0)
            return !doesCollide;

        int x = tile1.X;
        for (int y = tile1.Y; y <= tile2.Y; y++)
        {
            if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                doesCollide = true;
        }

        return doesCollide;
    }

    public bool CheckRight(Rectangle nextRectangle)
    {
        Point tile1 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X, nextRectangle.Y));
        Point tile2 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X + nextRectangle.Width,
                        nextRectangle.Y + nextRectangle.Height - 1));

        bool doesCollide = false;
        if (tile2.X >= mapWidth)
            !return doesCollide;

        int x = tile2.X;

        for (int y = tile1.Y; y <= tile2.Y; y++)
        {
            if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                doesCollide = true;
        }

        return doesCollide;
    }

    public bool CheckDownAndLeft(Rectangle nextRectangle)
    {
        Point tile1 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X, nextRectangle.Y));
        Point tile2 = TileEngine.VectorToCell(
            new Vector2(nextRectangle.X + nextRectangle.Width,
                        nextRectangle.Y + nextRectangle.Height));

        bool doesCollide = false;

        if (tile1.X < 0 || tile2.Y >= mapHeight)
            return !doesCollide;

        for (int y = tile1.Y; y <= tile2.Y; y++)
            for (int x = tile1.X; x <= tile2.X; x++)
                if (GetCollisionValue(x, y) == CollisionType.Unpassable)
```

```
                doesCollide = true;

    return doesCollide;
}

public bool CheckDown(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width - 1,
                    nextRectangle.Y + nextRectangle.Height));

    bool doesCollide = false;

    if (tile2.Y >= mapHeight)
        return !doesCollide;

    int y = tile2.Y;
    for (int x = tile1.X; x <= tile2.X; x++)
        if (GetCollisionValue(x, y) == CollisionType.Unpassable)
            doesCollide = true;

    return doesCollide;
}

public bool CheckDownAndRight(Rectangle nextRectangle)
{
    Point tile1 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X, nextRectangle.Y));
    Point tile2 = TileEngine.VectorToCell(
        new Vector2(nextRectangle.X + nextRectangle.Width,
                    nextRectangle.Y + nextRectangle.Height));

    bool doesCollide = false;

    if (tile2.X >= mapWidth || tile2.Y >= mapHeight)
        return !doesCollide;

    for (int y = tile1.Y; y <= tile2.Y; y++)
        for (int x = tile1.X; x <= tile2.X; x++)
            if (GetCollisionValue(x, y) == CollisionType.Unpassable)
                doesCollide = true;

    return doesCollide;
}
```

As you can see each of the methods are similar in structure and they work in the same way but each is implemented slightly differently. What the concept is that you get the tiles that the sprite occupies. The first tile the sprite will be in is found using the **X** and **Y** properties of the rectangle passed. You then use the **VectorToCell** method in the **TileEngine** class to convert the vector to a point which holds the tile.

To find the next tile for **X** value you take the **X** property of the rectangle and add the width of the rectangle. Since the sprite is the same width as a tile there will be times when the **X** property of the sprite is perfectly aligned to a tile and adding the width of the sprite will put it into the next tile so you need to subtract 1 from it. That is when you are moving the sprite straight up or straight down. To find

the next tile for the **Y** value you take the **Y** property of the rectangle and add the height of the rectangle. Again there can be the alignment problem I spoke of earlier and you will need to subtract 1 from it. This happens only when moving straight left of straight right. You again use the **VectorToCell** method to get the tile as a **Point**.

I then declared a variable to hold whether or not there is a collision with a tile. Now that I have the two tiles I need to check to make sure that the next tile is never out of the bounds of the map. The tile being less than 0 or greater than or equal to the width or height of the map. Since **doesCollide** it is set to false initially it will returns the opposite if the sprite will go out of the bounds of the map. Then there is either a nested loop or a single loop that will go through all of the tiles the sprite is in and check to see if the tile they collide with can not be walked into. If there is a collision detected the **doesCollide** variable is set to true. All of the methods return **doesCollide** at the end of the method.

The **CheckLeftAndUp** method checks to see if the the **X** and **Y** properties of the first tile are not less than zero. Since I've set **doesCollide** to be false initially I return the opposite meaning that

there was a collision and the movement will not take place. This is a case where I loop through all of the possible tiles to check if they collide. This is because the sprite is moving diagonally.

The **CheckUP** method is one of the cases where I need to deduct one for the width of the second tile. As I mentioned this is because the height of the sprite is the same as the height of the tiles and if the **Y** position lines up perfectly with a tile the method will think the sprite is in two tiles when it is actually only in one. This method just needs to check to make sure that the **Y** property of the first tile is less than zero because there is no horizontal motion. For this I also only need to check the tiles horizontally using the new **Y** property of the first tile. You do need to check the **X** property of all tiles though. If you don't and the top left corner of the sprite is in a passable tile and any other part of the sprite is not it will be able to pass through a tile it shouldn't be able too.

The third method, **CheckUpAndRight** method first checks that the **Y** property of the first tile is not zero and the the **X** property of the last tile is not greater than or equal to the width of the map. The reason why I use the last tile is it includes the width of the sprite. This is like when you are locking the sprite to the screen. To keep it from going off the screen you need to include the width. Because this is a diagonal you need to check all of the tiles as well.

The **CheckLeft** method is one of the cases where I need to deduct one from the height of the tile. Again, this is an alignment issue. This method only needs to make sure the the **X** property of the first tile is not less than zero because there is no vertical motion involved. You only need to check the **X** property of the first tile but you should check the **Y** property of all tiles for a similar reason as the **CheckUp** method.

The **CheckRight** method is similar to the **CheckLeft** method in that it is a case for the alignment issue. You check to make sure that the **X** property of the last tile is not greater than or equal to the width of the map. Because you are checking for motion to the right you only need to check the **X** property of the last tile but you need to check all of the **Y** properties.

The next method is the **CheckDownAndLeft** method. This method checks to make sure that the **X** property of the first tile is not less than zero. It also checks to make sure the **Y** property of the last tile is not greater than or equal to the height of the map. This is like checking to make sure the **X** property for the last tiles is not greater than or equal to the width of the map. Because the motion is on a diagonal it is a good idea to check all of the tiles for collision.

The **CheckDown** method needs to make sure the **X** property of the last tile is not out of the bounds of the map. This is the last case where there is an alignment issue and I deduct one from the width of the sprite. You really only need to check the **Y** property of the last tile for collision because the motion is vertical. You do need to check the full range of the **X** values though.

That leaves the **CheckDownAndRight** method. Since the motion is down and to the right it checks to make sure the **X** and **Y** properties are not greater to the width and height of the map respectively. Again, this is because you need to take the width and height of the sprite into consideration when you checking to make sure the sprite does not go off the edge of the map. Again, it is a good idea to check all of the tiles for collision.

That is probably one of the most complex tutorials so far. I hope that I didn't lose any of you along the way. If you have any questions as to how this works you can leave a comment on my blog or use the contact form on my web site and I will try and answer your questions as best I can.

That is it for this tutorial. I'm not sure what the next tutorial will be about. I'm considering on adding interaction with other sprites or 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.