Table of Contents

Introduction

This tutorial about tile based games is based on the old TonyPa flash tutorials. The original tutorial website has since dissapeared from the internet, but is still accessable on the way back machine. I also have the PDF version stashed away.

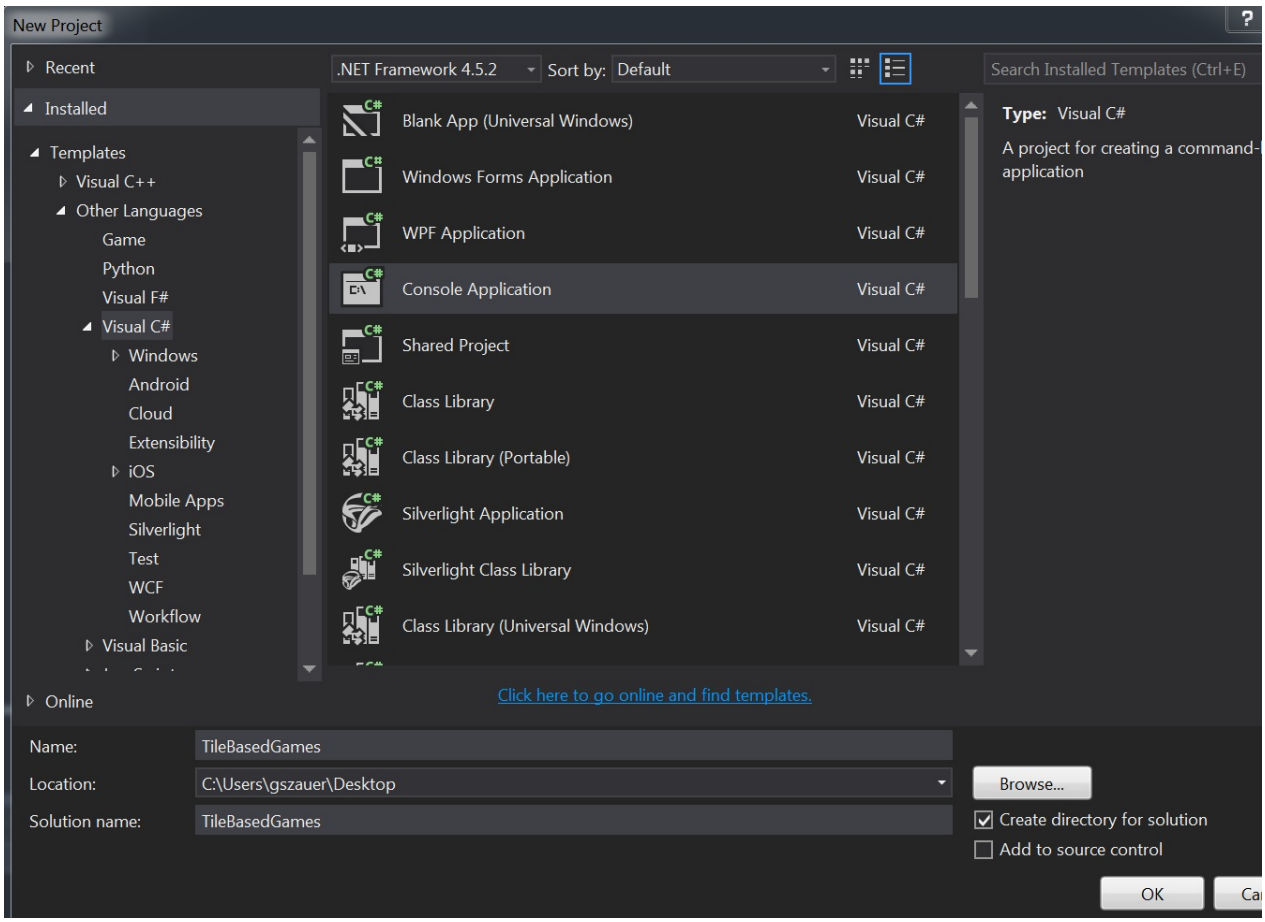
Converting these tutorials from AS2 to C# is no small feat, the languages are unbelivably different and the workflows don't match up at all. With that being said, the two tutorials are not the exact same. I have inserted sections i feel are important and remove sections i deem as not important. I also try to go into MUCH more detail than the original.

License

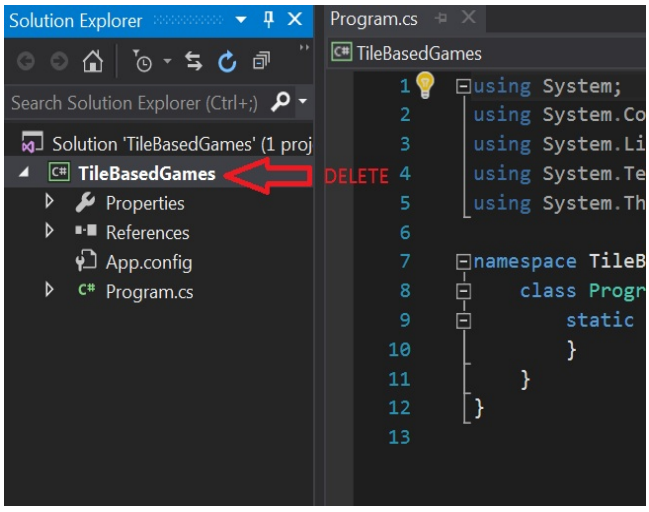
Starting Out

Setting up a solution

Make a new visual studio solution, i called mine "TileBasedGames"

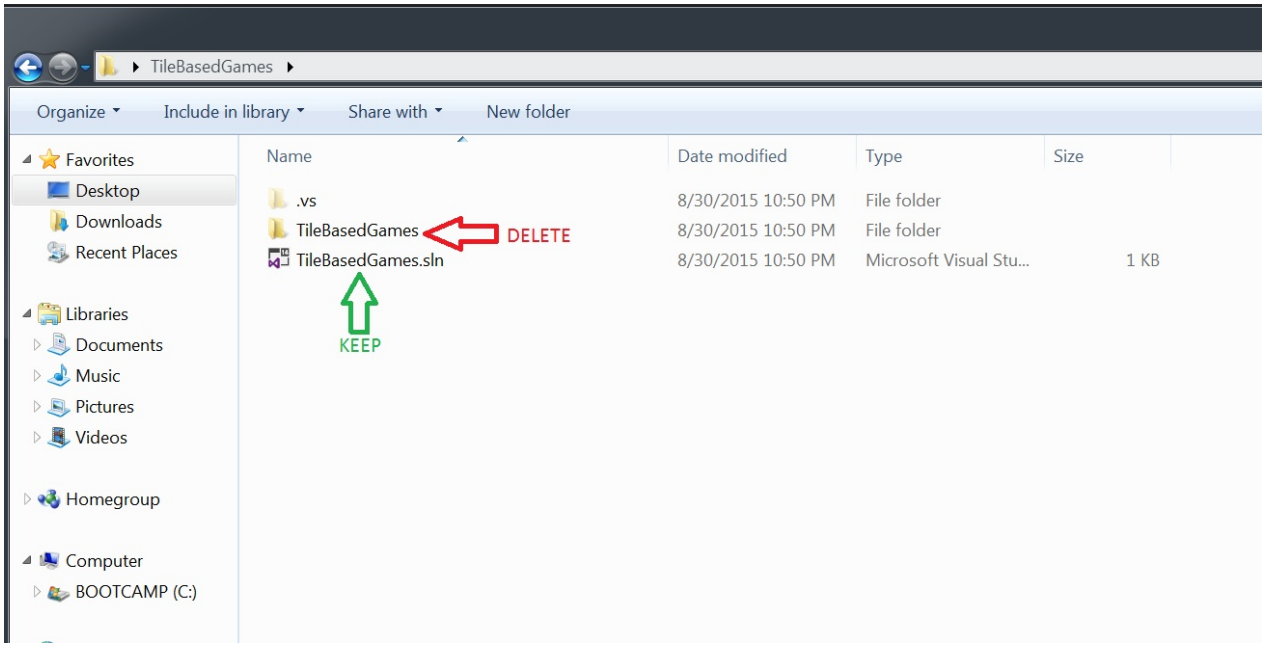


Notice that this created a new solution called TileBasedGames as well as a new project called TileBasedGames. Go ahead and delete the project. Right click the project and select Remove. Click "OK" on the confirm dialog.



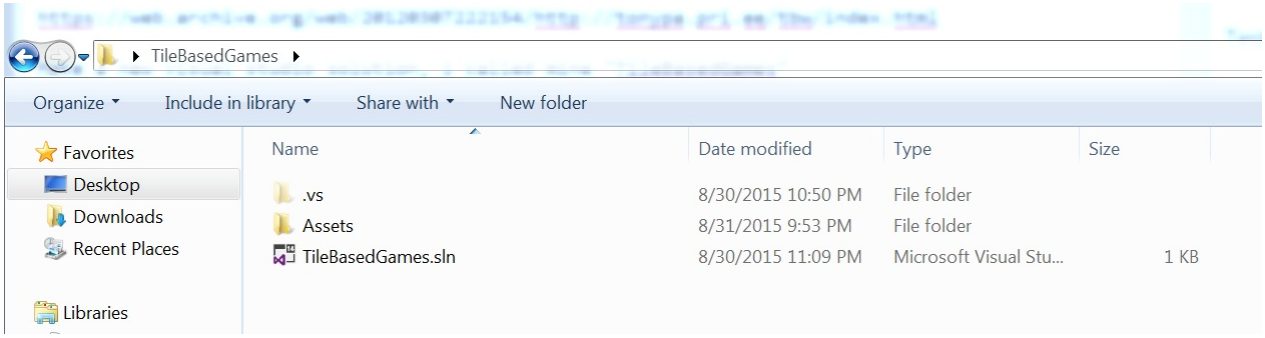
If you look at the "TileBasedGames" folder there are two things in there, a solution named "TileBasedGames" and a

folder named "TileBasedGames". The folder contains the project we just deleted. Go ahead and delete that folder.



Before we do anything else, add a .gitignore file. Use the Visual Studio template.

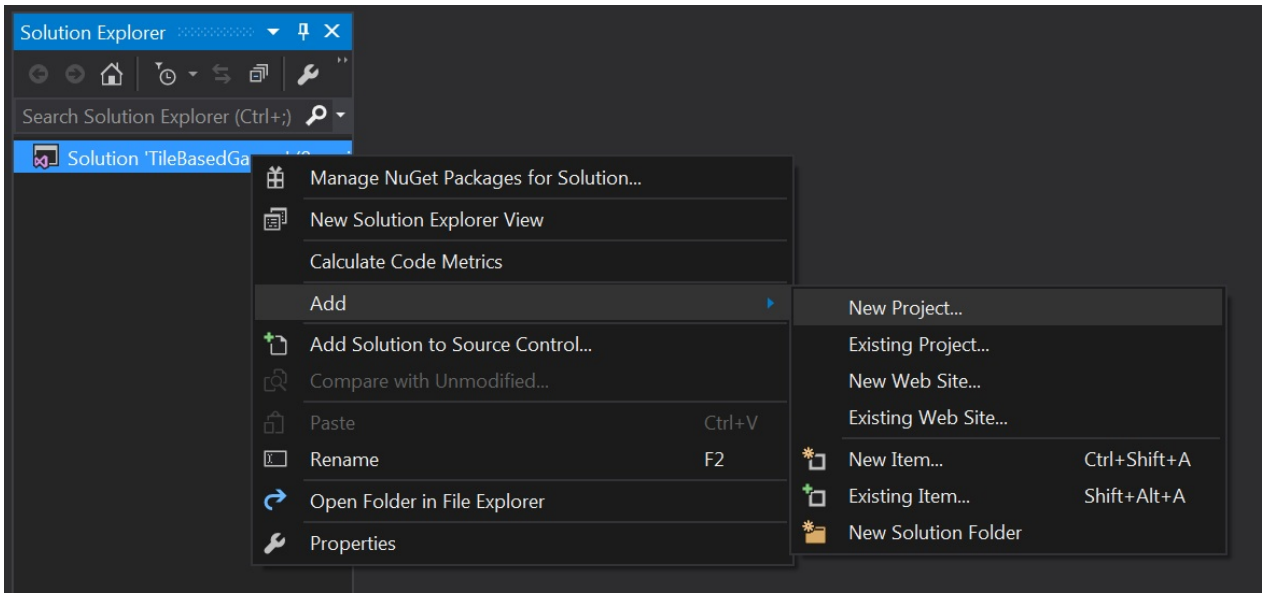
Lets also add an Assets folder next to the solution. This is where we are going to keep all shared game assets. All in all your folder structre should look like this:



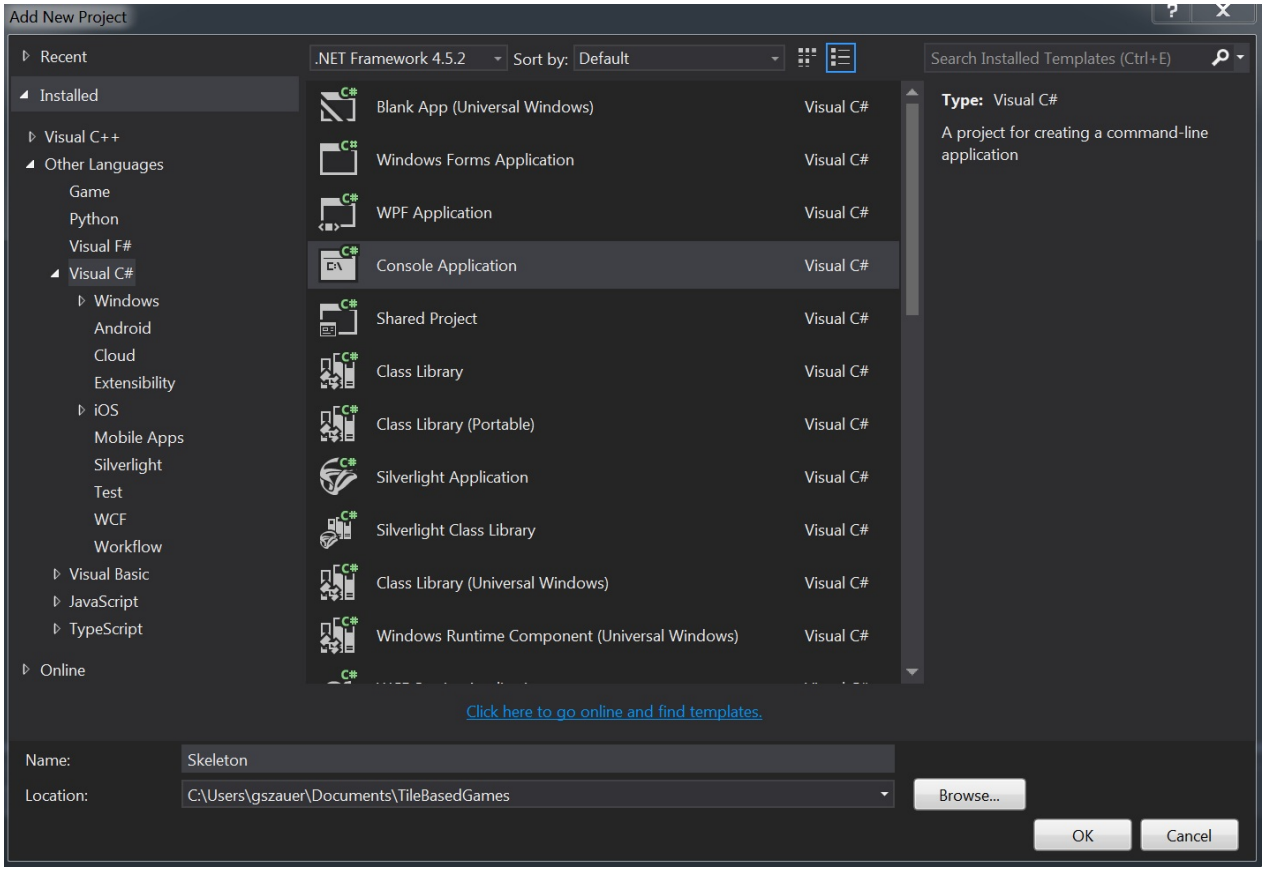
Setting up a project

Let's go trough setting up a project together. While you won't use this specific project, all subsequent tutorials will have the same setup. Ideally you want to have one solution with many projects inside it.

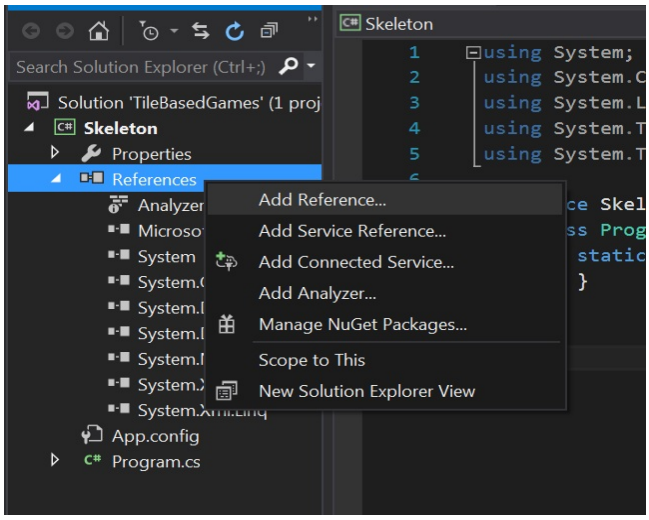
Let's add a new project called **Skeleton**. Right click on your solution and add a new project.



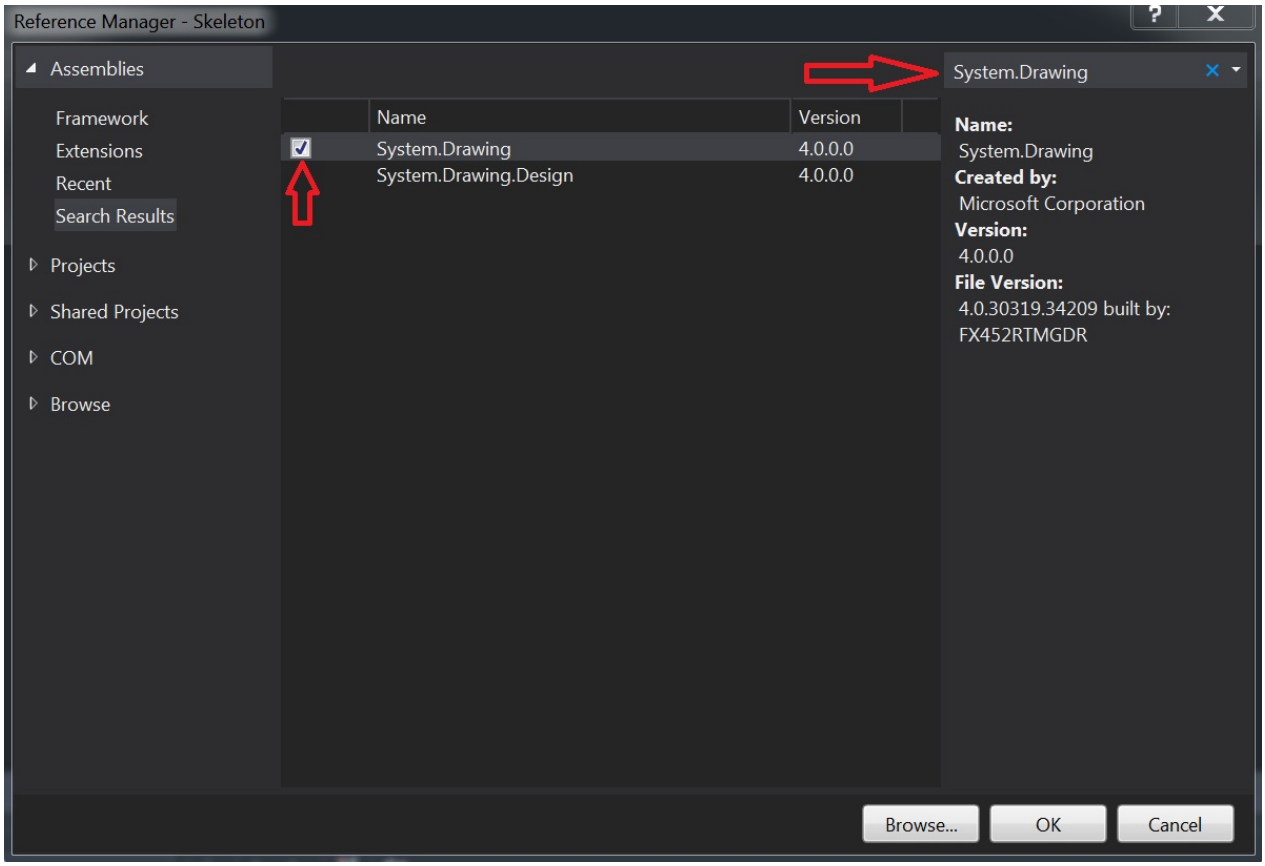
I'm going to go ahead and call mine **Skeleton**



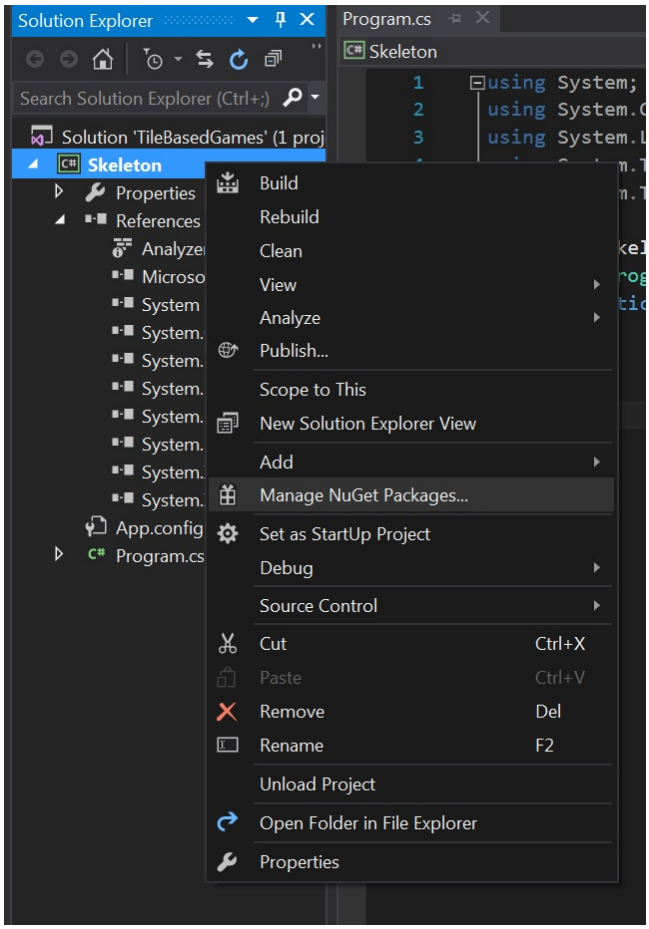
Link in System.Drawing. Right click on references and select "Add Reference"



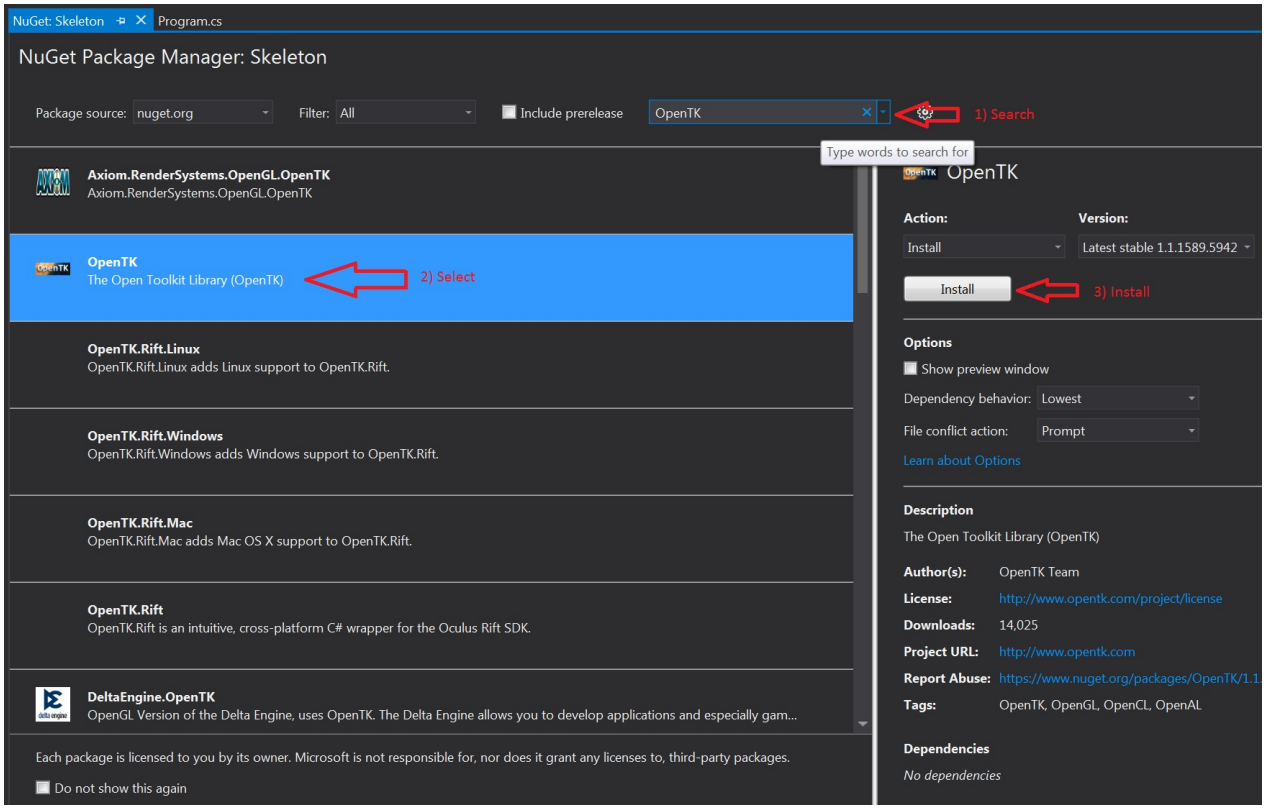
Type *System.Drawing* in the search box, check the appropriate package



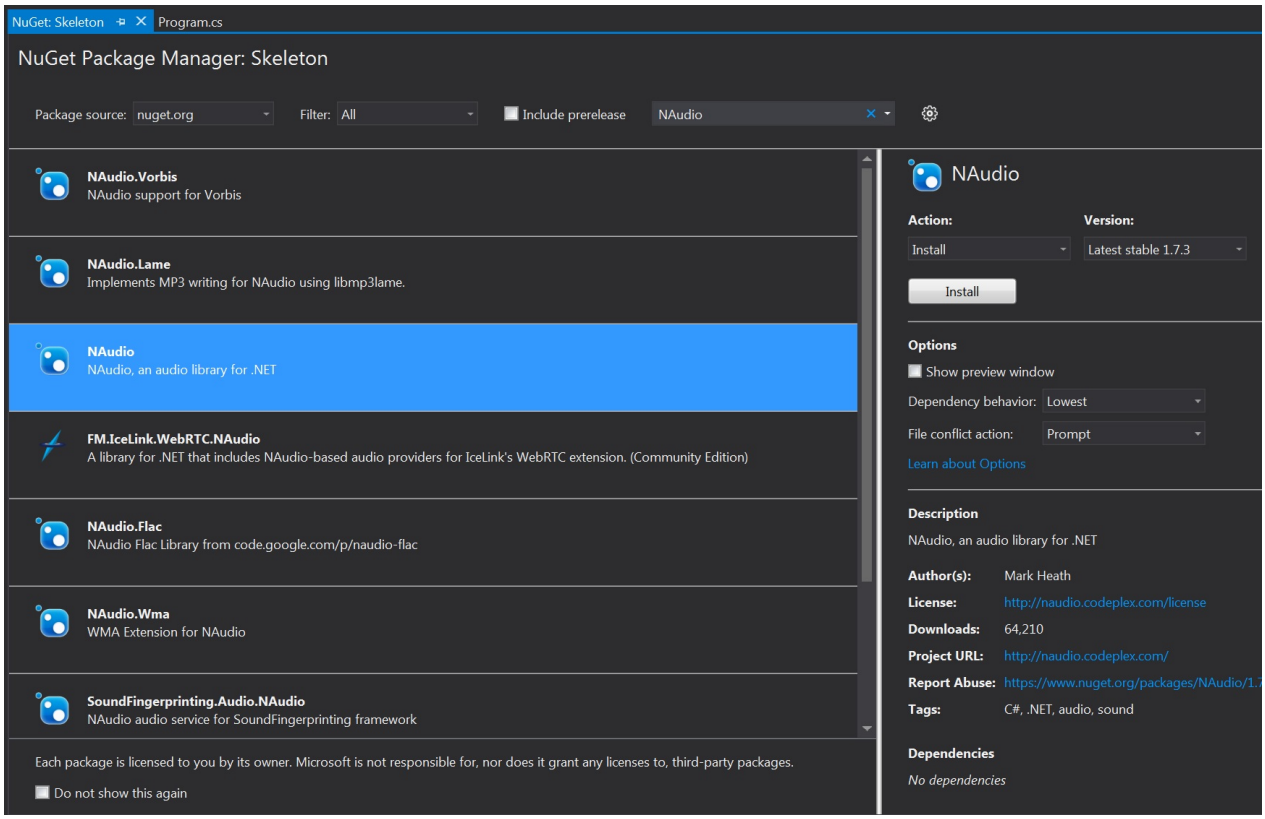
Next we're going to link in two NuGet packages, OpenTK and (https://naudio.codeplex.com/). Right click on the project and select **Manage NuGet Packages**



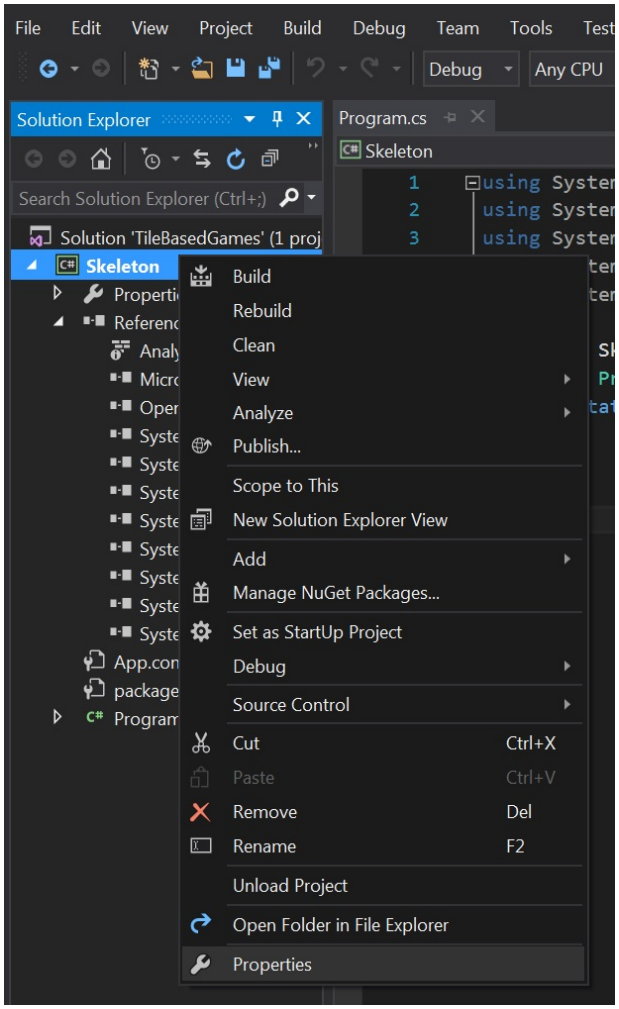
In the window that pops up, type **TopenTK** in the search bar, select the appropriate package and hit install.



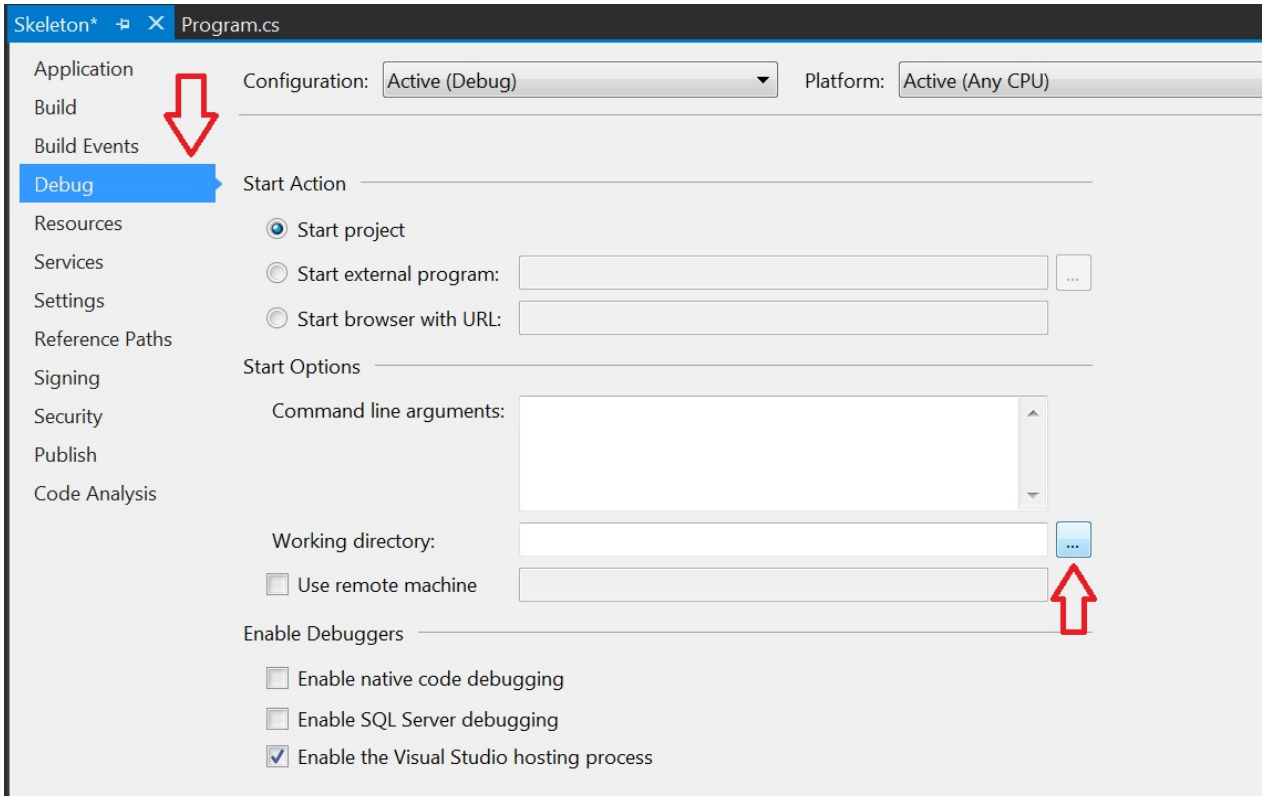
Repeat for NAudio



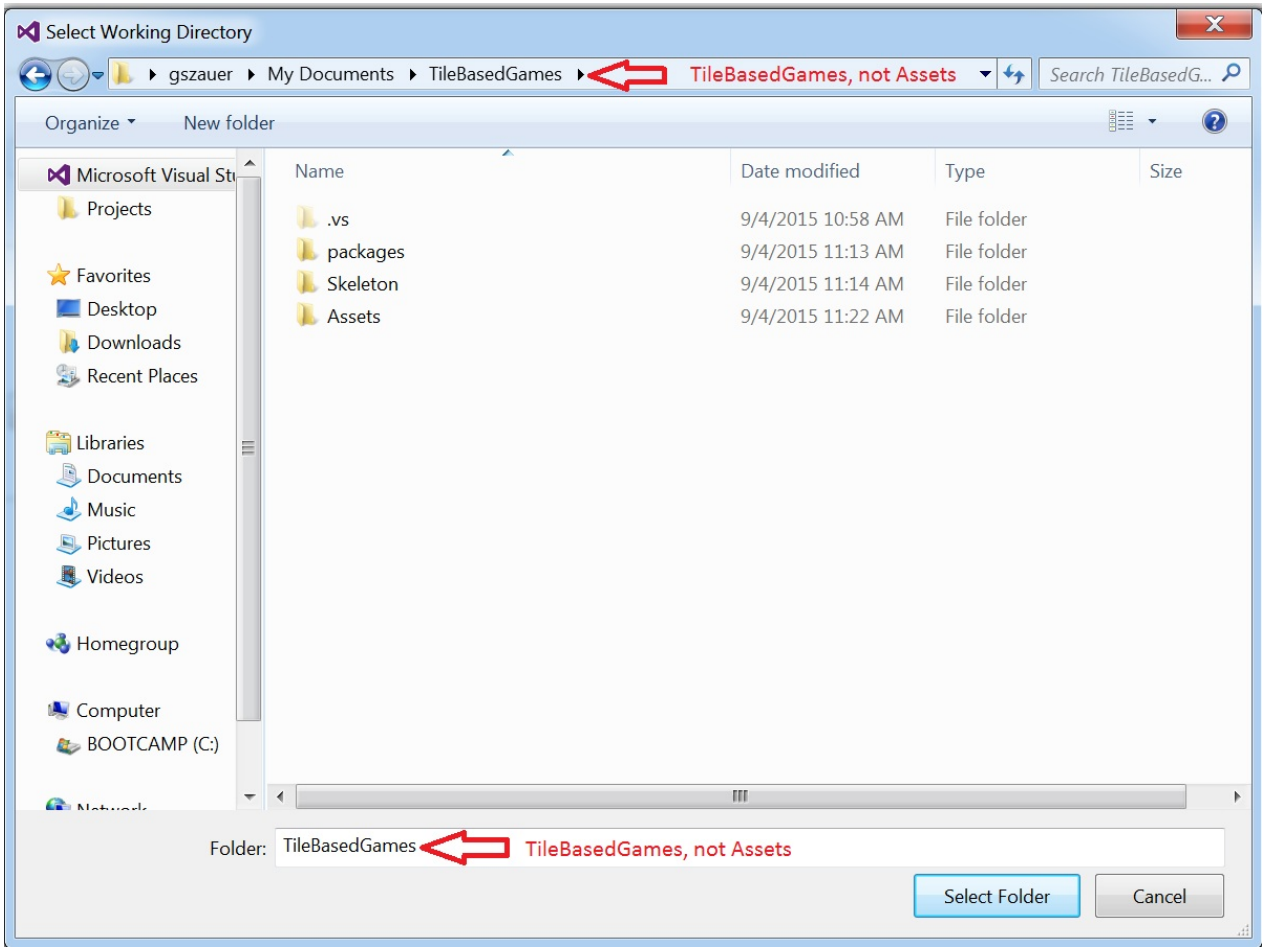
We set up an **Assets** folder when we created the soltuion. Let's set the projects working directory so we can actually use the Assets folder. Right click on the solution, select *Properties*



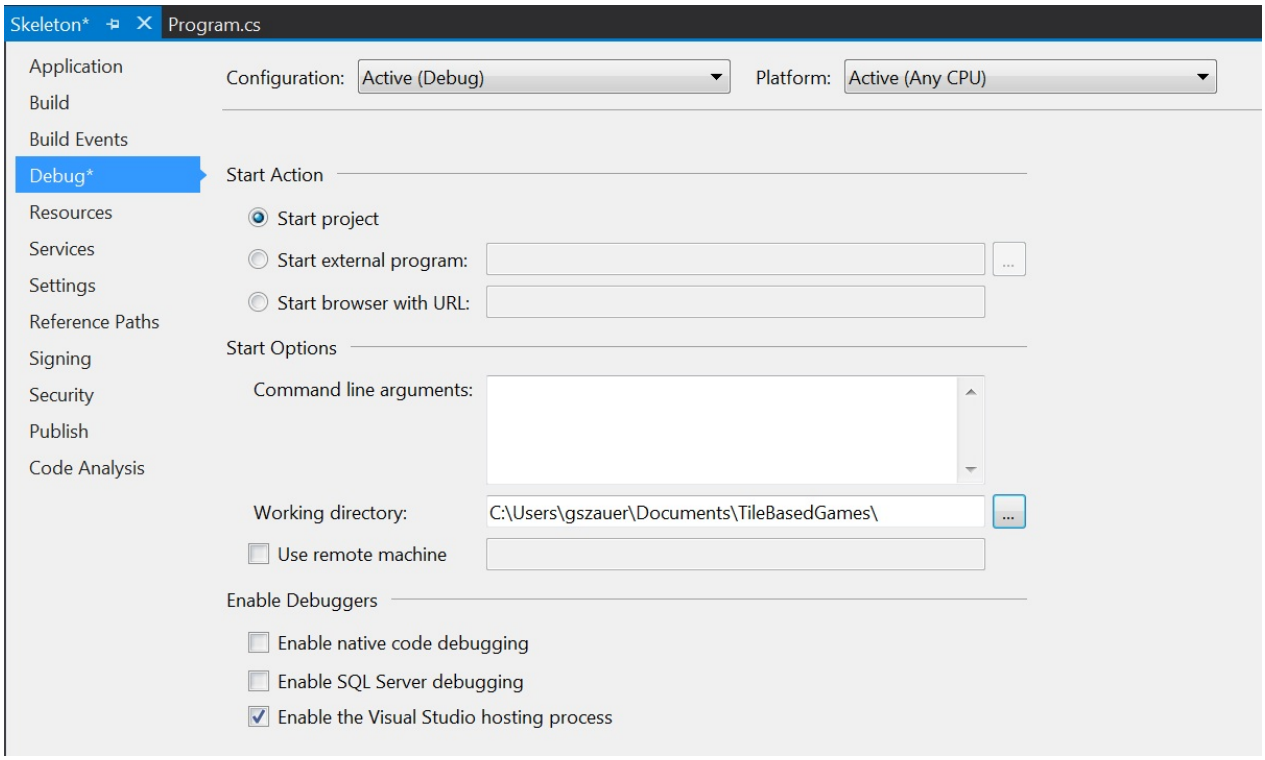
In the debug tab, browse for a new working directory



Select the **TileBasedGames** root directory. THIS IS NOT THE **Assets** DIRECTORY, RATHER THE Assets DIRECTORY'S PARENT.



We want to select the parent of the Assets directory. This way when you are ready to distribute your game, you just have to copy the **Assets** directory next to your exe file. Your properties should look like this:

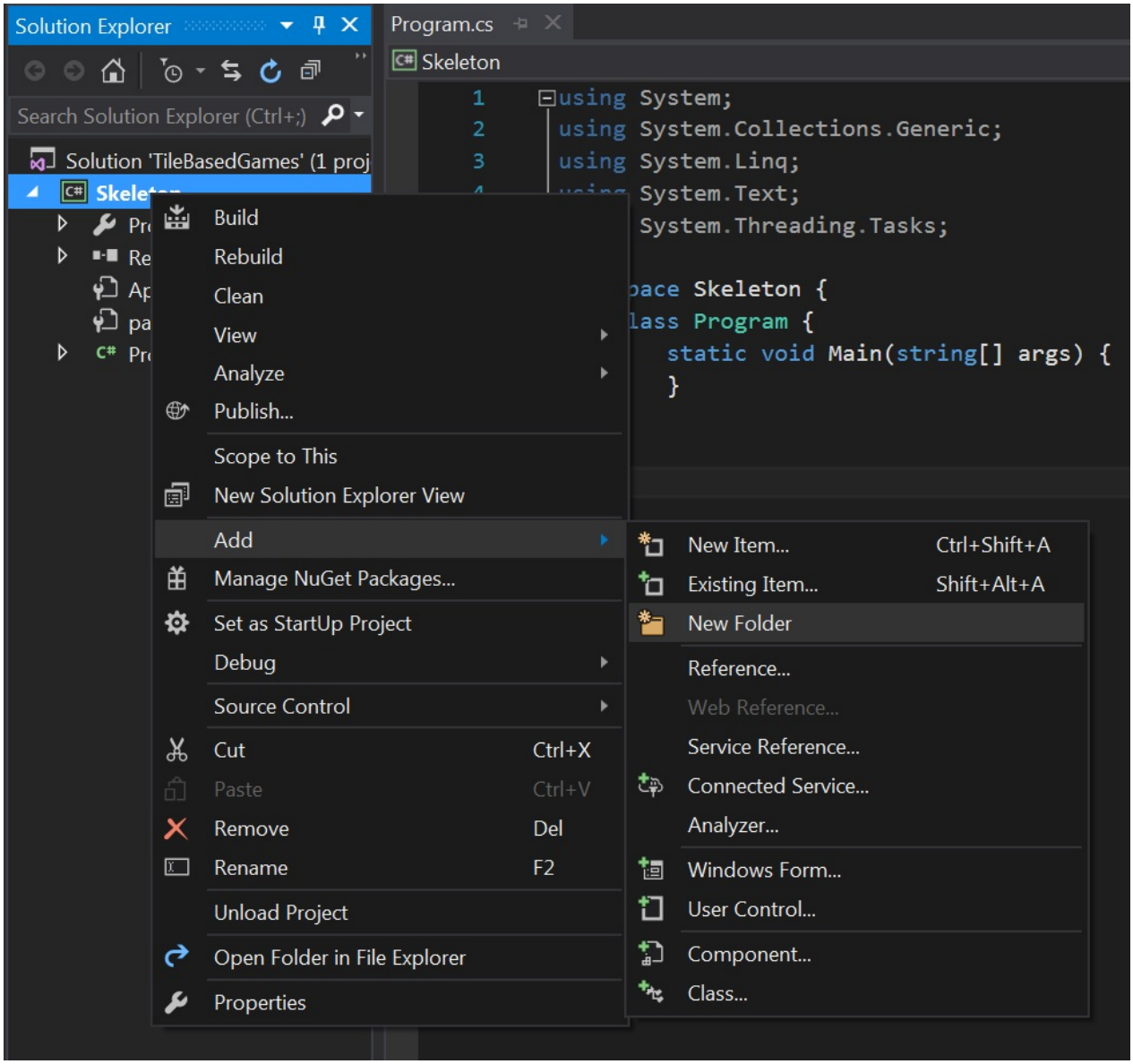


That is our basic project setup minus any code. **The Assets Directory Is Shared**. All of the projects we build in this tutorial will use the same **Assets** directory!

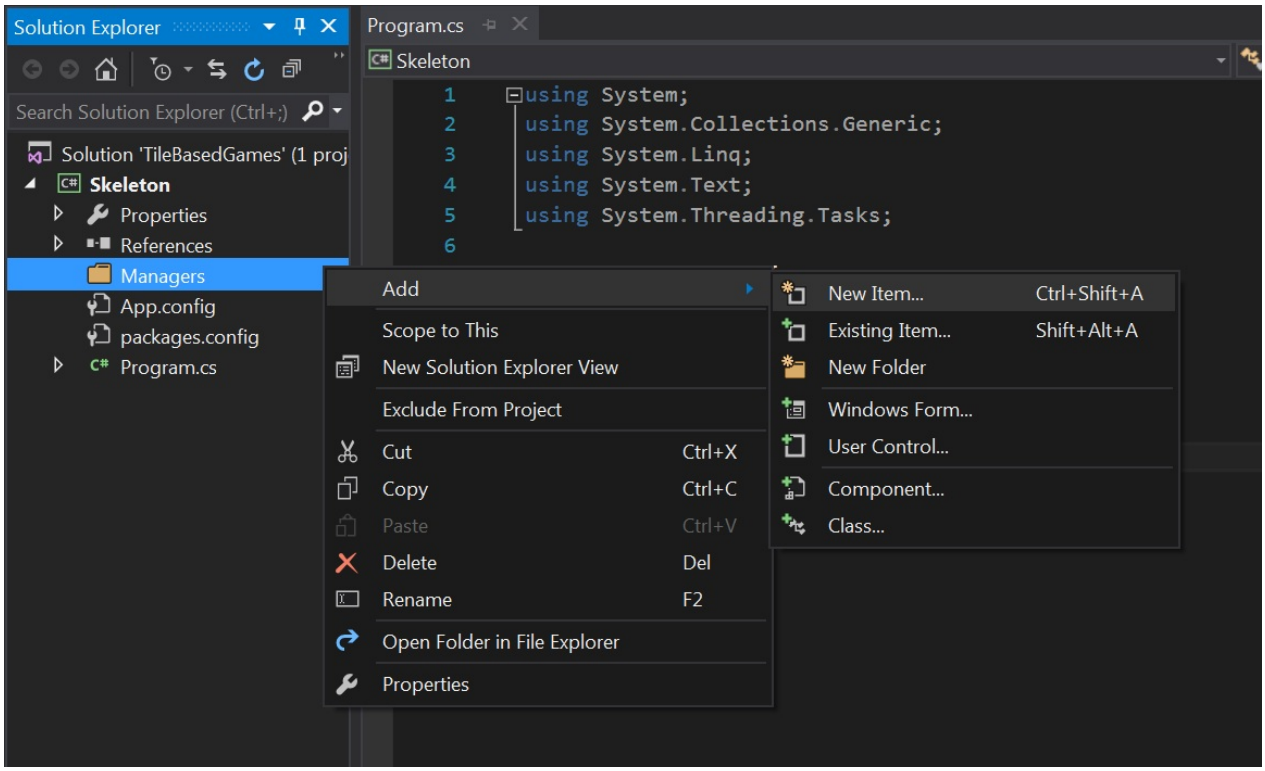
Adding code

We're going to add the OpenTK Framework Managers to this project. I'll link to the code on the main repo, but i'll also include it in the tutorial, in case the repo goes down.

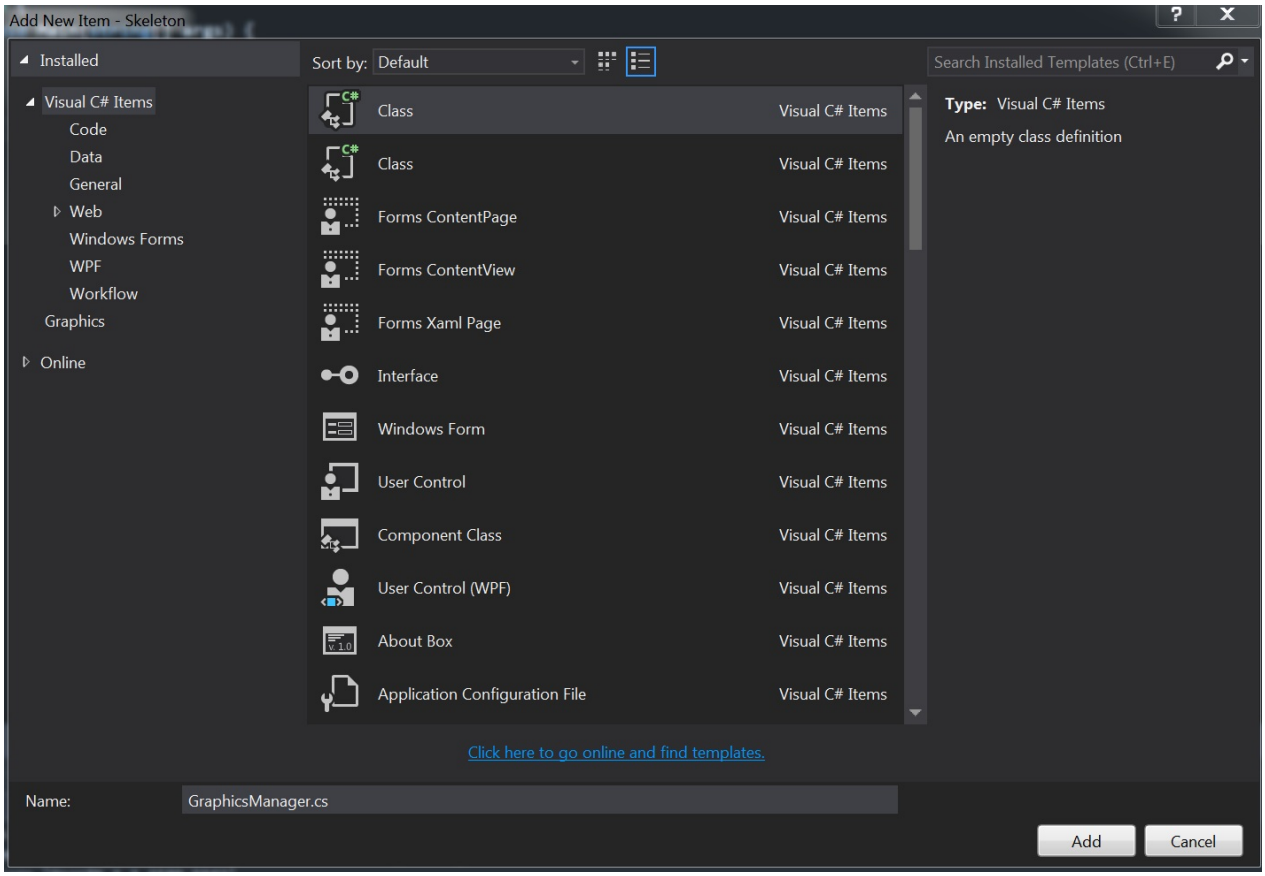
Right click on the project, select Add, then **New Folder**



Go ahead and name this folder **Managers**. Right click on the new **Managers** folder, select add, then new item



Call the new item **GraphicsManager**



Replace the contents of the new **GraphicsManager.cs** file with this. Make sure to use the latest code from the repo, as some of the code might have changed since you last saw it.

Repeat the above steps to add the TextureManager, InputManager and SoundManager. In case the main repo ever goes down i've included the source for all the managers in the tutorial, check the sub-sections of this section.

Let's add a new **Game** class, this is the class that we will be writing most of the example code in. Ideally you shouldn't have to touch any other classes. Add the game file to the solution (it's a top level file, next to Program.cs, not in the managers directory), and fill it with the following code:

```
using System;
using GameFramework;
using System.Collections.Generic;

namespace Skeleton {
    class Game {
        // Singleton
        private static Game instance = null;
        public static Game Instance {
            get {
                if (instance == null) {
                    instance = new Game();
                }
                return instance;
            }
        }

        protected Game() {

        }

        public void Initialize() {

        }

        public void Update(float dt) {

        }

        public void Render() {

        }

        public void Shutdown() {

        }
    }
}
```

The class is a singleton, each function in here should be relativley straight forward. The last thing we have to do is fill in the main Program. Change the contents of **Program.cs** to

```
using System;
using OpenTK;
using OpenTK.Input;
using System.Drawing;
using GameFramework;
using System.Collections.Generic;

namespace Skeleton {
    class Program {
        public static OpenTK.GameWindow Window = null;

        public static void Initialize(object sender, EventArgs e) {
            GraphicsManager.Instance.Initialize(Window);
            TextureManager.Instance.Initialize(Window);
            InputManager.Instance.Initialize(Window);
            SoundManager.Instance.Initialize(Window);
            Game.Instance.Initialize();
        }

        public static void Update(object sender, FrameEventArgs e) {
            float deltaTime = (float)e.Time;
            InputManager.Instance.Update();
```

```
            Game.Instance.Update(deltaTime);
        }

        public static void Render(object sender, FrameEventArgs e) {
            GraphicsManager.Instance.ClearScreen(Color.CadetBlue);

            Game.Instance.Render();

            // Show framerate
            int FPS = System.Convert.ToInt32(1.0 / e.Time);
            GraphicsManager.Instance.DrawString("FPS: " + FPS, new PointF(5, 5), Color.Black);
            GraphicsManager.Instance.DrawString("FPS: " + FPS, new PointF(4, 4), Color.White);

            GraphicsManager.Instance.SwapBuffers();
        }

        public static void Shutdown(object sender, EventArgs e) {
            Game.Instance.Shutdown();
            SoundManager.Instance.Shutdown();
            InputManager.Instance.Shutdown();
            TextureManager.Instance.Shutdown();
            GraphicsManager.Instance.Shutdown();
        }

        [STAThread]
        public static void Main() {
            // Create static (global) window instance
            Window = new OpenTK.GameWindow();

            // Hook up the initialize callback
            Window.Load += new EventHandler<EventArgs>(Initialize);
            // Hook up the update callback
            Window.UpdateFrame += new EventHandler<FrameEventArgs>(Update);
            // Hook up the render callback
            Window.RenderFrame += new EventHandler<FrameEventArgs>(Render);
            // Hook up the shutdown callback
            Window.Unload += new EventHandler<EventArgs>(Shutdown);

            // Set window title and size
            Window.Title = "Game Name";
            Window.ClientSize = new Size(800, 600);

            // Run the game at 60 frames per second. This method will NOT return
            // until the window is closed.
            Window.Run(60.0f);

            // If we made it down here the window was closed. Call the windows
            // Dispose method to free any resources that the window might hold
            Window.Dispose();

#if DEBUG
            Console.WriteLine("\nFinished executing, press any key to exit...");
            Console.ReadKey();
#endif
        }
    }
}
```

Program.cs just takes care of initializing and cleaning up our various managers. All of your actual game code will go into the **Game** class.

You should be good to go. Compile and run the game to make sure that the standard window pops up. You will likeley have to do the above steps for EVERY project you make. Don't Link to the managers, include them in each project.

Checkin

Go ahead and commit what you have to github. Before moving onto the next section check in with me. I want to look

over your code, make sure that the project is properly organized before moving on.

Managers

I originally included all the Managers as inline text on this page, but that ended up crashing web browsers, so instead i want to provide some useful links:

- Download all Managers in one zip file
- Graphics Manager Source
- Texture Manager Source
- Sound Manager Source
- Input Manager Source

Game (Skeleton)

In case the code repo ever goes down, this is a local backup of the **Game** class

```
using System;
using GameFramework;
using System.Collections.Generic;

namespace Skeleton {
    class Game {
        // Singleton
        private static Game instance = null;
        public static Game Instance {
            get {
                if (instance == null) {
                    instance = new Game();
                }
                return instance;
            }
        }

        protected Game() {

        }

        public void Initialize() {

        }

        public void Update(float dt) {

        }

        public void Render() {

        }

        public void Shutdown() {

        }
    }
}
```

Program

In case the code repo ever goes down, this is a local backup of the **Program** class

```
using System;
using OpenTK;
using OpenTK.Input;
using System.Drawing;
using GameFramework;
using System.Collections.Generic;

namespace Skeleton {
    class Program {
        public static OpenTK.GameWindow Window = null;

        public static void Initialize(object sender, EventArgs e) {
            GraphicsManager.Instance.Initialize(Window);
            TextureManager.Instance.Initialize(Window);
            InputManager.Instance.Initialize(Window);
            SoundManager.Instance.Initialize(Window);
            Game.Instance.Initialize();
        }

        public static void Update(object sender, FrameEventArgs e) {
            float deltaTime = (float)e.Time;
            InputManager.Instance.Update();
            Game.Instance.Update(deltaTime);
        }

        public static void Render(object sender, FrameEventArgs e) {
            GraphicsManager.Instance.ClearScreen(Color.CadetBlue);

            Game.Instance.Render();

            // Show framerate
            int FPS = System.Convert.ToInt32(1.0 / e.Time);
            GraphicsManager.Instance.DrawString("FPS: " + FPS, new PointF(5, 5), Color.Black);
            GraphicsManager.Instance.DrawString("FPS: " + FPS, new PointF(4, 4), Color.White);

            GraphicsManager.Instance.SwapBuffers();
        }

        public static void Shutdown(object sender, EventArgs e) {
            Game.Instance.Shutdown();
            SoundManager.Instance.Shutdown();
            InputManager.Instance.Shutdown();
            TextureManager.Instance.Shutdown();
            GraphicsManager.Instance.Shutdown();
        }

        [STAThread]
        public static void Main() {
            // Create static (global) window instance
            Window = new OpenTK.GameWindow();

            // Hook up the initialize callback
            Window.Load += new EventHandler<EventArgs>(Initialize);
            // Hook up the update callback
            Window.UpdateFrame += new EventHandler<FrameEventArgs>(Update);
            // Hook up the render callback
            Window.RenderFrame += new EventHandler<FrameEventArgs>(Render);
            // Hook up the shutdown callback
            Window.Unload += new EventHandler<EventArgs>(Shutdown);

            // Set window title and size
            Window.Title = "Game Name";
            Window.ClientSize = new Size(800, 600);

            // Run the game at 60 frames per second. This method will NOT return
```

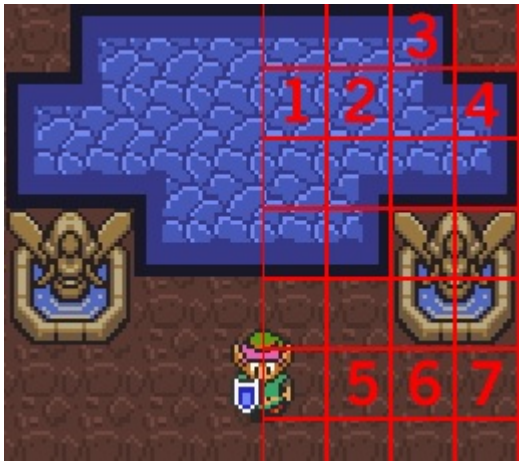
```
            // until the window is closed.
            Window.Run(60.0f);

            // If we made it down here the window was closed. Call the windows
            // Dispose method to free any resources that the window might hold
            Window.Dispose();

#if DEBUG
            Console.WriteLine("\nFinished executing, press any key to exit...");
            Console.ReadKey();
#endif
        }
    }
}
```

Why Tiles

Before delving into how to make tile based games, why do we use tiles to being with? Are they easier to make than their free form counterparts? Do they run faster? Is C# even a good candidate to make tile based games?

Tile based games came about back in the olden days of computing. When processor speeds where counted in megahertz, not gigahertz. When memory was kilobytes not megabytes. The limitation of memory and processing power kept programmers from being able to display a 512 x 512 sprite while keeping framerates reasonable.

Without being able to display large textures how can one draw a nice background? Slice the picture up into tiles! You don't need to waste memory on areas of the background that look the same.



In the picture you can see that parts of picture are exactly same. 1 is same as 2, 3 is same with 4 and parts 5-7 are all same thing. If you slice up the picture and reuse same parts in different areas, you have created the tiles. By only storing tiles we save both disk space and memory.

An unexpected side effect of using tiles is how easy they make it to update your game. If you are making a huge level and realize that you don't want to have the town in the lower left corner of the map, rather the upper right you can edit the tile map without having to edit any images. The dynamic nature of tiles makes developing games with tiles much faster, putting far less stress on the need of art support.

C# And Tiles

With modern hardware we can render thousands if not millions of polygons and textures at the same time. There is no practical need for tiles when we could just render huge maps. So why use tiles at all? Aside from the ease of development tiles provide a distinct retro art style. Also, the concepts of tile based games will be useful for developing any type of game.

C# isn't a particularly bad or good fit for tiles. It supports multidimensional (jagged) arrays and the ability to draw sprites. That's really all you need to create tile based games. With that in mind, let's jump in and get started.

Map Format

Knowing why we're going to use tiles, let's talk about how we are going to be treating them. One of the most important things in any good tile based game is the map. The map contains our tile data, the levels layout as well as information about each tile. Let's explore different ways we can represent this map.

Two dimensional arrays

I'm going to talk about two dimensional arrays a LOT. You will quickly find that i don't mean proper C# 2D arrays. I don't EVER use those.

When i talk about a 2D array i'm talking about a jagged array. An array of arrays, like so:

```
int[][] twoDim = new int[5];
for (int i = 0; i < twoDim.Length; ++i) {
    twoDim[i] = new int[5];
}
```

Map data

Now that we know we're storing the map in a 2D array, the big question is what the data type of the array supposed to be? With the map we want to have tiles have properties, these should denote things like Is the tile walkable, what sprite should the tile use? etc... We have two options to storing map data:

Tile Object: It only makes sense to create a tile object. This object could have member variables denoting IsWalkable, RenderSprite, etc... Tile objects have the advantage of being easy to use. Want to check if an object is walkable? It's as simple as:

```
if (game_map[tileX][tileY].IsWalkable) {
    DoWalk();
}
```

However there is a downside to this method of doing things. It becomes pretty complicated to save the map data. You can't really serialize a two dimensional array of objects to a text file cleanly.

Integers: The other option is to create a two dimensional array of integers. Each integer represents a tile type. How would you figure out if a given tile is walkable or not, how would you get the sprite of a tile if we just stored a 2D array of integers? Well, you would need to create a TileManager (Singleton), something like this:

```
class TileManager {
    private List<int> sprites = null;
    private List<bool> walkable = null;
    // One list for each property

    // Singleton code & constructor that creates lists

    public int AddTile() {
        sprites.Add(-1);
        walkable.Add(false);
    }

    public void SetWalkable(int tile, bool canWalk) {
```

```
        walkable[tile] = canWalk;
    }

    public bool GetWalkable(int tile) {
        return walkable[tile];
    }

    // Getters / setters for other properties
}
```

Using this method we would find out if a tile is walkable like so:

```
if (TileManager.Instance.GetWalkable(game_map[tileX][tileY])) {
    DoWalk();
}
```

As you can see, the code has become a LOT more verbose. At runtime we need to do a considerable amount of typing to achieve the same thing as we did with objects. So, is there an upside to this method? Yes, writing the map to a file and reading it back becomes dead simple! We write a few lists, and a two dimensional array to our saved map file. It's quick and easy!

Next Steps

Having some context on the two main methods of storing tile info, i want you to take a few minutes and think about how you would do things. Which method makes more sense to you? What's more important, having clear code and messy data or having messy code and clear data? There is no silver bullet for anything in programming, everything is a compromise, it's your job as a programmer to find the best option.

Creating Tiles

The last section left off with an open ended question. What's more important clean code and messy data, or messy code and clean data? Seeing how we're making a game, not a tool i would say having clean code take priority. Games are complicated beasts, it's hard enough to make a game when you write good code, let's not over complicate the code needed for the sake of data.

Getting the assets

I'm going to be using this zelda tile set for the tutorial. Download it, and unzip the textures into your "Assets" directory. All of the sprite sheets in that folder are already a power of two.

New Project

Create a new project, call it **CreatingTiles** and make the standard window appear. Don't forget to set your working directory correctly. Follow this chapter along in this new solution.

Tile Object

The first thing we need is a tile object. For now each object will have two properties, a sprite to display and a boolean to dictate if we can walk on it or not. In the future we will add properties to this tile definition as needed (eq: IsDestructable, IsHidden, etc...). The name of this class is going to simply be `Tile`:

```
class Tile {
    public int Sprite { get; private set; }
    public Rectangle Source { get; private set; }
    public bool Walkable { get; set; }

    public Point WorldPosition { get; set; }
    public float Scale { get; set; }

    public Tile(string spritePath, Rectangle sourceRect) {
        Sprite = TextureManager.Instance.LoadTexture(spritePath);
        Source = sourceRect;
        Walkable = false;
        WorldPosition = new Point(0, 0);
        Scale = 1.0f;
    }

    public void Render() {
        Point renderPosition = new Point(WorldPosition.X, WorldPosition.Y);
        renderPosition.X = (int)(renderPosition.X * Scale);
        renderPosition.Y = (int)(renderPosition.Y * Scale);
        TextureManager.Instance.Draw(Sprite, renderPosition, Scale, Source);
    }

    public void Destroy() {
        TextureManager.Instance.UnloadTexture(Sprite);
    }
}
```

Lets see what the **Variables** do:

- Sprite - Once loaded, this is a reference to the sprite sheet we will be rendering
- Source - What section of the sprite sheet do we want to display
- Walkable - Can the player walk on this tile

- WorldPosition - What X / Y should the tile render a, this respects scale
- Scale - We might want to scale the tile to display on a larger screen
  - If we have a tile of size 16 at x position 2, it will draw at pixel 2 with a width of 16
  - The same (width 16, x 2) with a scale of 2 it will render at piel 4 with a width of 32

The **constructor** takes the path to a sprite sheet and a rectangle of the subsprite that represents this tile. It sets the appropriate member variables, and sets the rest to defaults.

Because the constructor called `LoadTexture` we must also call `UnloadTexture`. The **Destroy** method serves this purpose. Once the Tile is no longer needed the Destroy function should be called.

The **Render** function is super simple, it just uses all the members we have to render the sprite. It's a simple wrapper for the `Draw` function of *TextureManager*. The only work render does is to account for scaling and world position.

Map Data

So now that we have the definition of what an individual tile is, let's see if we can make a map. The `map` should be in a game class, you should be able to figure out how to add a game instance to the window.

```
class Game {
    protected Tile[][] map = null;
    protected int[][] mapLayout = new int[][] {
        new int[] { 1, 1, 1, 1, 1, 1, 1, 1 },
        new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
        new int[] { 1, 0, 1, 0, 0, 0, 0, 1 },
        new int[] { 1, 0, 0, 0, 0, 1, 0, 1 },
        new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
        new int[] { 1, 1, 1, 1, 1, 1, 1, 1 }
    };

    // Singleton (private static instance, public static getter, private constructor)
}
```

- **map**: a 2D array to hold runtime map information
- **mapLayout**: Meta-data to hold an easily configurable map.
  - 0's are walkable tiles
  - 1's are not walkable

As your game gets more and more complicated it may not be possible to store data in a simple representation like the one given above. I'm going to try my best to keep things as simple as possible for the tutorial.

Also take note that my game class is a singleton. You should make yours a singleton too, it will make hooking it up to the game window super easy.

Now that we know what the map is going to look like, the next step is to actually load the map in! In order to create a tile object we need to know two things, the sprite sheet it's going to be on and the rectangle on that sheet. Before going any further, lets also add this meta-data to the game class

```
class Game {
    protected Tile[][] map = null;
    protected int[][] mapLayout = new int[][] {
        new int[] { 1, 1, 1, 1, 1, 1, 1, 1 },
        new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
        new int[] { 1, 0, 1, 0, 0, 0, 0, 1 },
        new int[] { 1, 0, 0, 0, 0, 1, 0, 1 },
        new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
        new int[] { 1, 1, 1, 1, 1, 1, 1, 1 }
```

```
    };

    protected string[] spriteSheets = new string[] {
        "Assets/HouseTiles.png",
        "Assets/HouseTiles.png"
    };
    protected Rectangle[] spriteSources = new Rectangle[] {
        new Rectangle(466,32,30,30),
        new Rectangle(466,1,30,30)
    };
}
```

Why did we add two arrays? Well, map layout refers to tiles by numbers, each number coresponds to an element in one of the new arrays we just created. Do we need the `spriteSheets` array? Not really. If we have a "world map" atlas that contains EVERY tile in the game, then the `spriteSheets` array is a lot of extra work. But what if we want tiles with a number **5** and above to come from an alternate sprite sheet? The `spriteSheets` array lets us do just that.

Creating the Map

We now have all the information we need to build the map array. Some information like if a map is walkable or not is obvious, others like the X / Y location of a map we have to figure out. Let's create a helper function to create the map, and call said function in Initialize.

```
Tile[][] GenerateMap(int[][] layout, string[] sheets, Rectangle[] sources) {
    Tile[][] result = new Tile[layout.Length][];
    float scale = 1.0f;

    for (int i = 0; i < layout.Length; ++i) {
        result[i] = new Tile[layout[i].Length];

        for (int j = 0; j < layout[i].Length; ++j) {
            // The 0's and i's in the layout array correspond to
            // strings and rectangles in the sheets and sources arrays
            string sheet = sheets[layout[i][j]];
            Rectangle source = sources[layout[i][j]];

            // We don't have to take scale into account, but we do need
            // to space our grid out accordingly. It might be smart to have
            // a width and height that's a constant, right now we are assuming
            // that all of the passed in rect's in sources will have a uniform size
            Point worldPosition = new Point();
            worldPosition.X = (int)(j * source.Width);
            worldPosition.Y = (int)(i * source.Height);

            result[i][j] = new Tile(sheet, source);
            result[i][j].Walkable = layout[i][j] == 0;
            result[i][j].WorldPosition = worldPosition;
            result[i][j].Scale = scale;
        }
    }

    return result;
}

public void Initialize() {
    TextureManager.Instance.UseNearestFiltering = true;
    map = GenerateMap(mapLayout, spriteSheets, spriteSources);
}
```

With `GenerateMap` containing all of the logic for initializing tiles you can imagine that it's going to get more complicated as your tiles get more complicated. This is the expected behaviour, it's just something to keep in mind.

Initialize calls a method of TextureManager we have not talked about yet:

```
TextureManager.Instance.UseNearestFiltering = true;
```

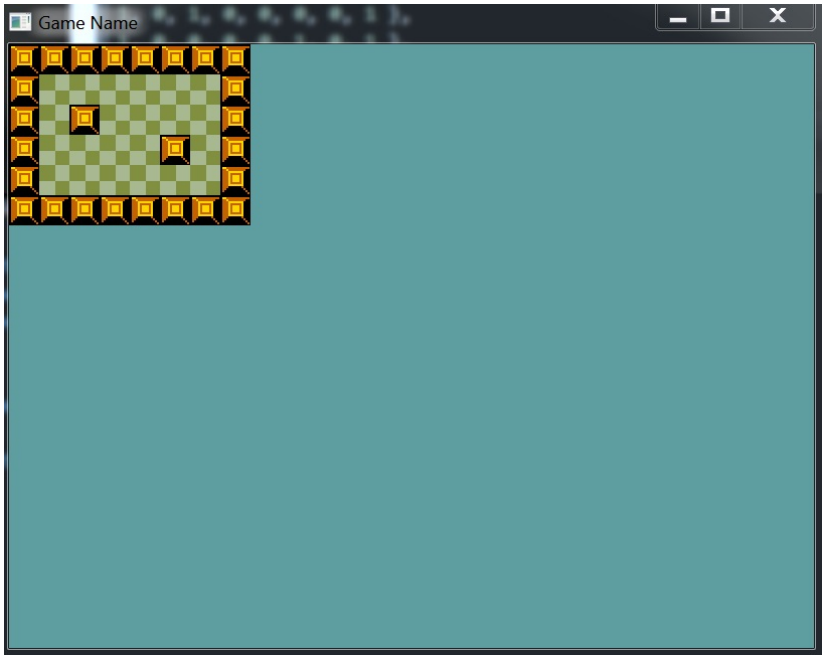
What does this do? Filtering is applied when an image is loaded. If filtering uses nearest, then everything resizes in a pixelated fashion. This is how you get pixel perfect images. If nearest filtering is NOT used then anti aliasing is applied to textures that are resized. This MAY cause blurry looking textures.

Displaying the Map

With all of that in place, i think we are ready to render our tiles! Because the tile class has a `Render` method that takes care of it's own rendering, this is going to be super simple. Inside the **Game** classes `Render` function just call `Render` on every tile:

```
public void Render() {
    for (int h = 0; h < map.Length; ++h) {
        for (int w = 0; w < map[h].Length; ++w) {
            map[h][w].Render();
        }
    }
}
```

The game should now look like this:



Cleanup

Don't forget, each tile loaded a sprite, we must now unload these!

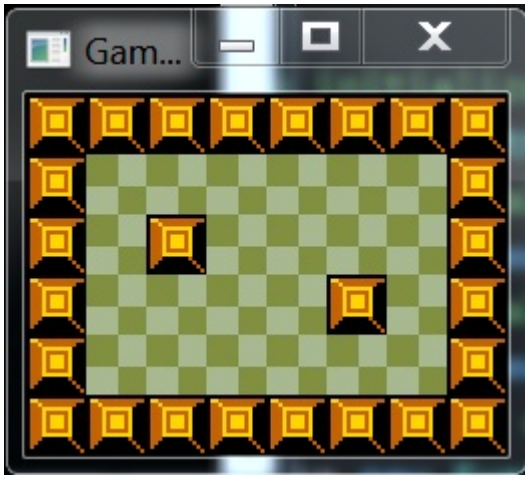
```
public void Shutdown() {
    for (int h = 0; h < map.Length; ++h) {
        for (int w = 0; w < map[h].Length; ++w) {
            map[h][w].Destroy();
        }
    }
}
```

Window size

If you want the tiles to fill out the full window, set the ClientSize of the window to **30 * 8** by **30 * 6**.

- 30 - width / height of sprite
- 8 / 6 - How many sprites we have horizontally (8) and vertically(6)

If our sprites had a scale factor to them, we would take that into account as well. Here is what the game looks like with the window size adjusted:



On Your Own

Now that we've covered how to create and draw a tile based map, it's time for you to play around a bit on your own. This challenge is going to test your understanding of creating tile based maps. You will need to create a single room. Try your best to write all the code for this setion from scratch, only use the code from the last section if you get stuck.

New Project

Create a new project, call it **TilePractice** and make the standard window appear. Don't forget to set your working directory correctly. Follow the instructions of this chapter in the new solution.

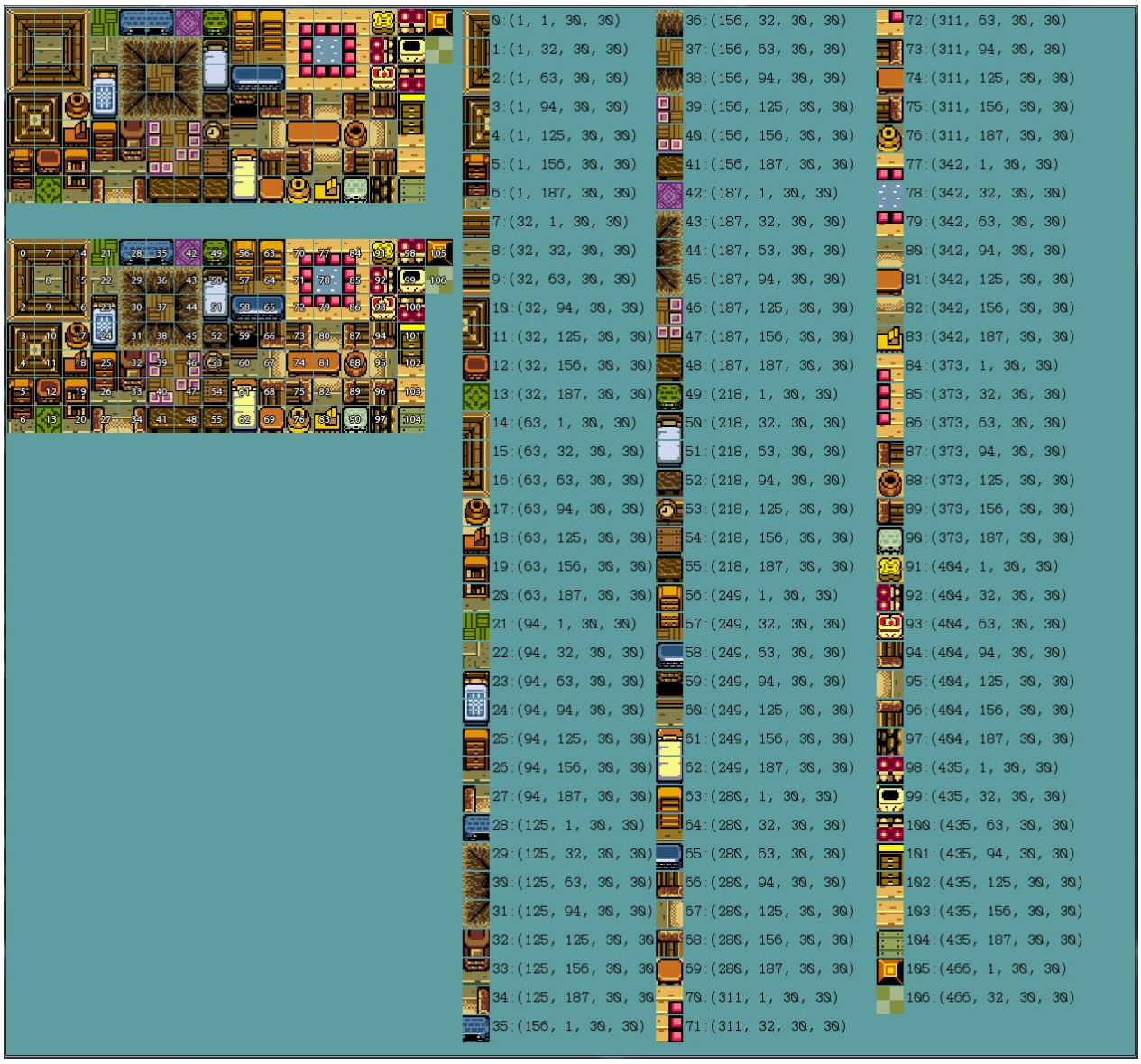
Tile Object

Make yourself a tile object. This can be extremly similar to the tile object from the **CreatingTiles** section. You do not need to worry about things like scale or walkability. Your tile needs to keep track of:

- Sprite Instance
- Source Rectangle
- World Position

Tile map

We are going to be re-using the **Assets/HouseTiles.png** sprite sheet for this project. It should already be in your shared Assets directory, assuming you set your working path correctly you shouldn't event need to add anything. Here is a position breakdown of each tile:



There are a lot of different ways to represent a tile map in memory, i strongly urge you to use the method we used in **CreatingTiles**. That is make a multidimensional array for the map layout, which indexes into two regular arrays for the sprite sheet names and sprite regions on the sheets.

Goal

Given the above information try to recreate this room:

Make sure that your window is a tight fit to the room

Extra Credit

This next part is optional, i suggest doing it as the extra practice tends to help.

Create another new solution, call it **TilePractice2**, repeat everything you just did, except instead of re-creating my room design design your own room using the sprite sheet. The room can be as big or as small as you want it to be.

Checkin

Before moving onto the next section check in with me. I want to look over the code for both **CreatingTiles** and **TilePractice**

Map Tools

As you can imagine, creating maps by hand becomes very tedious very fast! What we need is some tools to help us design these maps faster. Luckily, the internet is crawling with tools!

A quick google for 2D Tile Editor will result in numerous searches. Each editor has it's own set of strengths and weaknesses, you can probably figure out how most of them work.

Here, i want to go over two of what i think are the simplest / easyest tools to use to generate maps. These tools will not output our map format, but with a little bit of text editing we can transform their output into something useful.

Online Tilemap Editor

The Online Tilemap Editor is an HTML5 tile editor you can use right from your browser! Even more impressive, it's Open Source and on Github.

I like this editor because it's fast, easy and doesn't require any installation. It's a super quick editor to use, especially if you have a cleanly formatted map tiles, like our zelda map.

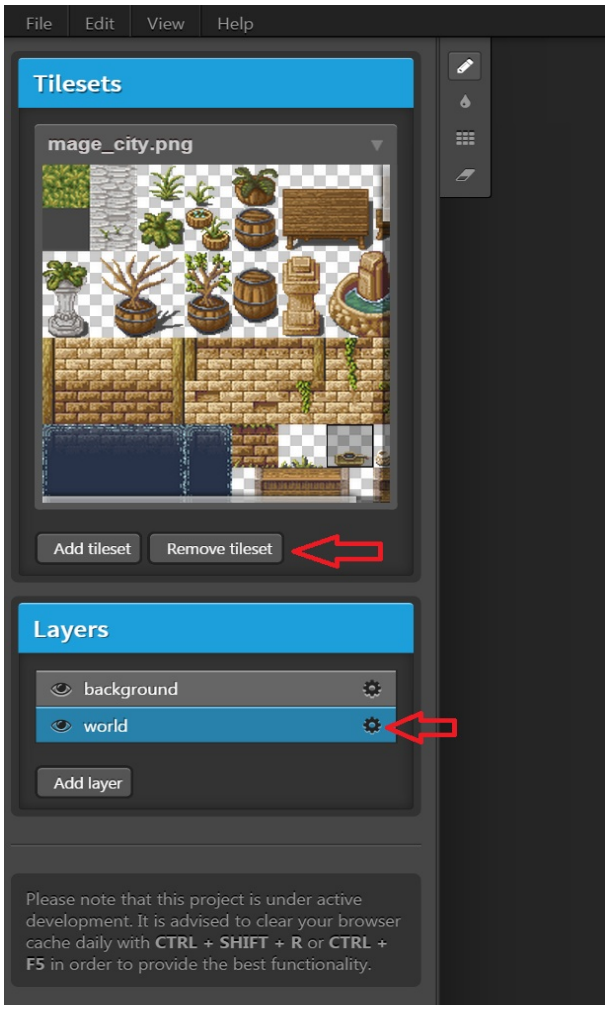
Using Online Tilemap Editor

Go to the Online Tilemap Editor website.

At this point the editor is generated with some default data.

Remove the tileset with the **Remove Tileset** button

Delete one of the layers by clicking the **gear icon** and selecting **Remove** from the dropdown menu.



Next click the **Add tileset** button, in the popup that follows:

- Browse for the tilemap image
- Tile Width is 30
- Tile Height is 30
- Tile Margin is 0
- Keep the Tile Alpha field blank

Now just click on the Tileset in the upper left and draw on stage in the right.

If you can't draw, make sure you have a layer selected. (Selected layer is blue)

Exporting Data

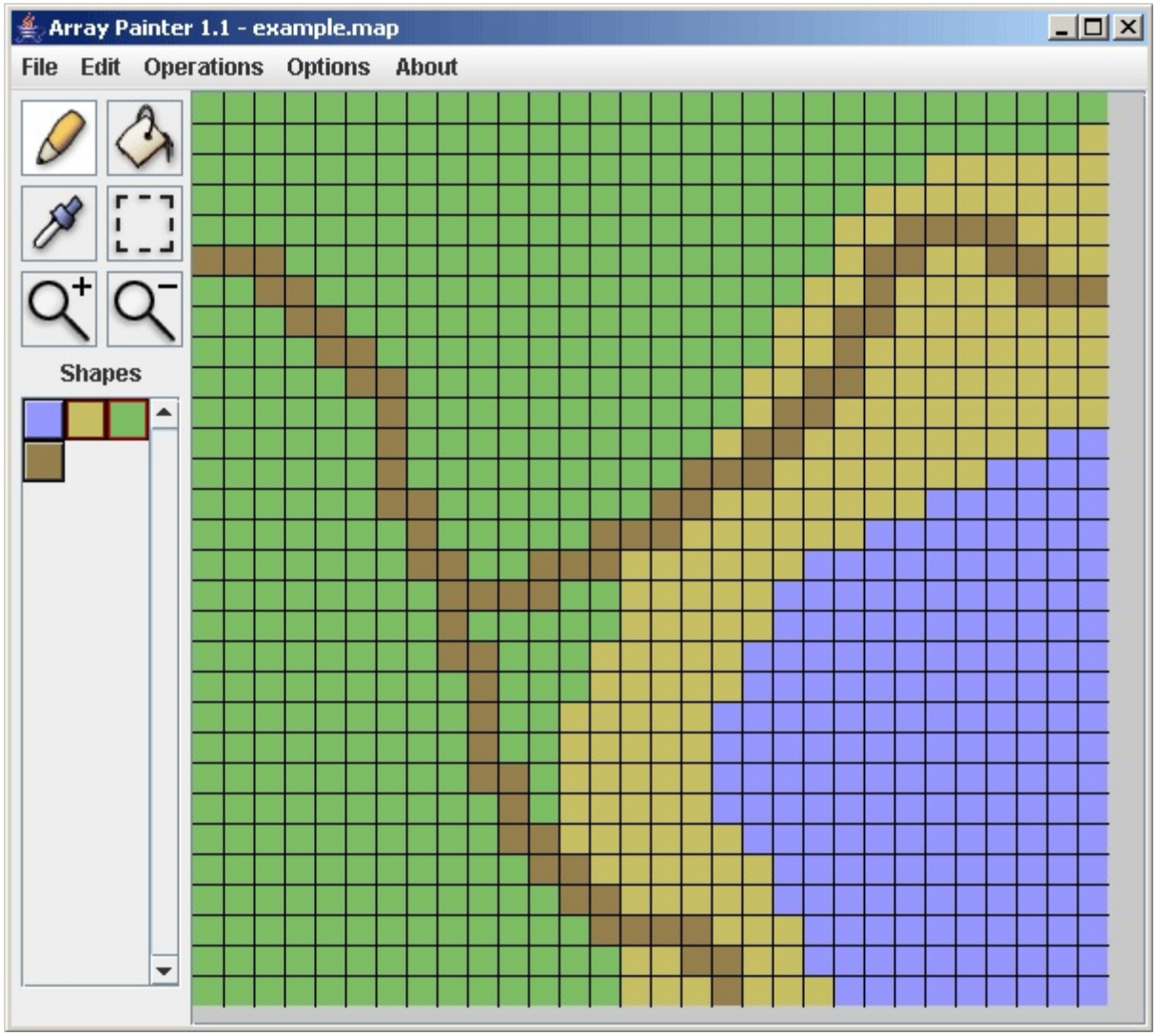
Click **File > Export**, in the popup that follows select:

- Output format: XML
- Format output: yes
- Include src: no

Open up the output file that is downloaded with sublime text. You should be able to reformat this text into a C# array with fairly minimal effort.

Array Painter

By far the simplest tool I've found has been a little Java Applet called Array Painter. I made a local backup of it's JAR file in case it ever goes offline. This is what the application looks like:



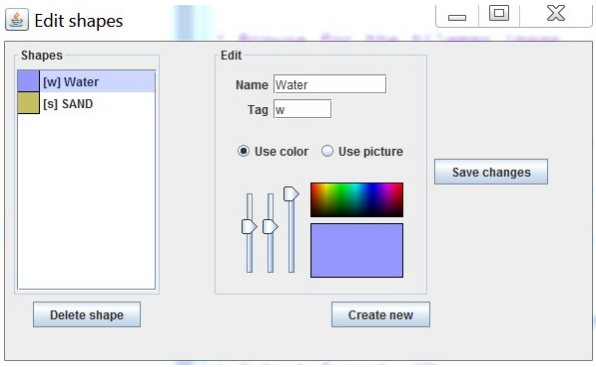
Go to the applications website, download the application. This download will give you a **jnlp** file, which is a shortcut to a Java Web App. (Want to know where the real app is? Open the jnlp file with sublimte text to see what it launches).

There is a catch tough, windows has blocked jnlp applets by default. To launch the application, you must do the following once:

- Open the Java Control Panel
- Click the **Security** tab
- Click the **Edit Site List** button
- Add the following URL to the list
  - http://www.arraypainter.com/webstart/webstart.jnlp

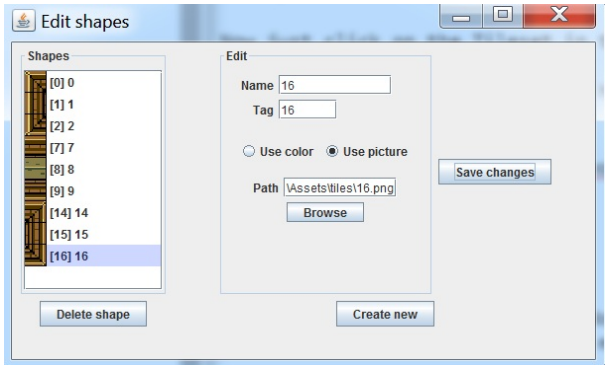
Using Array Painter

By default array painter has two tiles. A blue and a yellow square. Let's import our own tiles. Go to **Options > Edit Shapes**. This brings up the "*Edit Shapes*" window:



Click the **Use picture** check mark and browse for an indevidual tile. Cut out tiles are located in the "*tiles*" folder under assets. Sadly, you can't import a full tile sheet. For the name and tag of the tile, just use it's number. These will be used later to export data. Click the **Create New** button.

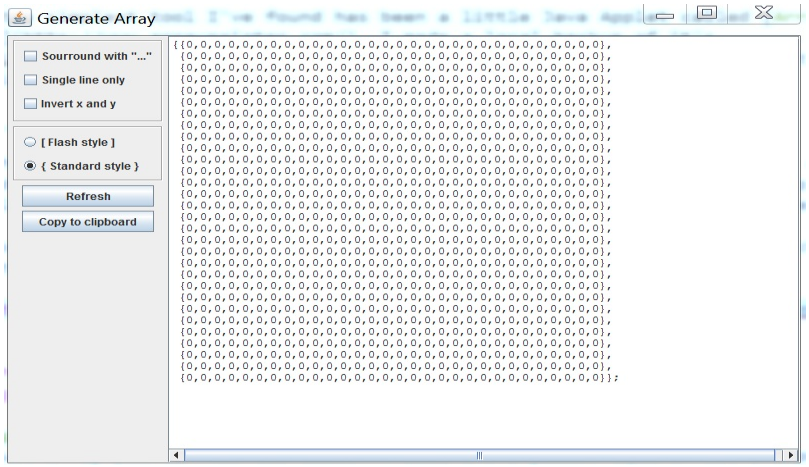
Select the blue square, hit **Delete shape**, select the yellow square and hit **delete shape**. Next add all the tiles you want to use. This is what my window looks like:



Now you can close this window and paint in the main application. If you want to set the map size the option is under **Option > Set Map Size**.

Exporting Data

When you have authored a map you like click **File > Generate Array**. You will be greeted with the following window:



Initially the text array is not populated inside the text box. Make sure you have the same things checked as i do (should be the default) and click the **Refresh** button.

You should be able to copy this array and reformat it into a 2D C# array.

A Better Tool?

All of the tools we've explored in this chapter are pretty nifty, but let's face it none of them are a slam dunk one stop solution for us. This is just the nature of tools, there is no such thing as really generic. If you feel like being a bad-ass you can write your own tool for editing maps. I for one would love to see a more up to date and useful version of Array Painter. It's a project i might actually take on one day.

Making a tile editor is not required, but if you feel up to it it's a fun little side project and a great review of using Windows Forms.

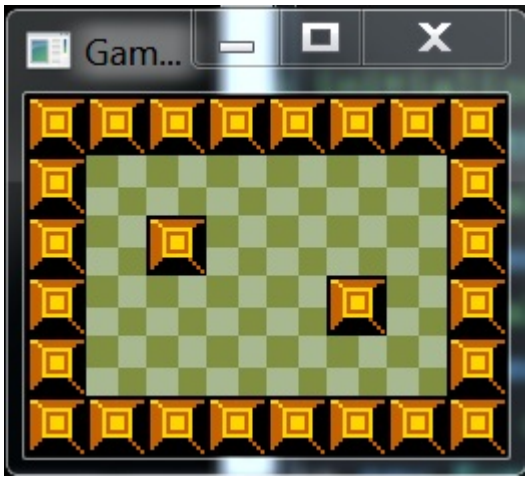
The Hero

Now that we can generate and render maps lets make our game a bit more interesting. We're going to add a hero!

New Project

Create a new project, call it **TheHero**. Make the standard window appear. Don't forget to set the working directory.

I'm going to be continuing from the **Game** class we built in the **Creating Tiles** chapter. You make a copy of that game file (**Copy** don't add a reference, we're going to be modifying this) and get to a point that the game can display this map:



Getting Started

I'm going to get started by giving the hero a spawn position. (Some code below is duplicated from the last example, just to give you an idea of where we are at)

```
namespace TheHero {
    class Game {
        protected Point SpawnTile = new Point(2, 1); // NEW

        protected Tile[][] map = null;
        protected int[][] mapLayout = new int[][] {
            new int[] { 1, 1, 1, 1, 1, 1, 1, 1 },
```

This is going to be the (zero based) tile that our hero spawns at. So, our hero should spawn in the upper left, right above that obstacle.

Now that we have a spawn position, how should we represent the Hero? We could make a bunch of variables in the `Game` class like *heroPosition*, *heroHealth* and *heroSprite* but as you can imagine this will get out of hand soon. A better approach is to make a `Hero` object, this way when we need to add new information to our hero we just add it to said object.

Knowing all of the properties a hero will have (health, position, sprite), the real question is will this information be shared amongst other classes? Enemies perhaps? In good OOP fashion we should account for this. Instead of creating a "Hero" class, let's create a "Character" class that specified characters can then extend.

Let's make a new file, call it **Character.cs** and start implementing our character.

```
using System.Drawing;
using GameFramework;

namespace CreatingTiles {
    class Character {
        public Point Position { get; private set; }
        public int Sprite { get; private set; }

        public Rectangle facingUpFrame1 { get; set; }
        public Rectangle facingDownFrame1 { get; set; }
        public Rectangle facingLeftFrame1 { get; set; }
        public Rectangle facingRightFrame1 { get; set; }

        public Rectangle displaySourceRect { get; set; }

        public Character (string spriteSheet, Point startPos) {
            Position = startPos;
            Sprite = TextureManager.Instance.LoadTexture(spriteSheet);
            displaySourceRect = facingDownFrame1;
        }

        public void Destroy() {
            TextureManager.Instance.UnloadTexture(Sprite);
        }

        public void Render() {
            TextureManager.Instance.Draw(Sprite, Position, 1.0f, displaySourceRect);
        }
    }
}
```

Looking at the above code, each character has a position and a sprite. These are obvious. Then come four rectangles, these are the positions on the sprite sheet to display for when the hero faces each direction. And finally displaySourceRect. This is the active region of the sprite sheet to display. The rest of the methods are pretty straight forward.

Displaying the hero

Let's add some more supporting code to actually display our hero on screen. The following code will all go into **Game.cs**. I included some extra code so you know where we are adding this.

```
namespace TheHero {
    class Game {
        protected Point SpawnTile = new Point(2, 1);
        protected Character hero = null; // NEW
        protected string heroSheet = "Assets/Link.png"; // NEW
```

In the **Initialize** method, create the hero object

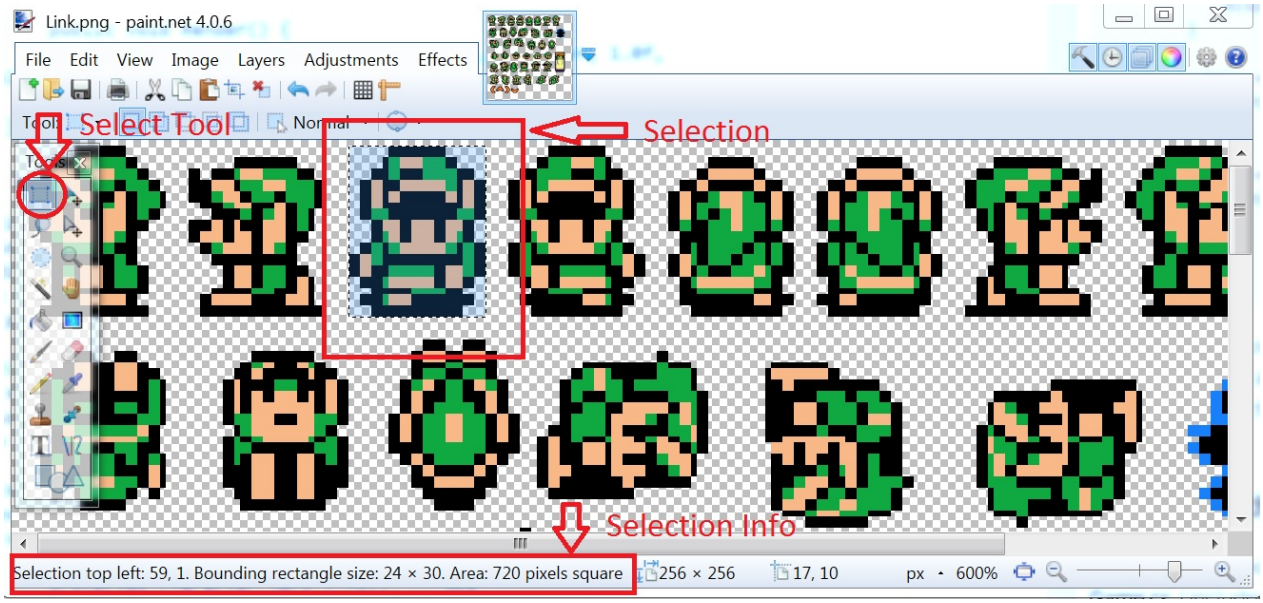
```
public void Initialize() {
    TextureManager.Instance.UseNearestFiltering = true;
    map = GenerateMap(mapLayout, spriteSheets, spriteSources);

    // Everything below this comment is NEW
    hero = new Character(heroSheet, new Point(SpawnTile.X * 30, SpawnTile.Y * 30));

    hero.facingLeftFrame1 = new Rectangle(1, 1, 26, 30);
    hero.facingDownFrame1 = new Rectangle(59, 1, 24, 30);
    hero.facingUpFrame1 = new Rectangle(115, 3, 22, 28);
    hero.facingRightFrame1 = new Rectangle(195, 1, 26, 30);
```

```
    hero.displaySourceRect = hero.facingDownFrame1;
}
```

Why is SpawnTile.X and Y multiplied by 30? 30 happens to be the width / height of each of our tiles. If you remember, SpawnTile is in tile coordinates, but the render function of the hero works in pixel coordinates. We have to convert that second argument to pixels, so (tile Index * tile Size)

Where did those magic numbers for each rectangle frame come from? Photoshop! (Well, i actually use Paint.Net). I just made rectangle selections on the sprite sheet and copy / pasted the numbers into code.

If you install Paint.Net and want to get your own rectangle regions, this is what you should be looking at:



Anyway, we created a hero object, if you recall the Character class holds on to a reference of the sprite sheet it was created with. We must call it's **Destroy** method in the Program's **Shutdown**

```
public void Shutdown() {
    for (int h = 0; h < map.Length; ++h) {
        for (int w = 0; w < map[h].Length; ++w) {
            map[h][w].Destroy();
        }
    }
    hero.Destroy(); // NEW
}
```

Now the last thing that's left to do is to call the Character classes **Render** method.

```
public void Render() {
    for (int h = 0; h < map.Length; ++h) {
        for (int w = 0; w < map[h].Length; ++w) {
            map[h][w].Render();
        }
    }
    hero.Render(); // NEW
}
```

Go ahead, try it out. Running your program, the application should look something (exactly) like this:


Game Name

On Your Own

The Character class we created in **The Hero** section works ok for what we need, but it's very static. By this i mean there are only ever 4 images, and even then the one that is to be displayed needs to be hard coded. Let's fix this. In this section we're going to make the hero more dynamic.

Think about it

How would you solve this problem? I'm going to present my solution below, but it's by no means the best or only way to tackle the problem. Before reading my answer think about how you would solve this problem. Maybe try to implement it! If you do come up with your own implementation, i want to see it! You can also use my implementation as a guide, and not follow it verbatim. Or you can do what i do, this is a learning experience!

New Project

Create a new project, call it **HeroPractice**. We will be continuing from the **The Hero** section, make sure that the new project is up to date / par with that.

Changing the hero

Any time you hear the word dynamic, you should probably start thinking about a Dictionary (Hash table). Dictionary's allow us to get super customizable by having dynamic string keys. We're going to re-do the **Character** class with dictionary support.

First, remove the following variables from the character class:

- facingDownFrame1
- facingLeftFrame1
- facingRightFrame1
- displaySourceRect

Add a new dictionary, the key should be a string, the value should be a rectangle. I would call this dictionary **spriteSources**, but the name is up to you. Also, add a string. Again what you call the string is up to you, i call mine **currentSprite**.

The idea here is pretty simple, we will be able to add any number of sprites to the **Character** object, and reference them by strings thanks to our new dictionary. Only one of these sprites will display at a time, and it will be the one that has the same key as **currenSprite**.

We don't want these variables to be just randomly set, make sure they are private. Next, make the following helper metod in the **Character** class:

```
public void AddSprite(string name, Rectangle source) {
```

This method should add a new entry into the **spriteSources** dictionary. If **currentSprite** has not been set yet (If it's null or empty), set **currentSprite** to be the **name** argument. This way if at least one sprite is added it will be displayed by default no matter what.

Also add this helper method:

```
public void SetSprite(string name) {
```

SetSprite should check if the **name** argument exists as a key inside the **spriteSources** dictionary. If it does set the **currentSprite** variable equal to the **name** argument. If not, do nothing.

Last, update your **Render** function. It should display a sprite out of the **spriteSources** dictionary. IF the **currentSprite** string has not been set (If it's null or empty), early out and draw nothing (as the dictionary access would trigger an NRE).

End result

The **Initialize** method of my **Progra.cs** looks like this:

```
public void Initialize() {
    TextureManager.Instance.UseNearestFiltering = true;
    map = GenerateMap(mapLayout, spriteSheets, spriteSources);
    hero = new Character(heroSheet, new Point(SpawnTile.X * 30, SpawnTile.Y * 30));

    hero.AddSprite("Down", new Rectangle(59, 1, 24, 30));
    hero.AddSprite("Up", new Rectangle(115, 3, 22, 28));
    hero.AddSprite("Left", new Rectangle(1, 1, 26, 30));
    hero.AddSprite("Right", new Rectangle(195, 1, 26, 30));
}
```

And i added this to **Update** so i could test:

```
public void Update(float dt) {
    InputManager i = InputManager.Instance; // Local reference to input manager
    // ^ I only use the reference to save on space below

    if (i.KeyDown(OpenTK.Input.Key.Left) || i.KeyDown(OpenTK.Input.Key.A)) {
        hero.SetSprite("Left");
    }
    if (i.KeyDown(OpenTK.Input.Key.Right) || i.KeyDown(OpenTK.Input.Key.D)) {
        hero.SetSprite("Right");
    }
    if (i.KeyDown(OpenTK.Input.Key.Up) || i.KeyDown(OpenTK.Input.Key.W)) {
        hero.SetSprite("Up");
    }
    if (i.KeyDown(OpenTK.Input.Key.Down) || i.KeyDown(OpenTK.Input.Key.S)) {
        hero.SetSprite("Down");
    }
}
```

Even tough link still doesn't move the arrow keys (and WASD) now make him turn in different directions :)

Keys To Move

We have a hero on the screen! But he's not very mighty... He can't even move! Let's fix that. In this section we're going to add the ability to walk around the map.

New Project

Make a new project, call it **KeysToMove**, we're going to be starting off from the **On Your Own** section of **The Hero**. Meaning your new project should pop up a window, the window has a room, link is in the room and pressing the directional buttons moves link around.

Player character

In **The Hero** section, we made a `Character` class. This class contains the bare minimum information needed to display a character. We could add code to this class for handling input and moving around but that would get messy, after all not every `Character` instance will be controlled by the player! We want enemies to be `Character`s as well.

The best way to solve this delima is to create a new class, we're going to call `PlayerCharacter`, this new class will extend `Character` and add player specific functionality to it! Let's get started. Make a new file, call it **PlayerCharacter.cs** and add the following code:

```
using System.Drawing; // For Point
using TheHero; // For Character

namespace KeysToMove {
    class PlayerCharacter : Character {

        public PlayerCharacter(string spriteSheet, Point startPos) : base(spriteSheet, startPos) {
            // ^ The Constructor which take a string and a Point of the base class (Character)
            // is called using the : notation.
        }
    }
}
```

I don't know if we have used constructor chaining before. Constructor chaining is when a base class does not contain a default constructor, so the child class must call it's parents non-default constructor. This is done by adding `: base(/*args*/)` after the consturctor. In our example, we pass the arguments from the `PlayerCharacter` constructor to the `Character` constructor (refered to as `base`).

When chaining constructors you don't always have to forward arguments, the following code would have also been perfectly valid:

```
public PlayerCharacter() : base("Default string", new Point(-2, 5)) {
```

But we want the classes to work the same, so instead we take two arguments and forward them to the base class.

Updating the character

The `PlayerCharacter` and `Character` classes at this point are interchangable. Go into your **Game.cs** file and change this line:

```
protected Character hero = null;
```

To this:

```
protected PlayerCharacter hero = null;
```

Also find this line:

```
hero = new Character(heroSheet, new Point(SpawnTile.X * 30, SpawnTile.Y * 30));
```

And change it to this:

```
hero = new PlayerCharacter(heroSheet, new Point(SpawnTile.X * 30, SpawnTile.Y * 30));
```

Beacuse the `PlayerCharacter` is specific to the hero, it's safe to put hero specific things in there. Take this bit of code from **Game.cs**:

```
hero.AddSprite("Down",new Rectangle(59,1,24,30));
hero.AddSprite("Up", new Rectangle(115,3,22,28));
hero.AddSprite("Left",new Rectangle(1,1,26,30));
hero.AddSprite("Right", new Rectangle(195, 1, 26, 30));
hero.SetSprite("Down");
```

And move it into the constructor of of the `PlayerCharacter` class:

```
public PlayerCharacter(string spriteSheet, Point startPos) : base(spriteSheet, startPos) {
    // ^ The Constructor which take a string and a Point of the base class (Character)
    // is called using the : notation.

    AddSprite("Down", new Rectangle(59, 1, 24, 30));
    AddSprite("Up", new Rectangle(115, 3, 22, 28));
    AddSprite("Left", new Rectangle(1, 1, 26, 30));
    AddSprite("Right", new Rectangle(195, 1, 26, 30));
    SetSprite("Down");
}
```

At this point you should be able to run the game. The game should display, and link should rotate when you press keys like nothing has happened.

Moving

Let's add code to actually move the character! In order to move the character we have to do a little bit of house cleaning first.

Currently we keep track of the `Character` position in a **Point**. This won't work, in order to move around the screen we need to store values as floating points. Find the following line in **Character.cs**

```
public Point Position { get; set; }
```

and change it to

```
public PointF Position { get; set; }
```

If you try to compile now, the compiler will throw an error. This is because our render method is trying to pass a **PointF** variable when a **Point** is expected. Still in **Character.cs**, find this bit of code:

```
public void Render() {
    TextureManager.Instance.Draw(Sprite, Position, 1.0f,spriteSource[currentSprite]);
}
```

and change it to:

```
public void Render() {
    TextureManager.Instance.Draw(Sprite, new Point((int)Position.X, (int)Position.Y), 1.0f,spriteSource[currentSprite]);
}
```

And you should be able to compile again! Back in **PlayerCharacter.cs** lets add a speed variable. This varaible is going to determine how fast the player moves. I'm going to set it to 90.0f, that is, move 90 pixels every second or 3 tiles in one second.

```
namespace KeysToMove {
    class PlayerCharacter : Character {
        float speed = 90.0f; // NEW

        public PlayerCharacter(string spriteSheet, Point startPos) : base(spriteSheet, startPos) {
```

Next let's add an `Update` method to the `PlayerCharacter` class. It should take one argument, `deltaTime` .

```
public void Update(float deltaTime) {
    // TODO: Add move code
}
```

This method is going to be responsible for checking Keyboard Input. When the appropriate keys are pressed it's going to update the inherited `Position` property to make the character move (Remember, the character is rendered at `Position`) Let's fill in the missing code:

```
public void Update(float deltaTime) {
    InputManager i = InputManager.Instance;

    PointF positionCpy = Position; // Don't forget to change to PointF in parent class
    // The Position getter is inherited from the Character class.
    // We simply store a COPY of it here, so we can modify the
    // X and Y properties indevidually. We also make

    if (i.KeyDown(OpenTK.Input.Key.Left) || i.KeyDown(OpenTK.Input.Key.A)) {
        positionCpy.X -= speed * deltaTime;
    }
    else if (i.KeyDown(OpenTK.Input.Key.Right) || i.KeyDown(OpenTK.Input.Key.D)) {
        positionCpy.X += speed * deltaTime;
    }
```

```
    if (i.KeyDown(OpenTK.Input.Key.Up) || i.KeyDown(OpenTK.Input.Key.W)) {
        positionCpy.Y -= speed * deltaTime;
    }
    else if (i.KeyDown(OpenTK.Input.Key.Down) || i.KeyDown(OpenTK.Input.Key.S)) {
        positionCpy.Y += speed * deltaTime;
    }

    Position = positionCpy; // Move the copy we made back into the Postion variable
}
```

Running the game now, pressing the arrow keys.... Nothing moves! What gives? Well, we added an `Update` method to the `PlayerCharacter` class, but we are not calling this method anywhere! Let's go into **Game.cs** and add this `Update` call to the `Game` s update method:

```
public void Update(float dt) {
    InputManager i = InputManager.Instance; //local ref to input manager
    //using i just saves time
    if (i.KeyDown(OpenTK.Input.Key.Left) || i.KeyDown(OpenTK.Input.Key.A)) {
        hero.SetSprite("Left");
    }
    if (i.KeyDown(OpenTK.Input.Key.Right) || i.KeyDown(OpenTK.Input.Key.D)){
        hero.SetSprite("Right");
    }
    if (i.KeyDown(OpenTK.Input.Key.Up) || i.KeyDown(OpenTK.Input.Key.W)) {
        hero.SetSprite("Up");
    }
    if (i.KeyDown(OpenTK.Input.Key.Down) || i.KeyDown(OpenTK.Input.Key.S)) {
        hero.SetSprite("Down");
    }

    hero.Update(dt);
}
```

Run the game now, and TADA! Your hero is moving! He's not colliding with any walls.... But we can fix that later. For now what matters is that the hero moved!

Cleanup

One thing you may have noticed is that there is a lot of similar code between the **Update** method of `Game` and `PlayerCharacter`. Whenever you have a lot of similar code, chances are it's time to refactor (clean-up).

Let's move all of the `SetSprite` methods for key input into the `PlayerCharacter` class. Updating this code, the `Update` method of **Game.cs** should now look like this:

```
public void Update(float dt) {
    hero.Update(dt);
}
```

And the `Update` method of **PlayerCharacter.cs** should look like this:

```
public void Update(float deltaTime) {
    InputManager i = InputManager.Instance;

    PointF positionCpy = Position; // Don't forget to change to PointF in parent class
    // The Position getter is inherited from the Character class.
    // We simply store a COPY of it here, so we can modify the
    // X and Y properties indevidually. We also make

    if (i.KeyDown(OpenTK.Input.Key.Left) || i.KeyDown(OpenTK.Input.Key.A)) {
        positionCpy.X -= speed * deltaTime;
```

```
            SetSprite("Left");
        }
        else if (i.KeyDown(OpenTK.Input.Key.Right) || i.KeyDown(OpenTK.Input.Key.D)) {
            positionCpy.X += speed * deltaTime;
            SetSprite("Right");
        }

        if (i.KeyDown(OpenTK.Input.Key.Up) || i.KeyDown(OpenTK.Input.Key.W)) {
            positionCpy.Y -= speed * deltaTime;
            SetSprite("Up");
        }
        else if (i.KeyDown(OpenTK.Input.Key.Down) || i.KeyDown(OpenTK.Input.Key.S)) {
            positionCpy.Y += speed * deltaTime;
            SetSprite("Down");
        }

        Position = positionCpy; // Move the copy we made back into the Postion variable
    }
```

The `SetSprite` method is callable because we inherited it from the `Character` class. Sometimes it can get a little hard to follow what is and isn't accessable. If you are feeling brave, try using draw.io or ArgoUML (i suggest draw.io) to make a UML diagram the current project. If you end up making the UML check the exported PNG image into your repo.

On Your Own

We now have a way to make the character walk on screen! Huzza! One small problem we're having here is that the hero is just kind of sliding around on the screen! While animations are not required to make a decent game demo, they help. A lot. So in this section we're going to add animations to our Hero!

New Project

Let's make a new project, call it **MovingPractice** and get this project up to par with the **KeysToMove** section of the writeup.

Getting started

We will have to change code in the `Character` class, and add new code to the `PlayerCharacter` class to support animations. Keep in mind, i'm only going to give you guidance to how i would implement animation. You can follow my lead, or come up with your own way of doing things. There is no right or wrong way. So long as it looks right and runs fast it's good!

The basic concept is simple, when we add a sprite i don't want to just add a single rectangle. I want to add a variable number of rectangles to the sprite name. I also want to add an animation flag. If animation is false, show only the current (first?) rectangle of the sprite name. If animation is true, flip trough all of the rectangles in the sprite name at a set framerate.

Refactoring Character

First things first, let's refactor the `Character` class. In **Character.cs** we have the following function:

```
public void AddSprite(string name,Rectangle source) {
```

I want to change this function to support multiple rectangles. Instead of taking a list, take a params array:

```
public void AddSprite(string name, params Rectangle[] source) {
```

This will allow the function to be called with a variable number of rectangles seperated by only commas. Of course this is going to break a few things. This bit of code is no longer valid:

```
private Dictionary<string, Rectangle> spriteSources = null;
```

That however is an easy fix, just change the value type to be a Rectangle array

```
private Dictionary<string, Rectangle[]> spriteSources = null;
```

Now the `AddSprite` method works again, but the `Render` method is broken!

There are a few ways to fix this. You could always render frame 0, but that's a bad idea! You could add an integer to

the `Character` class and call it `currentFrame`. This int would represent what frame to display. Adding an integer sounds like a good idea, it's certainly intuitive. But it does have a serious flaw: not every animation is going to have the same number of frames.

Because each animation can have a variable number of frames, we need to add house-keeping. Whenever a new animation is played, the currentFrame has to be reset to 0 to avoid going out of bounds. But then switching back to the old animation, you can't just resume where you left off. This may or may not be the behaviour you want. Is there an alternative?

Yes, yes there is. You could keep a parallel array. One who's key is a `string` and value is an `int`. So it would know the walk animation is on frame 5, while the idle animation is on frame 2. In this case you would also have to add a new string, and call it currentAnimation. This would be used to index into the dictionary. Wow, thats a lot of house keeping! I'm already getting confused!

So, how should we handle this?!?!?! Well, experience tells me having an integer called `currentFrame` is the best approach. It's easy to read and requires the least amount of house keeping. It also makes the most sense. So i'm just going to add a new integer to the `Character` class:

```
namespace TheHero {
    class Character {
        public PointF Position { get; protected set; }
        public int Sprite { get; private set; }

        public Dictionary<string, Rectangle> spriteSource { get; private set; }
        public string currentSprite { get; private set; }
        protected int currentFrame = 0; // NEW
        // ^ protected means any children of this class can modify the variable
```

and let's not forget to fix the `Render` method:

```
public void Render() {
    TextureManager.Instance.Draw(Sprite, new Point((int)Position.X, (int)Position.Y), 1.0f, spriteSources[currentSprite][cu
}
```

Running the game at this point, you should be able to walk around with link. From the players point of view nothing has changed! But from the programmers point of view, we have laid the groundwork for animating the character.

Refactoring Player Character

At this point, whatever the `currentFrame` variable is set to will be displayed. All you need to do is add some more frames. I'll spare you having to look up the frame positions manually, here are the walking frames for link:

```
public PlayerCharacter(string spriteSheet, Point startPos) : base(spriteSheet, startPos) {
    // ^ The Constructor which take a string and a Point of the base class (Character)
    // is called using the : notation.

    AddSprite("Down", new Rectangle(59, 1, 24, 30), new Rectangle(87, 1, 24, 30));
    AddSprite("Up", new Rectangle(115, 3, 22, 28), new Rectangle(141, 3, 22, 28));
    AddSprite("Left", new Rectangle(1, 1, 26, 30), new Rectangle(31, 1, 26, 31));
    AddSprite("Right", new Rectangle(195, 1, 26, 30), new Rectangle(167, 1, 26, 29));
}
```

Animation is as simple as setting `currentFrame`. I'll let you figure out how that should be done. I'll include some hints

as to how i did it below.

Animation, my method

I added three variables to the `PlayerCharacter` class to support animation:

- bool animating = false
- float animFPS = 1.0f / 9.0f;
- float timeTimer = 0.0f

The **animating** boolean is self explanatory. If it's set to true the animation is playing. If it's set to false then no animation plays. This is how i make link no animate if no button is pressed. That boolean is only true while one of the movement buttons is held down.

The **animFPS** float isn't as straight forward. Link animates at 9 frames / second. This means in one second you see 9 frames of animation. So, why *1.0f / 9.0f*? Simple, we are going to have a counter. That counter is going to count upwards from 0. If we want to display 9 frames every second, that means we want to show ONE new frame every 0.11 seconds. *1.0f / 9.0f* equals 0.11.

The **timeTimer** float is pretty simple. This is the counter variable that counts upwards. If **timeTimer** is greater than **animFPS** then we will flip one frame!

So how is this all implemented? In each of the if statements that checks if a key is down (and moves the position and sets the sprite if it is) i also set animating to true. After the if blocks that handle input i check if animating is true. If it is, i add deltaTime to animTimer.

Still inside the `if(animating) {` block, i put anoter nested if. If the animTimer is greater than animFPS, i add one to the currentFrame. I also subtract animFPS from animTimer, this gives a smooth animation. Time for yet another nested if, if the currentFrame is greater than the Length of the current spriteSources array then i reset currentFrame to 0 (to keep from going out of bounds).

Finally outside of any if statements, after the `if(animating) {` block i set animating back to false. It's the default state of the animating variable. This way, if no key is pressed no animation happens.

Check in

Skype or text me if you have any questions. Turning the `PlayerCharacter` class into a flip book is not easy, but i think you can do it!

If you finish this section skype or text me. I want to look over your work before you move on to the next section!

Hit The Wall

Having a hero who can walk is fun, but it would be even more fun if the hero could hit the brick wall... With his face! Handling collisions with walls is not that hard. We've done this in **WinFormGames** before. Given this situation:

We first want to find the rectangle for both link (outlined blue) and the obstacle (outlined green):

Using these two rectangles we want to see if they intersect. To do this we look for an intersection rectangle (outlined in red):

If the intersection(red) rectangle has an area greater than 0 (That is, if a collision has happened) we must resolve the collision. We do this by moving the character to the left, right up or down by the width or height of the intersection.

New Project

Let's make a new project, call it **HitTheWall** and get this project up to par with the **MovingPractice** section of the writeup. We're going to work from here.

Intersection

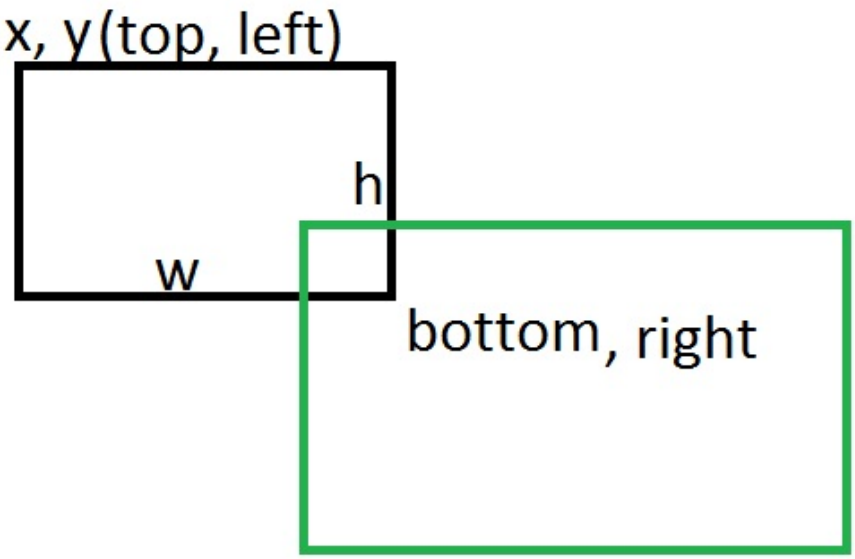
From the above description it becomes clear that we need to be able to get an intersection rectangle between two rectangles. We're going to add this ability to a common helper class. If this method is looking alien try to draw out all of the intersection cases on paper and follow it trough for each.

Let's make a new file, call it **Intersections.cs** and add the follwoing class to it:

```
using System.Drawing;

namespace HitTheWall {
    class Intersections {
        public static Rectangle Rect(Rectangle a, Rectangle b) {
            Rectangle result = new Rectangle(0,0,0,0);

            // Do they even intersect?
            if (a.Left < b.Right && a.Right > b.Left && a.Top < b.Bottom && a.Bottom > b.Top) {
                // They intersect, let's get the intersection

                result.X = System.Math.Max(a.Left, b.Left);
                result.Y = System.Math.Max(a.Top, b.Top);

                int right = System.Math.Min(a.Right, b.Right);
                int bottom = System.Math.Min(a.Bottom, b.Bottom);

                // Originally i assumed there might be an error in this function, because what if
                // right is less than result.X? After reviewing the above math on paper i realize
                // that these concerns where ungrounded, and that right will ALWAYS be greater!

                result.Width = right - result.X;
```

```
                result.Height = bottom - result.Y;
            }

            return result;
        }

    }
}
```

If you end up drawing and labeling these, remember X / Y is the same as Top / Left. But Width / Height are different from Bottom / Right:



Character Refactor

We're going to need to know some information about the character that we don't already know. Nameley we will need to know the center point of the character, and the bounding rectangle of the character. Remember, these can be different for every frame.

We're going to add some getters for the missing information, remember this is going in the `Character` class, not *PlayerCharacter*. You should have enough information to figure these attributes out. Implement these getters:

```
public PointF Center {
    get {
        // TODO
    }
}

public Rectangle Rect {
    get {
        // TODO
    }
}
```

If you have any questions about the implementation of these, don't hesitate to ask. While we're in the `Character` let's update the **Render** function with some debug code.

```
public void Render() {
```

```
    GraphicsManager.Instance.DrawRect(Rect, Color.Red);
    TextureManager.Instance.Draw(Sprite, new Point((int)Position.X,(int)Position.Y), 1.0f, SpriteSource[currentSprite][curr
    Rectangle center = new Rectangle((int)Center.X - 5, (int)Center.Y - 5, 10, 10);
    GraphicsManager.Instance.DrawRect(center, Color.Yellow);
}
```

Adding the debug code should draw link with a red square the size of his bounding box behind him and with a yellow box at his sprites center:

Let's also change the characters position from a getter to a variable. Find this line in **Character.cs**

```
public PointF Position { get; set; }
```

And change it to

```
public PointF Position = new PointF(0.0f, 0.0f);
```

Now that Position is a variable and not a getter we can modify the X and Y properties of the value directly. Open up **PlayerCharacter.cs**

Find and remove these two lines

```
PointF positionCpy = Position;
// ...
Position = positionCpy;
```

And replace every instance of `positionCpy` with `Position`

We're going to change one last thing about the `Character` class. Back in **Character.cs** add the following code to the class

```
public PointF[] Corners {
    get {
        float w = SpriteSource[currentSprite][currentFrame].Width;
        float h = SpriteSource[currentSprite][currentFrame].Height;
        return new PointF[] {
            new PointF(Position.X, Position.Y),            // Top Left
            new PointF(Position.X + w, Position.Y),        // Top Right
```

```
            new PointF(Position.X, Position.Y + h),         // Bottom Left
            new PointF(Position.X + w, Position.Y + h)      // Bottom Right
        };
    }
}

public static readonly int CORNER_TOP_LEFT = 0;
public static readonly int CORNER_TOP_RIGHT = 1;
public static readonly int CORNER_BOTTOM_LEFT = 2;
public static readonly int CORNER_BOTTOM_RIGHT = 3;
```

The `Corners` accessor will return an array of 4 points. Each point is one of the corners of the player sprite. After that we have some named constants. These constants will allow us to access the `Corners` array by name isntead of having to remember hard coded numbers. Like so:

```
Rectangle topRight = heroChar.Corners[2]; // Flaky, we don't want to do this
Rectangle topLeft = heroChar.Corners[Hero.CORNER_TOP_LEFT]; // Sexy, ALWAYS DO THIS!
```

Let's visualize what this looks like. Change the `Render` function in **Character.cs** to:

```
public void Render() {
    GraphicsManager.Instance.DrawRect(Rect, Color.Red);
    TextureManager.Instance.Draw(Sprite, new Point((int)Position.X, (int)Position.Y), 1.0f, SpriteSource[currentSprite][cu
    Rectangle center = new Rectangle((int)Center.X - 5, (int)Center.Y - 5, 10, 10);
    GraphicsManager.Instance.DrawRect(center, Color.Yellow);
    foreach(PointF corner in Corners) {
        Rectangle rect = new Rectangle((int)corner.X - 5, (int)corner.Y - 5, 10, 10);
        GraphicsManager.Instance.DrawRect(rect, Color.Green);
    }
}
```

Your code at this point should look like this:

Resolving collisions

Nine times out of ten resolving collisions is going to go in the same function that handles input. (What's the case when this isn't how it happens? When solving continous integration systems in a Physics engine). In order to resolve collisions we need to know a few things.

- Move the character
- Get a list of obstacles the character can hit.

- Check if character has hit obstacle
  - If so, undo the move action (clamp to obstacle)

Pretty simple. We basically move, check if the move is valid and undo the move if it's not.

If you are being clever and getting obstacles relative to theplayer, it's very important that you move the character first, then get the list of obstacles. This is because the list of obstacles will usually change based on the characters position. If the character moves you need the latest, most up to date obstacles. We're going to be clever and do this :)

The other option is to brute force the deal, this means checking and trying to resolve the characters collision against every collidable object all the time. This method might work for smaller games, but the list of collisions to check will quickly grow out of hand. Most of the time brute force will simply not be an option due to performance reasons. We will **not** be doing brute force.

Game Refactor

Looking at the above list of how to resolve collisions we already know how to move the character. We however don't know how to get a list of obstacles that the character can collide with. This functionality is going to be given to **Game.cs**.

We're not going to call any new functions in Game. Instead we're going to add helper functions that will be called from inside of `PlayerCharacter` we can do this because `Game` is a singleton. We're going to write two helper functions `GetTile` and `GetTileRect` both of them are going to take a `PointF` as an argument.

How are these new functions going to be used? Inside of `PlayerCharacter` we're going to get the tile for one of the corners of the character. If the tile is NOT walkable, we're going to get it's rectangle (again, using one of the corner points). We will try to resolve collisions against this rectangle.

With that, add these functions to **Game.cs**

```
public Tile GetTile(PointF pixelPoint) {
    // TODO: Get the x and y indices of the tile from pixelPoint
    // Return the appropriate tile
    return null; // Just so this compiles
}

public Rectangle GetTileRect(PointF pixelPoint) {
    // TODO: Convert pixel point into a Rectangle that is on the tile grid
    // Hint, xTile * tileSize = Grid X Position
    // return the resulting rectangle
    return new Rectangle(); // Just so this compiles
}
```

Try to fill those functions in. Let's also change how the rendering of the game works by adding more debug symbols! Change the `Render` function in **Game.cs** to this:

```
public void Render() {
    for (int h = 0; h < map.Length; h++) {
        for (int w = 0; w < map[h].Length; w++) {
            map[h][w].Render();
        }
    }

    foreach (PointF corner in hero.Corners) {
        if (GetTile(corner).Walkable) {
            GraphicsManager.Instance.DrawRect(GetTileRect(corner), Color.Blue);
        }
```

```
            else {
                GraphicsManager.Instance.DrawRect(GetTileRect(corner), Color.Violet);
            }
        }

        hero.Render();
    }
```

This render function actually demonstrates HOW we're going to use the `GetTile` and `GetTileRect` functions. Any tile that one of the player corners falls on is colored Blue or Violet. Blue tiles are walkable, Violet ones are not. When it comes time to Resolve Collisions, we only have to check the violet tiles. Checking so few tiles will make our game super performant.

Running your game, you should get something like the below screenshot. The blue / violet squares might not always be evenly distributed. That is at some points you might only see 2 tiles get colored. Walk around, you should see a few patterns.

Refactoring PlayerCharacter

Now comes the fun part. We're actually going to resolve collisions! **BUT FIRST**, lets do some house keeping! Let's make it so that the character can walk both up/down AND left/right at the same time. Inside **PLayerCharacter.cs** find and change:

```
else if (i.KeyDown(OpenTK.Input.Key.W) || i.KeyDown(OpenTK.Input.Key.Up)) {
```

to

```
if (i.KeyDown(OpenTK.Input.Key.W) || i.KeyDown(OpenTK.Input.Key.Up)) {
```

The way we do animations now is good, but an animation frame might change the Width / Height of our sprite. If the width or height is going to change, we want that to happen sooner, rather than later. Inside **PlayerCharacter.cs** we have a variable `bool animating = false;` Remove that.

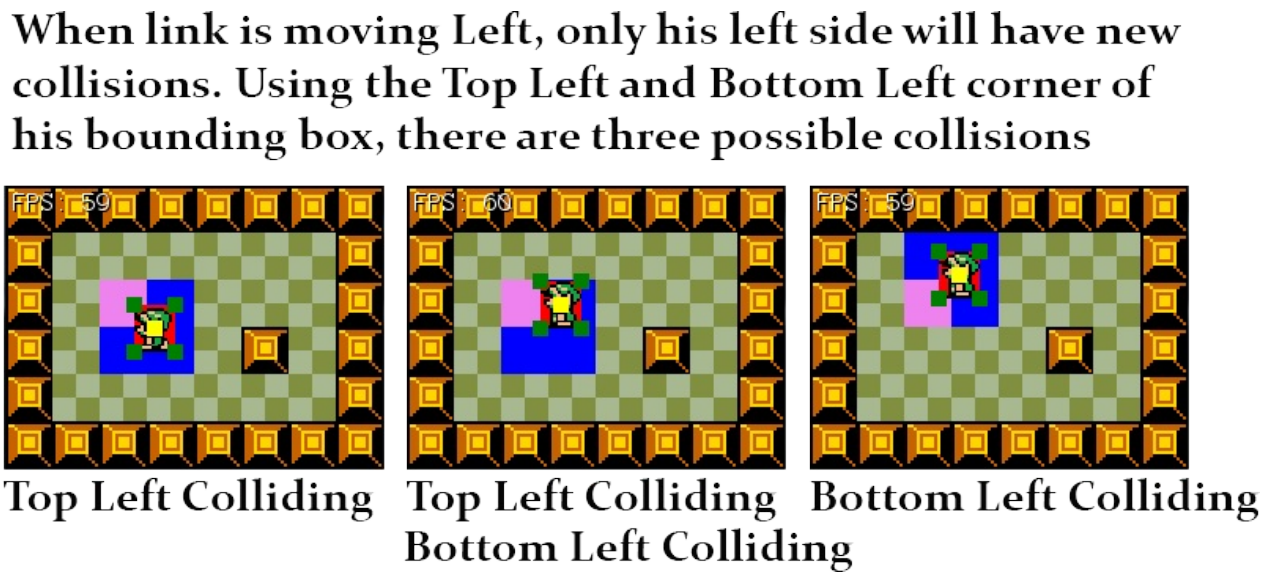
Find the animation code at the bottom of `Update` and remove it. We're going to refactor it into an `Animate` function, this is what it's going to look like:

```
protected void Animate(float deltaTime) {
    animTimer += deltaTime;
```

```
        if (animTimer > animFPS) {
            currentFrame += 1;
            animTimer -= animFPS;
            if (currentFrame > SpriteSource[currentSprite].Length - 1) {
                currentFrame = 0;
            }
        }
    }
```

Again, inside of the `Update` function replace everywhere aimation is set to true `animating = true;` to a call to Animate `Animate(deltaTime`.

Run the game it should at this point run as expected. Now it's time to resolve some collisions! The key to efficient collision resolution is to check as few things as possible. As long as the character is the same size or smaller than the tiles there can be at most two collisions at a time:



When link is moving in one direction, only his corners in that direction need to be checked. And it's not even his corners, it's the tile his corners fall on. In the middle example above, both the top and bottom corners fall on the same tile, so only one is checked. Here is a breakdown of which corners to check:

- Moving Left
  - Check Top Left Corner
  - Check Bottom Left Corner
- Moving Right
  - Check Top Right Corner
  - Check Bottom Right Corner
- Moving Up
  - Check Top Left Corner
  - Check Top Right Corner
- Moving Down
  - Check Bottom Left Corner
  - Check Bottom Right Corner

How do we actually go about checking these? Let's walk trough adding collision resolution when moving left. First, locate the code that moves link left:

```
if (i.KeyDown(OpenTK.Input.Key.A) || i.KeyDown(OpenTK.Input.Key.Left)) {
    SetSprite("Left");
```

```
    Animate(deltaTime);
    positionCpy.X -= speed * deltaTime;

    // Add collision resolution
}
```

- To add collision resolution, first lets check the TopLeftCorner.
- Use the Game class to get the tile at the top left corners Point
- Check to see if the tile is walkable. IF NOT:
  - Get the intersection rectangle of the player and the tile
  - An intersection happens if the intersection rect has an area > 0
  - Check if intersect.W * intersect.H > 0, IF IT IS:
    - Clamp player X to intersection rectangle right

Let's see what this would look like in code:

```
if (!Game.Instance.GetTile(Corners[CORNER_TOP_LEFT]).Walkable) {
    Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_LEFT]));
    if (intersection.Width * intersection.Height > 0) { // W * H == 0 if NO intersection happened!
        Position.X = intersection.Right;
    }
}
```

We don't need to fully qualify `Rect`, `Corner` or `CORNER_TOP_LEFT` because we inherited them from `Character`. We can access the helper functions of `Game` trough it's singleton instance.

That takes care of the top left collision of moving left, we need to also check bottom left. It's going to more or less be the same code. Here is the entire key checker part of the code:

```
if (i.KeyDown(OpenTK.Input.Key.A) || i.KeyDown(OpenTK.Input.Key.Left)) {
    SetSprite("Left");
    Animate(deltaTime);
    Position.X -= speed * deltaTime;

    if (!Game.Instance.GetTile(Corners[CORNER_TOP_LEFT]).Walkable) {
        Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_LEFT]));
        if (intersection.Width * intersection.Height > 0) { // W * H == 0 if NO intersection happened!
            Position.X = intersection.Right;
        }
    }

    if (!Game.Instance.GetTile(Corners[CORNER_BOTTOM_LEFT]).Walkable) {
        Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_BOTTOM_LEFT]));
        if (intersection.Width * intersection.Height > 0) { // W * H == 0 if NO intersection happened!
            Position.X = intersection.Right;
        }
    }
}
```

Run the game, walking to the left collisions should now happen. But you can walk trough tiles in any other direction. Go ahead and add collision resolution code to the rest of the movement directions. Keep in mind:

- Do walking Up after the Left sample. Up is easyer than Right
- When handling Right & Down, keep in mind that the players position 0,0 is top right
  - You will have to subtract player width or height from the clamp position

Smooth movement

Right now walking around you can kind of get stuck in the wall at random spots. We can fix that. You get stuck in walls because the animation method changes the height of the player sprite. So we do collision checking on one frame with one height, but then turning left will change the height. Changing the height between aniamtions causes us to get stuck. It's an easy fix, find this code in **PlayerCharacter.cs**

```
public PlayerCharacter(string spritePath, Point pos) : base(spritePath, pos) {
    AddSprite("Down", new Rectangle(59, 1, 24, 30), new Rectangle(87, 1, 24, 30));
    AddSprite("Up", new Rectangle(115, 3, 22, 28), new Rectangle(141, 3, 22, 28));
    AddSprite("Left", new Rectangle(1, 1, 26, 30), new Rectangle(31, 1, 26, 31));
    AddSprite("Right", new Rectangle(195, 1, 26, 30), new Rectangle(167, 1, 26, 29));
    SetSprite("Down");
}
```

And change it to have a uniform height

```
public PlayerCharacter(string spritePath, Point pos) : base(spritePath, pos) {
    AddSprite("Down", new Rectangle(59, 1, 24, 30), new Rectangle(87, 1, 24, 30));
    AddSprite("Up", new Rectangle(115, 3, 22, 30), new Rectangle(141, 3, 22, 30));
    AddSprite("Left", new Rectangle(1, 1, 26, 30), new Rectangle(31, 1, 26, 30));
    AddSprite("Right", new Rectangle(195, 1, 26, 30), new Rectangle(167, 1, 26, 30));
    SetSprite("Down");
}
```

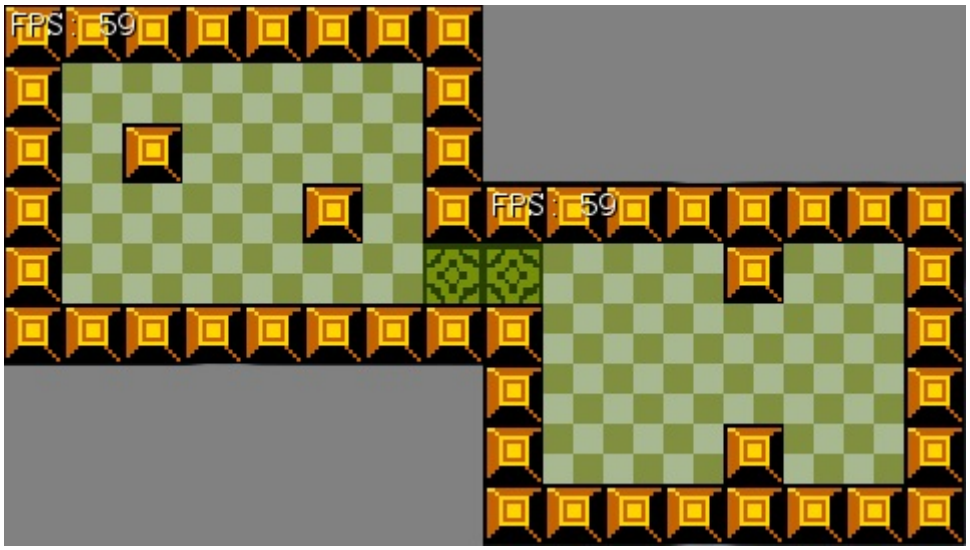
DONE, Run the game. Moving is smooth as a button and collisions work correctly in all directions.

Check in

Check in with me when you finished this part before moving onto the next bit. I want to review the code written up to this point.

Open The Door

In the last chapter we explored how to add collision to the walls of our room. This is aweome! But Link is feeling claustrophobic! He is an epic hero, confined to a tiny room... In this chapter we are going to double the size of Link's world by adding a second room to it! Here is the layout of what this is going to look like:



Of course only one of these rooms is going to be visible at a time, link's going to have to go trough the green door to visit the adjacent room.

The nice thing here, none of the code for this is going to have to go inside the `PlayerCharacter` class, as none of this is specific to the player. Instead we're going to be writing most of our code inside of **Game.cs**

New Project

Let's make a new project, call it **OpenTheDoor** and get this project up to par with the **HitTheWall** section of the writeup. We're going to work from here.

The door tile

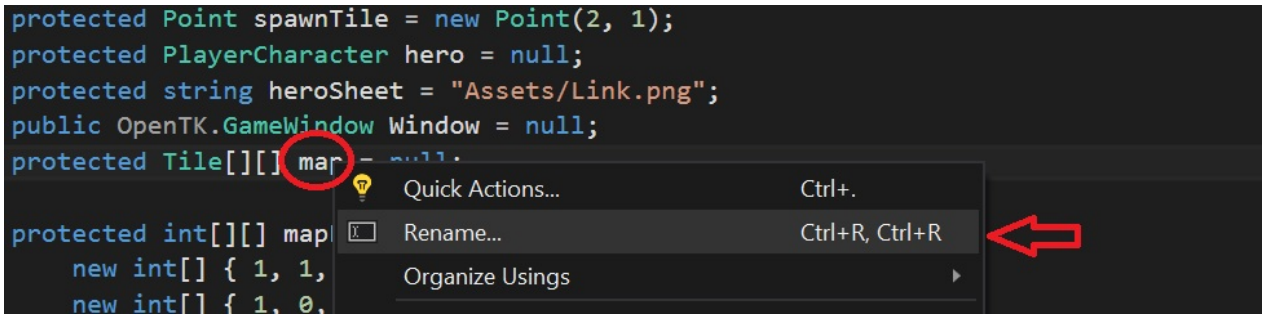
First thing is first, let's add our door tile that will lead into the second room. We're going to be working in **Game.cs**. Right now we define the map layout in the `mapLayout` variable where tile 1 is a wall and tile 0 is walkable. Let's add tile 2, which will be the door.

- In the `mapLayout` varibale, change the appropriate tile to be a door
  - Look at the above screnshot, it's the tile in the lower right
- Add `new Rectangle(32, 187, 30, 30)` to the `spriteSources` array
- Make tile 2 walkable. So tiles 0 and 2 should both be walkable.

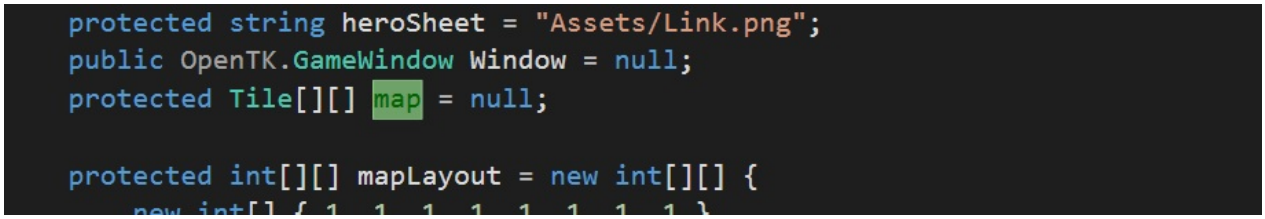
Be careful when walking around! While you should be able to walk a little bit into the door, if you walk too far into the new door tile the game will crash. This is expected. After the above changes, running the game should look like this:

The second room

Sweet we now have a door (well, visually at least), but it doesn't lead anywhere! We're going to add a second map. Still in **Game.cs**, refactor the name of the `map` variable to be `room1`. To do this, right click on `map` and select **Rename**



This will cause the `map` variable to highlight in some strange green color, just start typing and the name of the variable will change. When you are done, hit enter and all instances of `map` will be autmatically renamed for you.



Now rename the `mapLayout` variable to be named `room1Layout`.

Run The Game at this point it's important for you to run the game and make sure it still works correctly!

Next, let's add the definition of `room2` and `room2Layout`. Seeing how this is mostly a copy / paste job of `room1` with minor alterations, use this clode:

```
protected Tile[][] room2 = null;

protected int[][] room2Layout = new int[][] {
    new int[] { 1, 1, 1, 1, 1, 1, 1, 1 },
    new int[] { 2, 0, 0, 0, 1, 0, 0, 1 },
    new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
    new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
    new int[] { 1, 0, 0, 0, 1, 0, 0, 1 },
    new int[] { 1, 1, 1, 1, 1, 1, 1, 1 }
};
```

Inside of the **Initialize** function, make sure that `room2` is created! Also, inside **Destroy** make sure that `room2` is unloaded.

The current room

At this point we have two rooms, but they are kind of useless. This is because every function is hard-coded to work with room1. Let's fix that by introducing a `currentRoom` variable! Start off by adding these two member variables:

```
protected Tile[][] currentRoom = null;
protected int[][] currentLayout = null;
```

At the end of the initialize function point the `currentRoom` and `currentLayout` references at room2, like so:

```
currentRoom = room2;
currentLayout = room2Layout;
```

Of course running the game, link still starts out in `room1`. this is because nothing is using the `currentRoom` variable yet. Update the following to use `currentRoom` instead of `room1`:

- GetTile function
- Render function

Running the game, link should now be in room2. Make sure the collisions are correct.

Walking trough the door

The last thing left to do is to add the logic to walk link trough that door. And this is going to be pretty simple. In the `Update` method of **Game.cs**, we're going to check if the player rectangle intersects with any door rectangles. If it does, based on the door number we will set currentRoom to the appropriate room. Finally, we will set the position of the player to look like he's walked trough the door.

One thing you might have noticed about the above logic, if we transport the player as soon as he touches the door, it might not look natural. It definateley won't look like link is walking trough the door... More like he accidentally touched a magic portal. Because of this, instead of checking links bounding box, we're going to construct a 4PX rectangle at his center. And we will test this against intersecting with the door.

I originally wanted to make this an "On Your Own" exercise, but the code is pretty straight forward, and it's hard to give instructions on it. So, read the below code carefully, and let me know if you have any questions.

```
public void Update(float dt) {
    // Let the hero move, and sort out any collision issues
    hero.Update(dt);
    // We need to check for doors AFTER the player has moved

    // Loop torugh the layout, the Tile class does not know if it is a door or not.
    for (int row = 0; row < currentLayout.Length; ++row) {
        for (int col = 0; col < currentLayout[row].Length; ++col) {
            // The tile is a door if it had a 2 in the layout!
            if (currentLayout[row][col] == 2) {
                // Get the door's bounding rectangle
                Rectangle doorRect = GetTileRect(new PointF(col * 30, row * 30));
```

```
                // Get a small rectangle in the center of the player
                Rectangle playerCenter = new Rectangle((int)hero.Center.X - 2, (int)hero.Center.Y - 2, 4, 4);

                // Look for an intersection
                Rectangle intersection = Intersection.Rect(doorRect, playerCenter);

                // Intersection happens if the intersect rectangle has an area > 0
                if (intersection.Width * intersection.Height > 0) {
                    // If we are in room1, move to room 2
                    if (currentRoom == room1) {
                        // Set active room
                        currentRoom = room2;
                        currentLayout = room2Layout;

                        // Reposition hero to tile outside of door
                        hero.Position.X = 1 * 30; // Column 1
                        hero.Position.Y = 1 * 30; // Row 1
                    }
                    // If we are in room 2, move to room 1
                    else {
                        // Set active room
                        currentRoom = room1;
                        currentLayout = room1Layout;

                        // Reposition hero to tile outside of door
                        hero.Position.X = 6 * 30; // Column 6
                        hero.Position.Y = 4 * 30; // Row 4
                    }
                }
            }
        }
    }
}
```

Run the game, you should now be able to walk from room to room!

You might be thinking to yourself, we could make things easyer by storing a bolean in the `Tile` class, then we wouldn't need to loop trough the tileLayout array.... And you are correct! In fact, that's what the next section is all about. Read on!

Cleaning Up

Follow along to the code in the **Cleaning** Up subsection inside of this (OpenTheDoor) project. The **Cleaning Up** section just gives us an easy way to store room's as objects, instead of cluttering up the **Game.cs** file.

Cleaning Up

We can now open doors to multiple rooms in the game. Very cool! But our `Game` class got very mangled in the process. There is now considetably more code in there! You can imagint the situation will quickly get out of hand as we add more rooms.

Each room makes for a pretty good object. In this section we're going to clean the code up a bit by refactoring the rooms into objects, and hopefully getting the `Game` class back under control.

We're going to call this new object that represent a room `Map` .

Tile Refactor

Before we create the new `Map` class, lets make a quick change to the `Tile` class. In **Tile.cs** add a public bool to this class, call it **IsDoor**. It should be false by default.

Now, inside **Game.cs**, in the `GenerateMap` function where we set the `Walkable` variable, also set the IsDoor variable. It's going to be true if the layout tile is 2, false otherwise.

Finally, in **Game.cs** let's change the `Update` method. The part where it loops trough the `currentLayout` variable and checks the layout index if it's a door or not, this part:

```
for (int row = 0; row < currentLayout.Length; ++row) {
    for (int col = 0; col < currentLayout[row].Length; ++col) {
        // The tile is a door if it had a 2 in the layout!
        if (currentLayout[row][col] == 2) {
```

Let's change that to loop trough the tiles (`currentRoom`) and check the new bool we added. The refactored code looks like this:

```
for (int row = 0; row < currentRoom.Length; ++row) {
    for (int col = 0; col < currentRoom[row].Length; ++col) {
        // The tile is a door if it had a 2 in the layout!
        if (currentRoom[row][col].IsDoor) {
```

Go ahead, **Run the game**. It plays like nothing has happened. If there is no visual difference, why did we make this change? Simple, this loop and tile check was the only reason we needed to keep the `currentLayout` variable around. Now, when we refactor `Map` to be it's own object, it won't need to track layout.

Given that there are no more practical references to `currentLayout` anymore, go ahead and remove the variable. Delete it's definition and any lingering assignments. Get the code to compile without it.

This isn't the last refactor we're going to make to the Tile class in this section, but let's hold off on the other refactors until they make sense.

Map class

Go ahead and create a new file, let's call it **Map.cs**. We're not going to add a lot of new code to this class, or the most part we're just going to move code out of **Game.cs** and into here.

A map is just a 2D array of `Tile` objects, so lets add a protected 2D `Tile` array to this class and call it **tileMap**. If we expose the same API as an array being used, then the `Map` class and a 2D int array will behave the same (inside of **Game.cs**). So what do we need to mimic an array API? We need a bracket accessor and a .Length property.

So far our class looks like this:

```
using System.Drawing;

namespace OpenTheDoor {
    class Map {
        protected Tile[][] tileMap = null;

        // Accessor to the tileMap variable. With this we can index the map object
        // without directly exposing the underlying tileMap. EX:
        // Map myMap = new Map();
        // myMap[row][col] = new Tile(row, col);
        public Tile[] this[int i] {
            get { return tileMap[i]; }
        }

        // We may need to know how many rows / columns there are in the map, that's what
        // this function is great for!
        public int Length {
            get {
                return tileMap.Length;
            }
        }
    }
}
```

Next, let's add a constructor:

```
public Map(int[][] layout, string sheets, Rectangle[] sources, params int[] walkable) {
```

If you notice the constructor here has a very similar signature to the `GenerateMap` function inside of **Game.cs**. That's because it essentially does the same thing. Go ahead and migrate the `GenerateMap` function into the map constructor.

Take these lines:

```
result[i][j].Walkable = layout[i][j] == 0 || layout[i][j] == 2;
result[i][j].IsDoor = layout[i][j] == 2;
```

And replace them with sane defaults (don't worry, we will change these later):

```
result[i][j].Walkable = false;
result[i][j].IsDoor = false;
```

After setting all defaults, we want to loop trough the params argument at the end. If the value of the tile we just created is in the walkable array, then we should mark the tile as walkable. Because an array doesn't have a search function, we will just do a linear search manually:

```
foreach (int w in walkable) {
    if (layout[i][j] == w) {
        tileMap[i][j].Walkable = true;
    }
```

```
}
```

It makes no sense for the `Game` class to have to loop torugh a room and destroy all textures indevidually. Same thing with rendering, why is the `Game` class looping? Let's fix that by moving the loops into the `Map` class like so:

```
public void Render() {
    for (int h = 0; h < tileMap.Length; h++) {
        for (int w = 0; w < tileMap[h].Length; w++) {
            tileMap[h][w].Render();
        }
    }
}

public void Destroy() {
    for (int h = 0; h < tileMap.Length; h++) {
        for (int w = 0; w < tileMap[h].Length; w++) {
            tileMap[h][w].Destroy();
        }
    }
}
```

That's our first round on the `Map` object, let's see what we need to do to **Game.cs** to make it work:

Game refactor

We have to do a little bit of re-factoring in the `Game` class to make it work with our new `Map` class. Here is a pretty comprehensive list of what needs to happen

- Change `Tile[][] room1 = null;` to `Map room1 = null`
- Repeat the above for room2
- Repeat the above for currentRoom
- Change the `Initialize` function to create a new `Map`
  - Last argument (params) is going to be 2 and 0 as they are walkable
- Inside `Initialize` still, map 1: manually set tile 4, 7 IsDoor to true
- Inside `Initialize` still, map 2: manually set tile 0, 2 IsDoor to true
- Delet the `GenerateMap` function from **Game.cs**
- In the **Game.cs** `Destory` function, don't loop torugh maps.
  - Call that map object's `Destroy` function instead
- In the **Game.cs** `Render` function, don't loop trough the currentMap.
  - Instead just call `currentMap.Render();`

If you need a reference, at this point my **Game.cs** looks like This.

Run the game, at this point your game should run like if nothing had changed.

Whats next?

The project is a little bit better orgonized now. We removed some code from ***Game.cs**, but not enough. I think we should be able to make the `Update` function into a 2 line function. I want it to work like this

```
// PSEUDOCODE, NOT FOR PRODUCTION
public void Update(float dt) {
    hero.Update(dt);
    currentMap = currentMap.ResolveDoors(hero);
```

If the player leaves the room, `ResolveDoors` will return the new room the player is in, and move the player to the correct new location. In order to do this the map class will need to be able to check for doors, and the doors will need to know where to take the player.

Tile refactor

Door tiles will need to know two more pieces of information, where the door should take the player in pixels on screen and which room to take the player to. With that in mind, add these two variables to **Tile.cs**

- `public Map DoorTarget = null;`
- `public Point DoorLocation = new Point();`

Set them to good defaults (null, and empty Point). Next we're going to make a function that sets both of these, and the IsDoor variable. Lets call this function **MakeDoor** The function is super straight forward, this is what it looks like:

```
public void MakeDoor(Map target, Point location) {
    DoorTarget = target;
    DoorLocation = location;
    IsDoor = true;
}
```

That's it, for the moment the `Tile` class is done. We're not going to be touching it for the rest of this section.

Map Refactor

Let's add a new method to the `Map` class. This is the method we're going to use to test if a door intersection has happened.Let's call that method `ResolveDoors` and give it the following signature:

```
public Map ResolveDoors(PlayerCharacter hero) {
```

Go ahead and **try to move** the door related update function out of **Game.cs** and into **Map.cs** I expect you will have a hard time with the code that decides that a collision has happened. Again, the code here is pretty straight forward, i'll include my version:

```
public Map ResolveDoors(PlayerCharacter hero) {
    Map result = this; // Return this map by default!

    for (int row = 0; row < tileMap.Length; ++row) {
        for (int col = 0; col < tileMap[row].Length; ++col) {
            if (tileMap[row][col].IsDoor) {
                Rectangle doorRect = new Rectangle(col * 30, row * 30, 30, 30);
                Rectangle playerCenter = new Rectangle((int)hero.Center.X - 2, (int)hero.Center.Y - 2, 4, 4);
                Rectangle intersection = Intersection.Rect(doorRect, playerCenter);

                // Intersection happens if the intersect rectangle has an area > 0
                if (intersection.Width * intersection.Height > 0) {
                    result = tileMap[row][col].DoorTarget;
                    hero.Position.X = tileMap[row][col].DoorLocation.X * 30;
                    hero.Position.Y = tileMap[row][col].DoorLocation.Y * 30;
                }
            }
        }
    }

    return result;
}
```

That's a short, good looking function! And with that we are done refactoring the `Map` class. We just need to fix up **Game.cs** now.

Game Refactor

I order to utilize these shiny new features we've added we have to do some refactoring on **Game.cs**, luckly, it's just two things:

- In initialize, wehre we where setting `` `IsDoor `` to true, don't do that
  - Instead call the `MakeDoor` function of `Tile`
- In Update, tear out that big ugly nested loop ment for door checks.
  - Replace it with: `currentRoom = currentRoom.ResolveDoors(hero);`

And that's it! You should now be able to **Run Your Game**, and it should look exactly like our game did at the begening of the tutorial. The big difference? The code is now maintainable! In fact, the **Game.cs** class ended up so small, i'm going to include it verbatin:

```
using GameFramework;
using System.Drawing;
using Collision;

namespace OpenTheDoor {
    class Game {
        protected Point spawnTile = new Point(2, 1);
        protected PlayerCharacter hero = null;
        protected string heroSheet = "Assets/Link.png";
        public OpenTK.GameWindow Window = null;

        Map room1 = null;
        Map room2 = null;
        Map currentRoom = null;

        protected int[][] room1Layout = new int[][] {
            new int[] { 1, 1, 1, 1, 1, 1, 1, 1 },
            new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
            new int[] { 1, 0, 1, 0, 0, 0, 0, 1 },
            new int[] { 1, 0, 0, 0, 0, 1, 0, 1 },
            new int[] { 1, 0, 0, 0, 0, 0, 0, 2 },
            new int[] { 1, 1, 1, 1, 1, 1, 1, 1 }
        };

        protected int[][] room2Layout = new int[][] {
            new int[] { 1, 1, 1, 1, 1, 1, 1, 1 },
            new int[] { 2, 0, 0, 0, 1, 0, 0, 1 },
            new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
            new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
            new int[] { 1, 0, 0, 0, 1, 0, 0, 1 },
            new int[] { 1, 1, 1, 1, 1, 1, 1, 1 }
        };

        protected string spriteSheets = "Assets/HouseTiles.png";
        protected Rectangle[] spriteSources = new Rectangle[] {
            new Rectangle(466,32,30,30),
            new Rectangle(466,1,30,30),
            new Rectangle(32, 187, 30, 30),
            new Rectangle(32, 187, 30, 30)
        };

        public Tile GetTile(PointF pixelPoint) {
            return currentRoom[(int)pixelPoint.Y / 30][(int)pixelPoint.X / 30];
        }

        public Rectangle GetTileRect(PointF pixelPoint) {
            int xTile = (int)pixelPoint.X / 30;
            int yTile = (int)pixelPoint.Y / 30;
            Rectangle result = new Rectangle(xTile * 30, yTile * 30, 30, 30);
```

```
            return result;
        }

        private static Game instance = null;

        public static Game Instance {
            get {
                if (instance == null) {
                    instance = new Game();
                }
                return instance;
            }
        }

        protected Game() {

        }

        public void Initialize(OpenTK.GameWindow window) {
            Window = window;
            window.ClientSize = new Size(room1Layout[0].Length * 30, room1Layout.Length * 30);
            TextureManager.Instance.UseNearestFiltering = true;

            hero = new PlayerCharacter(heroSheet, new Point(spawnTile.X * 30, spawnTile.Y * 30));
            room1 = new Map(room1Layout, spriteSheets, spriteSources, 0, 2); // 0 and 2 ae walkable
            room2 = new Map(room2Layout, spriteSheets, spriteSources, 0, 2); // 0 and 2 are walkable
            currentRoom = room1;

            room1[4][7].MakeDoor(room2, new Point(1, 1)); // Go to room2, at tile 1, 1
            room2[1][0].MakeDoor(room1, new Point(6, 4)); // To to room1, at tile 6, 4
        }

        public void Update(float dt) {
            hero.Update(dt);
            currentRoom = currentRoom.ResolveDoors(hero);
        }

        public void Render() {
            currentRoom.Render();
            hero.Render();
        }

        public void Shutdown() {
            room1.Destroy();
            room2.Destroy();
            hero.Destroy();
        }
    }
}
```

Screen Borders

The door section was a bit more intense than originally expected. Luckly for you, this section of the tutorial has no code for you to write. I just wanted to present you with an alternate way to use doors. There are of course many more ways to use them, not just the one we have already seen and the one we're about to see. I encourage you to think about and play with door mechanics.

The legend of zelda actually uses two different types of doors. Real doors and hidden doors. A real door looks like this:

Notice how link is standing right in the doorway. If he moves just one more pixel down he will leave the house. This is similar to how we handled door interaction.

In outdoor environments however, hidden doors are used. Take a look at this picture:

Link is just standing on the edge of the screen. If that was a door tile, the entire screen would already be transitioning. Instead, link has to walk down one more tile to trigger the screen transition. Here that is in action:

So how is this done? The simple answer is that there are door tiles outside of the render area! When link walks off-screen instead of the game breaking because there not being any more tiles to check for collision against, there are hidden tiles off-screen. These tiles are collided with, colliding with a door tile causes a screen trnasition.

Here is an example of what the hidden tiles *might* look like:

The legend of zelda is unique in that the world doesn't scroll. At the edge of every screen is a hidden door or 7. Other games, like Final Fantasy do scroll with the player. They don't need any hidden doors. We will discuss how to scroll the world with the player later.

Jumping

Tile games don't al have to be top down. We can change perspectives pretty easily. In this chapter we are going to do just that. Instead of the game playing top-down we're going to make it play from a side view. Mario style! The gameboy zelda games do include some side scrolling sections:

New Project

Let's make a new project, call it **Jumping** and get this project up to par with the cleaned up OpenTheDoor section of the writeup. We're going to work from here.

PlayerCharacter refactor

One of the most important things in turning our tile engine from top down into a side scroller is to not allow horizontal movement! Let's add a new `#define` the top of the **PlayerCharacter.cs**

```
//#define ENABLE_VERTICAL_MOVEMENT
```

Have the new define be commented out as we're not going to be using it! Now go to the `Update` function, and find the if-elseif block that handles up and down movement. Put this into a `#if` block, add a `#else` block, we're going to be working in the `#else` block.

```
#if ENABLE_VERTICAL_MOVEMENT
    // Vertical movement code
#else if
    // We're going to be writing the code for this section in here.
#endif
```

Gravity

Adding gravity is going to be pretty straight forward, first let's declare a new member variable and call it `gravity`. This veriable represents how many *pixels* the character is going to fall per second. So, if we wanted him to fall 2 tiles we would have to set gravity equal to 2 * 30.

```
protected float gravity = 7 * 30; // Fall 7 tiles / second
```

Now go to the `#else` block and add gravity times delta time to the players y position every frame. If you **run the game** the player character will simply fall down, your application might or might not crash. Let's fix this by adding

some collision detection.

In the commented out (by the `#if` section of code we have some code to handle the case of link trying to walk down and off screen. We're going to copy **ONLY** the collision handling part of that code, and paste it after updating the players position by gravity. This is the bit of code i'm talking about:

```
if (!Game.Instance.GetTile(Corners[CORNER_BOTTOM_LEFT]).Walkable) {
    Rectangle intersection = Intersection.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_BOTTOM_LEFT]));
    if (intersection.Width * intersection.Height > 0) {
        Position.Y = intersection.Top - Rect.Height;
    }
}
if (!Game.Instance.GetTile(Corners[CORNER_BOTTOM_RIGHT]).Walkable) {
    Rectangle intersection = Intersection.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_BOTTOM_RIGHT]));
    if (intersection.Width * intersection.Height > 0) {
        Position.Y = intersection.Top - Rect.Height;
    }
}
```

Run the game again and you should now be colliding with tiles. Link starts off on a platform and can walk left or right to fall. You can even enter the other room!

Jumping Theory

If you are trying to figure out how to make a 2D character jump, google will sooner or later take you to this stack overflow answer. The answer gives the standard jump parabola, to jump you need two forces:

- **Gravity** - always pulls the player down
- **Velocity** - can push player up or down

We're going to use these two forces, but they will not both effect the player. Gravity is going to be constantly pulling velocity down. Velocity will be moving the player. We will also introduce a third force, **Impulse**.

- **Impulse** - force exerted by character to change velocity

Gravity will always be pulling velocity down, and velocity will move the character. So how can velocity ever push the character up? In order to do so we have to apply another force to velocity, this force is impulse. While gravity is constantly pulling down, impulse will suddenly push up.

Example 1: Character is at (30, 60). Our velocity is 0. Gavity is 2.

- Frame 1: *Character in initial position*
  - Character is stationary velocity is 0, character at y pixel 60
  - Velocity is pulled down by gravity (2) it's now 2
- Frame 2: *Character starts falling*
  - Character is moved down by velocity (2) to y pixel (58)
  - Velocity is pulled down by gravity (2) it's now 4
- Frame 3:
  - Character is moved down by velocity (4) to y pixel (54)
  - Velocity is pulled down by gravity (2) it's now 6
- Frame 4:
  - Character is moved down by velocity (6) to y pixel (48)
  - Velocity is pulled down by gravity (2) it's now 8
- Frame 5:

    - Character is moved down by velocity (8) to y pixel (40)
    - Velocity is pulled down by gravity (2) it's now 10

As you can see from this 5 frame example, the simple model outlined above has one simple flaw, velocity is constantly increasing! The longer the character falls, the faster he will fall! We're going to fix this by clamping velocity to gravity: `velocity = Math.Min(velocity, gravity` . I don't think i need to include an example for the clamped velocity. But let's see an example with it in place where we apply an impulse!

Example 2: Character is at (30, 60). Our velocity is 0. Gavity is 2. Impulse is -6

- Frame 1: *Character in initial position*
    - Character is stationary velocity is 0, character at y pixel 60
    - Velocity is pulled down by gravity (2) it's now 2
- Frame 2: *Character starts falling*
    - Character is moved down by velocity (2) to y pixel (58)
    - Velocity is pulled down by gravity (2) it's 4, it gets clamped back to 2 (the value of gravity)
- Frame 3:
    - Character is moved down by velocity (2) to y pixel (56)
    - Velocity is pulled down by gravity (2) it's 4, it gets clamped back to 2 (the value of gravity)
- Frame 4:
    - Character is moved down by velocity (2) to y pixel (54)
    - Velocity is pulled down by gravity (2) it's 4, it gets clamped back to 2 (the value of gravity)
- Frame 5:
    - Character is moved down by velocity (2) to y pixel (52)
    - Velocity is pulled down by gravity (2) it's 4, it gets clamped back to 2 (the value of gravity)
- Frame 6: **Impulse is added**
    - Character is moved down by velocity (2) to y pixel (50)
    - Velocity is pulled down by gravity (2) it's 4, it gets clamped back to 2 (the value of gravity)
    - Impulse (-6) is added to velocity (2), velocity is now -4
- Frame 7: *Character is moving up*
    - Character is pushed up by velocity (-4) to y pixel (56)
    - Velocity is pulled down by gravity (2) it's now -2
- Frame 8:
    - Character is pushed up by velocity (-2) to y pixel (54)
    - Velocity is pulled down by gravity (2) it's now 0
- Frame 9: *nothings happening*
    - Character is stationary velocity is 0, character at y pixel 60
    - Velocity is pulled down by gravity (2) it's now 2
- Frame 10: *falling down again*
    - Character is moved down by velocity (2) to y pixel (56)
    - Velocity is pulled down by gravity (2) it's now 2

I feel like a companion gif would be really helpful, but i have no idea how to make proper gif images. If the above doesn't make sense give me a call and i will try my best to explain it!

Jumping implementation

Let's start by adding our new variables. Gravity and impulse are constant forces, they don't change frame over frame. But velocity does:

```
float gravity = 210.0f; // Fall 7 tiles / second, constant
```

```
float impulse = -180.0f; // Randomly chosen, constant
// Impulse is negative because the force is going up!
float velocity = 0.0f; // Changes every frame
```

Next, we need to change the update method.

- Instead of adding `gravity * deltaTime` to `Position.Y` add `velocity * deltaTime`
- Above that, add `gravity * deltaTime` to velocity! This moves velocity down.
- Don't forget to clamp `velocity`, it can never be greater than gravity!

Run the game now, and nothing has changed! If you did everything correctly you should be able to walk around, fall off the platform, all kinds of fun! Let's add the code to actually jump. *Before applying gravity to velocity, check is the space bar is pressed*. If it is, **set velocity equal to impulse**. This is how we apply the impulse force. **Run the game now** and you should be able to jump after having fallen off the platform the character starts on.

Hint The above paragraph contains some code instructions that are easy to miss I tried to bold it to make them stand out.

Run the game and confirm you can jump. The jumping might feel a bit floaty right now, but that's ok. We will fix that later. Right now we have a bigger problem. There is no top collision! You can jump right out of the game! Kind of an easy fix. We already have code to prevent link from moving trough tiles upwards in the commented out `#if` block, let's go ahead and add that back in. Paste the top collision checking code after the code to check for floor collision. This is the bit that i'm talking about:

```
if (!Game.Instance.GetTile(Corners[CORNER_TOP_LEFT]).Walkable) {
    Rectangle intersection = Intersection.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_LEFT]));
    if (intersection.Width * intersection.Height > 0) {
        Position.Y = intersection.Bottom;
    }
}
if (!Game.Instance.GetTile(Corners[CORNER_TOP_RIGHT]).Walkable) {
    Rectangle intersection = Intersection.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_RIGHT]));
    if (intersection.Width * intersection.Height > 0) {
        Position.Y = intersection.Bottom;
    }
}
```

It's almost perfect. However if you jump up and hit your head, you kind of float around for a while... This is immediateley obvious when you jump below the first platform. The reason for this should be obvious by now, velocity is pulling link up, but the tile collision clamps his position. This however doesn't stop velocity from pulling him up.

The fix is simple, we need to change velocity! We need to apply this code in two places, after each collision handler (`Position.Y = intersection.Bottom;`, still inside the area check if statement). But what do we set `velocity` to??? As it turns out we have three options, i encourage you to try all three:

- **0** - This will cause link to fall from the top of his parabola. That is, slowly
  - Can be visually jarring if you hit something mid jump.
  - This is because the falling / jumping speed is different based on jump time
- **gravity** - This will cause a very abrupt fall.
  - Can be visually jarring, if link is at the top of his jump his fal will be super fast
- **Math.Abs(velocity)** or **velocity * -1.0f**. This will keep fall speed the same as jump speed.
  - The speed in which link travels stays the same, only direction changes
  - The most visually appealing in my opinion.

So yeah, in my code i set `velocity = Math.Abs(velocity);`, i suggest you do the same, it looks the best and is closest to what would actually happen in the real world.

Jump animation

Now that the jump is functional, let's add an animation to it. If you look at the link sprite sheet, he has a jump animation! First, add the jump sprite to the player character:

```
AddSprite("Jump", new Rectangle(122, 75, 23, 30), new Rectangle(154, 76, 22, 30));
```

Next, in the if statement that sets impulse, set the sprite as well

```
if (i.KeyPressed(OpenTK.Input.Key.Space)) {
    velocity = impulse;
    SetSprite("Jump");
}
```

Lastly, after the above if statement add a new one. We're going to check if velocity is anything else than gravity. If so, we're going to call animate, because link is mid-jump.

```
if (velocity != gravity) {
    Animate(deltaTime);
}
```

Run the game and you're going to notice several stange artifacts. Link starts out animated. Landing a jump shows the wrong sprite. Landing on a higher platform keeps the animation playing. These are simple bugs introduced by the new animation. Luckly we can fix them pretty easy. Let's do that one at a time.

Link start out animated, this is the easyest one. When the game starts velocity is equal to 0, not gravity. So, technically even tough link has hit a platform he is still falling. Find the code that handles links collision with objects below him, and set velocity equal to gravity. Hint: You will have to do this in two places, just like the code where he hit his head.

Now, when link lands a jumphe stays in a strange sate that is just his last jump sprite. The fix would be to set the sprite back to link left or link right, but we don't know which direction he started the jump from. While we could add some bools and figure this out, let's take a more creative approach. Find the code where link lands on an object below him. Before setting velocity to gravity, if velocity does not equal gravity, set his sprite to down. Like so:

```
if (intersection.Width * intersection.Height > 0) {
    Position.Y = intersection.Top - Rect.Height;
    if (velocity != gravity) {
        SetSprite("Down");
    }
    velocity = gravity;
}
```

This works because the only time link connects with the ground and has a velocity different from gravity is if he is comming out of a jump. When link falls his velocity is already equal to gravity. One more big bug remains, if you jump and press left or right to move mid jump, link plays his walk animation. Luckily we can fix this using the same trick.

Find where you set links sprite to left and right, and wrap it in an if statement. The if statement should check if velocity equals gravity (`if (velocity == gravity) {`) and encompas both the SetSprite call and the Animate call.

Double Jumps

Right now link can jump indefinateley, i think that might actually be called flying at this point! Let's limit him so he can only do one jump at a time. Now, one jump is a bit tricky, he will be able to jump if he's on the ground, or mid fall (so long as he is falling from a platform, not a jump).

The fix here is easy, only add impulse if the players velocity is not the same as gravity. Because the players velocity is only the same as gravity if they are on the ground, or if they are falling from a platform. So find this bit of code:

```
if (i.KeyPressed(OpenTK.Input.Key.Space)) {
    velocity = impulse;
    SetSprite("Jump");
}
```

And change the if statement to this:

```
if (i.KeyPressed(OpenTK.Input.Key.Space) && velocity == gravity) {
```

Next

We're almost done! Jumping is mechanically correct and the visuals look decent. The only problem is the "feel" of the jump. It's a bit floaty for my taste, i also think it could go a tad higher. We're going to fix that in the next section.

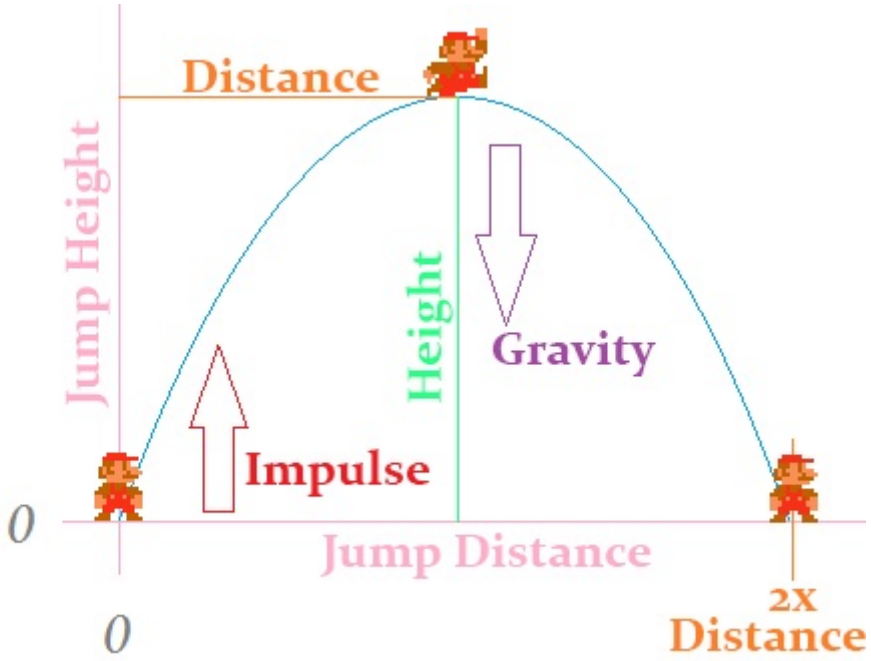
Configure the jump

The ability to jump is great! But right now it feels a bit off. Jumping is controlled by two variables, gravity and impulse. If we want to change the jump behaviour we have to change these two numbers. Here is the thing tough, i'm an engineer and i don't know what the gravity needs to be to jump 3 tiles high in half a second.

When you design a jump having gravity and impulse is not intuitive. You usually want to design a jump around two other values, **Height** and **Duration**. Saying i want to jump 3 tiles tall in 0.25 seconds is super intuitive, and makes creating Different jump behaviours a lot easyer.

So, can we actually do this? Come up with a way to specify height and duration instead of gravity and velocity? Yes, yes we can. With math. Complicated math. Hard math. Confusing math. Man math! What follows might be confusing or hard. I didn't come up with it. This is math i can't actually derive, i had to look up the formulas. I'll link where i found those formulas at the end of the chapter.

The jumping system



In the above picture there are four forces controlling the jump:

- **height**: The maximum peak height of our jump in world units
- **duration**: The duration from when we jump until we reach the peak in seconds
- **impulse**: Our initial upward velocity when we jump in units per second
- **gravity**: Our acceleration due to gravity pulling us back down in units per second squared

Only impulse and gravity is needed to **define the jump**, height and duration actually **describe the jump**. We already have the jump definition in our code, all we need is to figure out a way to convert the jump description into a definition. Lickly given any two of these four items, we can find the other two.

Jumping conversions

Given a height and duration:

- **impulse** = 2 * height / duration
- **gravity** = negative impulse / duration

Using the above formulas, add this method to the **Player Character** class, and fill in the blanks:

```
public void SetJump(float height, float duration) {
    impulse = // TODO
    impulse *= -1; // Remember, in our implementation model, impulse starts off negative!
    gravity = // TODO
}
```

The above are the only conversions we care about, but for the sake of fullness, here are ALL of the conversions:

- **height** = impulse * duration / 2
- **impulse** = 2 * height / duration
- **duration** = 2 * height / impulse
- **impulse** = negative gravity * duration
- **duration** = negative impulse / gravity
- **gravity** = negative impulse / duration

Configuring the jump

In the **PlayerCharacter.cs** `Initialize` function, after you've added all the sprites call the new `SetJump` function. Just for fun you can add a console log to the SetJump function to see what the impulse and gravity you are setting are eventually going to be.

Play around wih the numbers for a bit to see it working. Remember, height is in world pixels. This means if you want to jump 3 tiles height will need to be 3 * 30. Also, duration is in seconds. If you set either number too high the game is going to crash, this is because link will simply tunnel trough the obstacles, into the void.

Now that you've played around with the values, here are the defaults we will be using for the rest of the book:

- **height**: 3.0f * 3.0f // 3 tiles
- **duration**: 0.75f // three fourths of a second

Check in

Before moving onto the next section (Clouds) i want you to check in with me, we did some pretty complicated code for jumping and i want to make sure that there are no errors in your code at this point.

Formula origins

Like i said, i don't know the formulas for height / impulse / gravity / duration conversion by heart. What's worse, i'm not good enough at math to derive them! This is actually something pretty common in games, we need complicated math but can't always derive said math. Luckly other people have done so for us. A little google goes a long way.

All of the formulas i've used on this page have come from this article. While researching i also found another promising article, but the first one had the exact formulas i needed so i never read the second one. You should read trough those articles. They might make more sense to you than they did to me.

Clouds

A cloud is a pretty simple construct in a platformer. You can walk trough a clouds left or right side. You can jump from below the clowd over it. So, there is no collision on it's bottom, left or right. However, you can land on a cloud. So it does have collision on the top. In this section, we're going to build a simple cloud. Things like breakable bricks or enemies in super mario are similar, they do one thing from one direction and another thing from another direction.

New Project

Let's make a new project, call it **Clouds** and get this project up to par with the **Configurable Jumping** section of the writeup. We're going to work from here.

Updated tile sheet

I've updated **HouseTiles.png** to be a new image. Right click on the below image, save as and overwrite your old one. This tile sheet is still shared between projects, i didn't touch any of the existing tiles, instead the sheet is now larger at 512x512 pixels.

Updated rooms

In **Game.cs** replace the **SpriteSources** array with this revised version:

```
protected Rectangle[] spriteSources = new Rectangle[] {
    // OLD
    /* 0  */new Rectangle(466,32,30,30),
    /* 1  */new Rectangle(466,1,30,30),
    /* 2  */new Rectangle(32,187,30,30),
    // NEW
    /* 3  */new Rectangle(466, 125,30,30), // blue border
    /* 4  */new Rectangle(311, 249,30,30), // blackness
    /* 5  */new Rectangle(466, 63,30,30), // ground layer
    /* 6  */new Rectangle(63, 218,30,30), // blank ladder
    /* 7  */new Rectangle(156, 218,30,30), // ground ladder
    /* 8  */new Rectangle(63, 249,30,30),  // skele 1
    /* 9  */new Rectangle(94, 249,30,30), // slele 2
    /* 10 */new Rectangle(125, 249,30,30), // skele 3
    /* 11 */new Rectangle(156, 249,30,30), // cloud 1
    /* 12 */new Rectangle(187, 249,30,30), // cloud 2
};
```

Next update room 1 layout and room 2 layout

```
protected int[][] room1Layout = new int[][] {
    new int[] { 3, 3, 3, 3, 3, 3, 3, 3 },
    new int[] { 3, 4, 4, 4, 4, 4, 4, 3 },
    new int[] { 3, 4, 4, 4, 4, 4, 4, 3 },
    new int[] { 3, 4,11,12, 4, 4, 4, 3 },
    new int[] { 3, 4, 4, 4, 4, 4, 4, 2 },
    new int[] { 3, 5, 5, 5, 5, 5, 5, 5 }
};
protected int[][] room2Layout = new int[][] {
    new int[] { 3, 3, 3, 3, 3, 3, 3, 3 },
    new int[] { 3, 4, 4, 4, 4, 4, 8, 3 },
    new int[] { 3, 4, 4, 4, 4, 4, 9, 3 },
    new int[] { 3, 4, 4, 4, 4, 4,10, 3 },
    new int[] { 2, 4, 4, 4, 4, 4, 4, 3 },
    new int[] { 5, 5, 5, 5, 5, 5, 5, 3 }
};
```

Find where we make the rooms, and add 4, 8, 9 and 10 as walkable tiles:

```
room1 = new Map(room1Layout, spriteSheets, spriteSources, 2, 0, 4, 8, 9, 10);
room2 = new Map(room2Layout, spriteSheets, spriteSources, 0, 2, 4, 8, 9, 10);
```

The new rooms have adjacent doors. **On your own, update the door tiles**.

Running the game, you should see these two rooms:

You should be able to walk around without crashing. Confirm that you can walk from room 1 to room 2 and from room 2 from room one. You are ready to proceed when it works.

Refactoring Tile

Go into **Tile.cs** and add a new public boolean to the `Tile` class. Call this bool `IsCloud` and give it a default vlaue of false!

Refactoring Game

Back in **Game.cs** find the code that makes doors. Immediateley after that set the stone platform (11 and 12) to be clouds. Do this by indexing the map array and setting the `.IsCloud` bool in the tiles to true. So far nothing has really changed, if you try to run the game the platform is still solid from all sides.

Refactoring PlayerCharacter

In the Left / Right movement, find where we check if the tile is walkable, it looks something like this:

```
if (!Game.Instance.GetTile(Corners[CORNER_TOP_LEFT]).Walkable) {
```

We have to edit this check, the new check should be **If corner is not walkable AND corner is not cloud**. Remember, both left and right have two checks one for the upper corner and one for the lower corner.

Finally, find where the upper collision does that same check. Modify the if statement in the same way.

Run the game, you should be able to do everything you have been able to do before. The big difference is now you can walk under the platform and jump up. It will land you on the platform.

Smooth landing

The only problem we have at this point is when the player passed trough the platform he was snapped to it. This is a jarring action that we want to avoid. Instead we want to go ahead and let him finish the jump.

The key to finishing the jump is going to be in velocity. Where we check if the feet of the player collide with the top of the platform, we are also going to check the platform type. If the platform type is cloud AND the players velocity is negative then we will not clamp.

Find this bit of code:

```
if (!Game.Instance.GetTile(Corners[CORNER_BOTTOM_LEFT]).Walkable) {
    Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_BOTTOM_LEFT]));
    if (intersection.Width * intersection.Height > 0) {
        // Ground Player
```

and change it according to the above description. Here is a pseudo code guide to help you get started:

```
if (!Game.Instance.GetTile(Corners[CORNER_BOTTOM_LEFT]).Walkable) {
    Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_BOTTOM_LEFT]));
    if (intersection.Width * intersection.Height > 0) {
        if ( [Not a cloud] OR [Velocity greater than 0]) {
            // Ground Player
```

All we do is wrap the grounding code in another if statement. Remember, the grounding code exists twice, once for

the bottom left and once for the bottom right corner.

What's up with that if statement!?!?! **Not cloud || height > 0** ??? I tought we wanted to check **is cloud && height < 0**!! The thing is, those if statements are logically the same. The logic on each side of the operator is flipped, and so is the operator. This works fine in this case.

Run the game standing under the platform, you should now be able to jump trough it, complete the jump and land!

Done

This was a fairly quick lesson, this is because adding special case tiles is easy. If you need to create something like a brick for super mario, or a bush for the legend of zelda follow the steps we just took.

- First, you update the tile to contain the new variable.
- Then you update Game to set the new variable.
- Last you update player (or enemy, or game) to handle the new special case tile.

Simple Enemy

Having a game where your hero is in an empty room is not much fun! Let's add some enemies to the game! For now they are going to be simple enemies, mario style. The enemy will spawn and start walking. It will walk until it hits a wall, then turn around. If at any point the enemy touches the player, the player dies. Just like mario.

The Plan

How will this work? First off, we need to make an `EnemyCharacter` class. Games have multiple `EnemyCharacter` classes, one for each type of enemy. So for zelda you might have **MoblinEnemyCharacter**, **ChompChompEnemyCharacter**, **ChickenEnemyCharacter** and so on. We are only going to have one enemy class for now `EnemyCharacter`.

So, where do enemies go? Right now the `Game` owns the `PlayerCharacter`. Can we just put an enemy array in `Game`? Not really. Because our game has multiple rooms (or levels, whatever we are calling them), each room has it's own set of enemies. If you are in `Room1` then the enemies in `Room2` should not update or render! Therefore it makes sense to add a `EnemyCharacter` array inside of the `Map` class.

Now that map has a list of enemy characters it's going to need an update method. It needs this so the enemies can update themselves. It would also make sense to check for player collision in here! Instead of the player having to know about all the enemies, we just mape the map know about the player. Because map knows about the enemies and the player it's trivial to check for collision in there.

Finally, we need to add a game over state. This is going to be a simple bool in the `Game` class that controls what text we see on screen.

New Project

Let's make a new project, call it **SimpleEnemy** and get this project up to par with the cleaned up version of the **OpenTheDoor** section of the writeup. We're going to work from here.

Character refactor

Before creating the enemy class we need to refactor the `Character` and `PlayerCharacter` classes a bit. Right now all of the animation code is in `PlayerCharacter` seeing how both the player and the enemy will animate, we should move this functionality into the **Character** class.

The collision handling code is also all in `PlayerCharacter`, and both player and enemy are going to collide with the wall, but for now we're going to duplicate the collision code across these two classes.

This refactor is super simple. Take the **Animate** method from `PlayerCharacter` and move it into `Character`. Next take the **animFPS** and **animTimer** variables and also move them into `Character`. That's it. The animation code is now shared for all characters!

Run the game, everything should work like before.

Creating EnemyCharacter

The enemy characters are going to be simple. They are either going to move up or down. They are going to start walking and keep walking until they hit a wall. Once they hit a wall they are going to change direction. This is about

as easy as you can get with an enemy, coincidentally most enemies in Super Mario work like this.

Add a new file, let's call it **EnemyCharacter.cs**, in it make the `EnemyCharacter` class a child class of `Character` (Just like PlayerCharacter is a subclass of Character). Add three variables to this new class:

- float **speed** = 60.0f;
  - Determines how fast an enemy walks. Measured in pixels per second.
  - The player moves at 90, so the enemies are a little slower.
  - 60 pixels per second on a 30 pixel grid means enemies move 2 tiles / second
- bool **moveUpDown** = false;
  - If true, the enemy will move in an up and down direction
  - If false, the enemy will move in a left to right direction
- float **direction** = 1.0f;
  - Multiplies speed to determine which direction the enemy walks
  - If positive the enemy will move down or right
  - If negative the enemy will move up or left

The **Constructor** for the enemy class is trivial, it does almost exactly what the Player constructor does. The only exception is the last argument that determines if the enemy is moving up/down or left/right. Also note, the sprite positions are different, i looked these values up using Paint.Net for you.

```
public EnemyCharacter(string spritePath, Point pos, bool movingUpDown) : base(spritePath, pos) {
    AddSprite("Down", new Rectangle(68, 112, 29, 30), new Rectangle(101, 112, 29, 30));
    AddSprite("Up", new Rectangle(134, 112, 30, 29), new Rectangle(167, 112, 30, 29));
    AddSprite("Left", new Rectangle(1, 113, 30, 30), new Rectangle(34, 112, 30, 30));
    AddSprite("Right", new Rectangle(201, 112, 30, 29), new Rectangle(234, 113, 30, 29));
    SetSprite("Down");

    moveUpDown = movingUpDown;
    if (!moveUpDown) {
        SetSprite("Right");
    }
}
```

The **Update** method for the enemy is very important. It animates the enemy, moves the enemys position and handles wall collisions. Let's write all of this in one go. I'm going to start you off with a template of the update function that animates and moves the enemy:

```
public void Update(float deltaTime) {
    Animate(deltaTime);

    if (moveUpDown) {
        // Direction is VERY important here. If it's positive the enemy moves up.
        // If its negative the player moves down. This happens because direction can
        // only be 1 or -1, so the code essentially translates to:
        // y += +(speed * delta) OR y += -(speed * delta)
        // depending on the value of direction. Nifty!
        // The alternate to this is to add a bunch of bools (movingLeft, movingRight, etc...)
        // and flip them during collisions and add or subtract from y based on those bools
        // The bool approach is obviously aweful! Just multiplying a positive or -1 is very elegant.
        // Remember this trick, it's going to save you so many if statements down the line.
        Position.Y += direction * speed * deltaTime;

        // TODO: Handle wall collision (4 possible collisions)
    }
    else {
        Position.X += direction * speed * deltaTime;
        // TODO: Handle wall collision (4 possible collisions)
    }
}
```

This enemy is taking shape niceley! If we where to add him to the game right now it would just start walking in a direction and never stop. All we have to do is add wall collision and we are done with the enemy class. Lickly we have a template for collision code! We wrote all collision cases for the player class, all we have to do is copy them over and modify them slightly. I'll get you started with an example. Let's look at what the player character does when the up button is pressed:

```
if (i.KeyDown(OpenTK.Input.Key.W) || i.KeyDown(OpenTK.Input.Key.Up)) {
    SetSprite("Up");
    Animate(deltaTime);
    Position.Y -= speed * deltaTime;
    if (!Game.Instance.GetTile(Corners[CORNER_TOP_LEFT]).Walkable) {
        Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_LEFT]));
        if (intersection.Width * intersection.Height > 0) {
            Position.Y = intersection.Bottom;
        }
    }
    if (!Game.Instance.GetTile(Corners[CORNER_TOP_RIGHT]).Walkable) {
        Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_RIGHT]));
        if (intersection.Width * intersection.Height > 0) {
            Position.Y = intersection.Bottom;
        }
    }
}
```

This is more than what we need. We don't care about keyboard input, and we already took care of moving. All we really need out of that top blob is the collision code, like so:

```
if (!Game.Instance.GetTile(Corners[CORNER_TOP_LEFT]).Walkable) {
    Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_LEFT]));
    if (intersection.Width * intersection.Height > 0) {
        Position.Y = intersection.Bottom;
    }
}
if (!Game.Instance.GetTile(Corners[CORNER_TOP_RIGHT]).Walkable) {
    Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_RIGHT]));
    if (intersection.Width * intersection.Height > 0) {
        Position.Y = intersection.Bottom;
    }
}
```

This is checking for the TOP collision. Meaning if the enemy is walking upwards he will hit the wall and not do anything, just keep walking into the wall. But hey at least there is collision and he won't walk off the screen! Let's add code in there to flip the enemy direction:

```
if (!Game.Instance.GetTile(Corners[CORNER_TOP_LEFT]).Walkable) {
    Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_LEFT]));
    if (intersection.Width * intersection.Height > 0) {
        Position.Y = intersection.Bottom;
        direction = 1.0f;
        SetSprite("Down");
    }
}
if (!Game.Instance.GetTile(Corners[CORNER_TOP_RIGHT]).Walkable) {
    Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_RIGHT]));
    if (intersection.Width * intersection.Height > 0) {
        Position.Y = intersection.Bottom;
        direction = 1.0f;
        SetSprite("Down");
    }
}
```

Our update method should look like this now:

```
public void Update(float deltaTime) {
    Animate(deltaTime);

    if (moveUpDown) {
        // Direction is VERY important here. If it's positive the enemy moves up.
        // If its negative the player moves down. This happens because direction can
        // only be 1 or -1, so the code essentially translates to:
        // y += +(speed * delta) OR y += -(speed * delta)
        // depending on the value of direction. Nifty!
        // The alternate to this is to add a bunch of bools (movingLeft, movingRight, etc...)
        // and flip them during collisions and add or subtract from y based on those bools
        // The bool approach is obviously aweful! Just multiplying a positive or -1 is very elegant.
        // Remember this trick, it's going to save you so many if statements down the line.
        Position.Y += direction * speed * deltaTime;

        if (!Game.Instance.GetTile(Corners[CORNER_TOP_LEFT]).Walkable) {
            Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_LEFT]));
            if (intersection.Width * intersection.Height > 0) {
                Position.Y = intersection.Bottom;
                direction = 1.0f;
                SetSprite("Down");
            }
        }
        if (!Game.Instance.GetTile(Corners[CORNER_TOP_RIGHT]).Walkable) {
            Rectangle intersection = Intersections.Rect(Rect, Game.Instance.GetTileRect(Corners[CORNER_TOP_RIGHT]));
            if (intersection.Width * intersection.Height > 0) {
                Position.Y = intersection.Bottom;
                direction = 1.0f;
                SetSprite("Down");
            }
        }
        // TODO: Bottom collision (2 if statements)
    }
    else {
        Position.X += direction * speed * deltaTime;
        // TODO: Left collision (2 if statements)
        // TODO: Right collision (2 if statements)
    }
}
```

Go ahead and fill out the collision for the other directions of the enemy. Unfortunitaley we can't test our progress at this point as we havent added any enemies to the game yet. Lets go ahead and fix that!

Refactoring Map

Inside of **Map.cs** add a new `List` of enemies to the `Map` class. We are going to use a list, not an array because we don't know in advance how many enemies a map will contain.

Let's add a new function to `Map`, we're going to call this function `AddEnemy` all it's going to do is add a new enemy to the enemy list. It's as straight forward as things get. This will allow us to add enemies to a room even at runtime! So if the player trips a trap an enemy can spawn.

```
public void AddEnemy(string spritePath, Point pos, bool movingUpDown) {
    enemies.Add(new EnemyCharacter(spritePath, pos, movingUpDown));
}
```

Update the **Render** method of `Map` to loop trough all the enemies and call `Render` on them.

Update the **Destroy** method of `Map` to loop trough all the enemies and call `Destroy` on them.

Add a new **Update** method to `Map`, it should take a `float` as an argument (`delatTime`). In the new `Update` method loop trough all the enemies and call `Update` on them.

That's it for now. This is the minimum ammount of code we need to add to support enemies. We will add player collision soon, but first let's get to a state where we can visually confirm that the enemies are working.

Refactoring Game

We added a **Update** method to `Map`. Make sure to call `Udpate` on the **currentRoom** in the `` `Game `` class. Let's add a new constant to game:

```
protected string npsSheet = "Assets/NPC.png";
```

It's a good idea to make all commonly used strings, integers and other constants into variables. This way if you decide that ALL NPC characters should use "Assets/Baddies.png" later, you only have to change the path in one place instead of tracking down every occurance of the old string.

Now, inside of **Initialize**, let's add a couple of enemies:

```
room1.AddEnemy(npsSheet, new Point(6 * 30, 1 * 30), true);
room2.AddEnemy(npsSheet, new Point(1 * 30, 4 * 30), false);
```

That should to it. Run the game and you should see this:

They don't do much yet, but the enemies should walk until they hit a wall and then turn around.

Loosing the game

Before adding collision to the player and enemy, let's add support for loosing the game to **Game.cs**. Add a new member variable, call it **GameOver** with a default state of false. This new variable should be public! Change the update method to respect this new variable.

In **Update**, if `GameOver` is true check to see if the space key was pressed. If the space key was pressed:

- Set game over to false
- Set currentMap to room1
- Set the hero position X to be at the spawn tile
- Set the hero position y to be at the spawn tile

else (if wame over was false) update the game like normal. Let's also update the Render method, if GameOver is true it should print a game over message over the existing screen. By not updating the game state, but still rendering it, we make it look like the game gets paused when there is a game over. It's a cool effect. Here is what the render should look like:

```
public void Render() {
    currentMap.Render();
    hero.Render();

    if (GameOver) {
        GraphicsManager.Instance.DrawRect(new Rectangle(0, 70, 240, 50), Color.CadetBlue);

        GraphicsManager.Instance.DrawString("Game Over", new PointF(70, 80), Color.Black);
        GraphicsManager.Instance.DrawString("Game Over", new PointF(69, 79), Color.White);

        GraphicsManager.Instance.DrawString("Press Space to play again", new PointF(5, 96), Color.Black);
        GraphicsManager.Instance.DrawString("Press Space to play again", new PointF(4, 95), Color.White);

    }
}
```

Enemy collision

Working in **Map.cs**. Let's update the **Update** method of the `Map` class to check for collision against the player.

First, add a new argument to the Update method, it should be of type `PlayerCharacter`. Next, after updating the enemy check if it collides with the player. If it does use the `Game` singleton to set `GameOver` to true.

How do we check if a player an an enemy collide? Both classes inherit the `Rect` getter from `Character` which returns their bounding rectangle. Check if the bounding rectangles intersection area is greater than 0. If it is, an intersection has happened.

```
public void Update(float dt, PlayerCharacter hero) {
    foreach (EnemyCharacter enemy in enemies) {
        enemy.Update(dt);

        Rectangle intersection = Intersections.Rect(enemy.Rect, hero.Rect);
        if (intersection.Width * intersection.Height > 0) {
            Game.Instance.GameOver = true;
        }
    }
}
```

Don't forget to add the exta argument to the function call in **Game.cs**. This is what my entire Update for Game looks like:

```
public void Update(float dt) {
    if (GameOver) {
        if (InputManager.Instance.KeyPressed(OpenTK.Input.Key.Space)) {
            GameOver = false;
            currentMap = room1;
            hero.Position.X = spawnTile.X * 30;
            hero.Position.Y = spawnTile.Y * 30;
        }
    }
    else {
        currentMap = currentMap.ResolveDoors(hero);
        currentMap.Update(dt, hero);
        hero.Update(dt);
    }
```

```
}
```

Run the game and walk into an enemy, you should see a game over message, and pressing space should let you start playing again!

Thats it! We now have support for simple enemies!

Shooting

The enemies can kill the player, but the player can't kill the enemies. Doesn't seem very fair does it? Let's add a simple way for the player to kill an enemy. If during gameplay the player presses the sapce bar he will shoot a bullet that can kill the enemy.

The way to achieve this is actually pretty simple. We're going to implement a bullet class. Whenever space is pressed a bullet will be added to a list. Every frame we loop trough the list. If a bullet has hit a wall we remove it from the list. If a bullet has hit an enemy, we remove the bullet AND the enemy. Simple.

New Project

Let's make a new project, call it **Shoting** and get this project up to par with the **SimpleEnemy** section of the writeup. We're going to work from here.

Bullet class

Let's start by adding the new bullet class. This should be an easy one. Bullet has only two members, position and velocity. The update method of the class should apply the velocity to position over time. The render method should just draw a square around the point at which the bullet is. IF you want try to write this class without looking at the below code:

```
class Bullet {
    public PointF Position = new PointF(0.0f, 0.0f);
    public PointF Velocity = new PointF(0.0f, 0.0f);

    // Eventually everything that renders as a square
    // is going to ned access to its area. Adding a Rect
    // getter is a good habit to get into
    public Rectangle Rect {
        get {
            return new Rectangle( (int)Position.X - 5, (int)Position.Y - 5, 10, 10 );
        }
    }

    public Bullet(PointF pos, PointF vel) {
        Position = pos;
        Velocity = vel;
    }

    public void Update(float deltaTime) {
        Position.X += Velocity.X * deltaTime;
        Position.Y += Velocity.Y * deltaTime;
    }

    public void Render() {
        Rectangle renderRect = new Rectangle(0, 0, 10, 10);
        renderRect.X = (int)(Position.X - 5f);
        renderRect.Y = (int)(Position.Y - 5f);

        GraphicsManager.Instance.DrawRect(renderRect, Color.Red);
    }
}
```

Game Refactor

To actually shoot / use bullets we need to refactor the **Game** class a little bit. First thing is first, add a new member

variable, let's call it **projectiles** its type is going to be `List<Bullet>`, don't forget to initialize this new list.

In the **Render** function, after rendering the hero, but before rendering the game over screen, loop trough every projectile in projectiles and call render on all of them.

In the **Update** method, after updating the hero, but before updating the map, loop trough all theprojectiles and call `Update` on each of them, passing in deltaTime.

At this point we still can't see anything on screen. We need to add some code to actually shoot the projectile. Above the loop that updates all the projectiles, add an if statement. In this if statement check if the space key was pressed. If it was, use the hero objects `currentSprite` variable to determine which direction the velocity needs to face. Velocity should be either 90, or -90, that is 3 tiles per second.

Once you know which way the velocity goes, add a new bullet at the center of the player. Like so:

```
if (InputManager.Instance.KeyPressed(OpenTK.Input.Key.Space)3) {
    PointF velocity = new PointF(0.0f, 0.0f);
    if (hero.currentSprite == "down") {
        velocity.Y = 100;
    }
    else if (hero.currentSprite == "up") {
        velocity.Y = -100;
    }
    if (hero.currentSprite == "left") {
        velocity.X = -100;
    }
    else if (hero.currentSprite == "right") {
        velocity.X = 100;
    }
    // projectiles is the name of my member variable that is of type List<Bullet>
    projectiles.Add(new Bullet(hero.Center, velocity));
}
```

Run the game, every time you press space there shoudl be a new projectile. Each projectile shoots, and eventually makes it off-screen. You may have notices that we don't have a Destroy function for the projectiles, that's because they don't need a Destroy function. The projectiles just draw squares, if no texture / resources are being held onto then a resource class is useless.

Map Refactor

Becuse the **Map** class knows about both walkable / non-walkable tiles AND enemies, it seems as good a place as any to add logic that will make bullets dissapear. There are two instances where bullets will dissapear, if they hit a non walkable tile; or if they hit an enemy.

Inside the **Update**method of the ``map `class. Add a new artument. A` List `, call this argument` projectiles```. You want to loop trough this array before looping trough the enemies array. Remember, you are going to be removing items for the array, loop accordinly.

Inside this loop:

- Find the xTile of the projectile
- Fint the yTile of the projectile
- If the xTile is out of bounds
  - remove the loop index (i?) from the projectiles array.
- ELSE If the yTile is out of bounds
  - remove the loop index (i?) from the projectiles array.

- ELSE If the tile at [yTile][xTile] is not Walkable
    - remove the loop index (i?) from the projectiles array.

Remember, you can use the `RemoveAt` function of a vector to remove an item at a specific location. Next, let's update the way you check for enemu collision. Right now you have a loop counting up. Make this loop count down, we are going to potentially be deleting specific enemies from the `enemies` array.

This next bit can get tricky. After the game over check, loop trough all the projectiles. You probably want to loop in reverse, we might remove things from this array.. For every projectile, see if it intersects with the current enemy. If it does, remove the enemy, and remove the projectile.

Hint, at this point the game might break. That is because even tough the enemy was removed we might still be checking for collision against the projectiles for the removed enemy. Therefore when a projectile and enemy collide, break out of the inner (j) loop.

That's it!

Run the game you are now a bad-ass who can walk around and shoot monsters just for the fun of it!

Getting Items

We can walk around, switch rooms and even kill enemies. It's almost like we have a game starting to shape up! In the last section we added bullets, now let's add the opposite. Items! Our items are going to be simple. The player will have a running score, collecting an item will add some number (item dependant) to that score.

Why do i call items the opposite of bullets? Because they don't interesect the enemies, they intersect the player! You will find much of the code is going to be similar to how bullets work. I still think an item is a non moving anti-bullet.

New Project

Let's make a new project, call it **GettingItems** and get this project up to par with the **Shooting** section of the writeup. We're going to work from here.

Getting Started

Let's get started by updating the **HouseTiles.png** texture atlas with a new version that i've added some items to:

Item class

Let's go ahead an create a new class for items. If this was a real game,we might create a class called Item, then a

subclass that into various other subclasses like powerup, weapon, secret, and tose further into specific subclasses like sword, potion, map, etc... But this is just a simple demonstration of how to pick items up, so we will only have a simple Item class.

This class will have 4 member variables:

- int **Sprite**
  - Which sprite sheet id is the sprite on
- Rectangle **Source**
  - What part of the sprite sheet to draw
- int **Value**
  - How many points the item is worth
- Point **Position**
  - Where on screen to draw the sprite of the item

The constructor will take 1 argument for each of these, and simply set the appropriate values. We will have a **Render** function that just draws the subsprite to screen. Because this class holds on to a texture reference we must add a **Destroy** function, in which the textur emanager unloads textures. For good measure we also implement a Rect getter, the X and Y are provided by Position while the width and height are provided by Source. Try to implement the class yourself first, here is my implementation:

```
class Item {
    protected int Sprite;
    protected Rectangle Source;
    public int Value { get; private set; }
    public Point Position;

    public Rectangle Rect {
        get {
            return new Rectangle(Position.X, Position.Y, Source.Width, Source.Height);
        }
    }

    public Item(string spriteSheet, Rectangle sourceRect, int value, Point position) {
        Sprite = TextureManager.Instance.LoadTexture(spriteSheet);
        Source = sourceRect;
        Value = value;
        Position = position;
    }

    public void Render() {
        TextureManager.Instance.Draw(Sprite, Position, 1.0f, Source);
    }

    public void Destroy() {
        TextureManager.Instance.UnloadTexture(Sprite);
    }
}
```

Refactoring Map

Just like we have unique enemies in each room, we have unique items in each room. Therefore it makes sense to make the `Map` class be the owner of the items list. We need a list because we don't know how many items we're going to have in advance, but more importantly because we're going to remove items as they are collected.

Make a new **List** of type **Item** in **Map.cs**, i call mine `items`. Make sure to initialize this list.

In **Render**, loop trough all of the items and call Render on each one.

In **Destroy**, loop trough all of the items and call Destroy on each one.

Inside the **Update** function, AFTER the loop that goes trough all the enemies, add a loop that loops trough all the items. We're going to potentially delete stuff from the list, so loop appropriately. Loop trough every item, and check for collision against the hero. If a collision occurs, remove the item. (At this point we just want to remove it, we're not keeping track of score just yet.

When you remove an item from the list it will no longer be referencable. Therefore it will no longer be auto unloaded by Destroy. Make sure to call the items `` `Destroy `` function before reoving it from the list.

Now, when we created enemies, we added a very helpful **AddEnemy** function. Do the same for items. This function should of course take the same arguments as an item constructor. Call this new function **AddItem**.

Refactoring Game

In **Game.cs** find where we added the enemies to the rooms. Use the following code to add items:

```
room1.AddItem(spriteSheets, new Rectangle(350, 255, 16, 16), 10, new Point(3 * 30 + 7, 2 * 30 + 7));
room1.AddItem(spriteSheets, new Rectangle(381, 256, 13, 15), 20, new Point(5 * 30 + 7, 4 * 30 + 7));
room2.AddItem(spriteSheets, new Rectangle(412, 256, 16, 15), 30, new Point(4 * 30 + 7, 2 * 30 + 7));
```

Run the game you should have items in the roomw. You should also be able to walk around and pick up said items! Here is what the first map looks like:

Lets add the ability to view a score that increases when items are piced up. Add a public integer to the `Game` class, call it **Score**. Add the follwoing at the end of the **Render** function (outside any if statements) to display the new score:

```
GraphicsManager.Instance.DrawRect(new Rectangle(150, 0, 90, 20), Color.CadetBlue);
GraphicsManager.Instance.DrawString("Score:" + Score, new PointF(155, 3), Color.Black);
GraphicsManager.Instance.DrawString("Score:" + Score, new PointF(154, 2), Color.White);
```

Adding up the score

All that's left is to add up the score. Go back into **Map.cs**, find where we Destory and Remove items on collision with the player. In this same block add one to the `Score` variable of game. You can access the Score variable using the games singleton instance.

Run the game you should now be able to collect items for points! That's all there is to it!

Scrolling

So far out game has taken place inside of small rooms, we've discussed adding an overworld map where different screens are conneced trough hidden doors. This time let's make a proper overworld, one that scrolls!

Many RPG games like the early Final Fantasy or Fire Emblem games used overworld scrolling like this. Some early games like Super Mario also implemented scrolling.

There are two ways to approach scrolling. Move the world and shift the view. Lets take some time to talk about each method:

Move The World, like the name implies moves the entire world! In this method if you want to move the hero to the right you update the position of EVERYTHING, including the hero in the world to be moved the opposite direction of the hero (to the left), then move the hero to the right.

Because everything shifted left, but only the hero shifted right you get the illusion tha the hero is in the center and the world scrolls around him. As you can imagine this method will quickly fall appart with LARGE (Over 1000 tiles) game worlds becuase the cost of updating everything grows with each object added.

Shift the view: This method calulates how much offset the player would need to appear in the center of the screen. When it comes time to render anything (beit the player a tile or an enemy) that difference is subtracted from the visual position.

Note, only the visual position is changed, the world position and update logic remain in tact. This method is preferable because it doesn't break anything we've done so far, it's strictly visual.

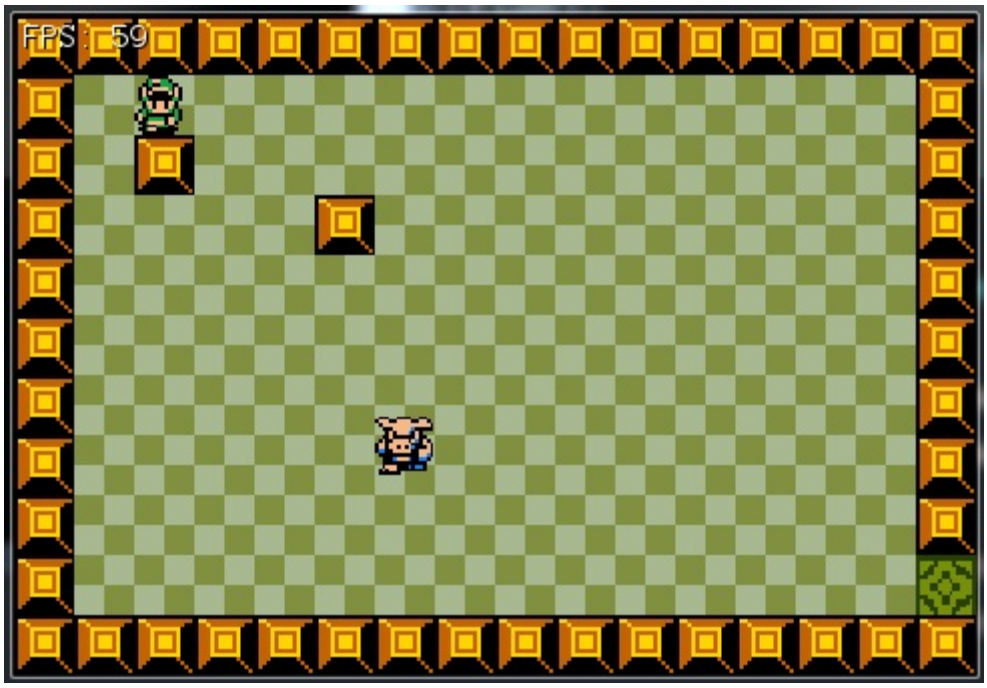
Needless to say **we are using the shift view method**, which coincidentally is how a 3D camera works (Just with a z-dimension added)

Concept of a camera

This might be a bit confusing, but i'm going to try to explain the concept of a camera in a game. If it makes no sense, give me a call and we will talk about it. Don't worry, we're not making a camera class or anything, this is just to try to clear things up.

The way a camera works is kind of like in real life. We place a camera at a position in the world, then we see what the camera sees. The details of how this works are a bit more involved. In order to understand cameras we have to unserstand spaces.

World Space: World Space refers to all of the objects in their world position. For instance, when we put link at xPixel 90, yPixel 30, his world position is Point(90,30). Here is an image of all our game objects in world space:



Camera Space: Refered to as **View Space** if talking about 3D graphics, **Camera Space** if talking about 2D graphics. This is what your camera sees! Camera space is usually a subset of world space. Think of this as a shifted coordinate system. Think of it like this:

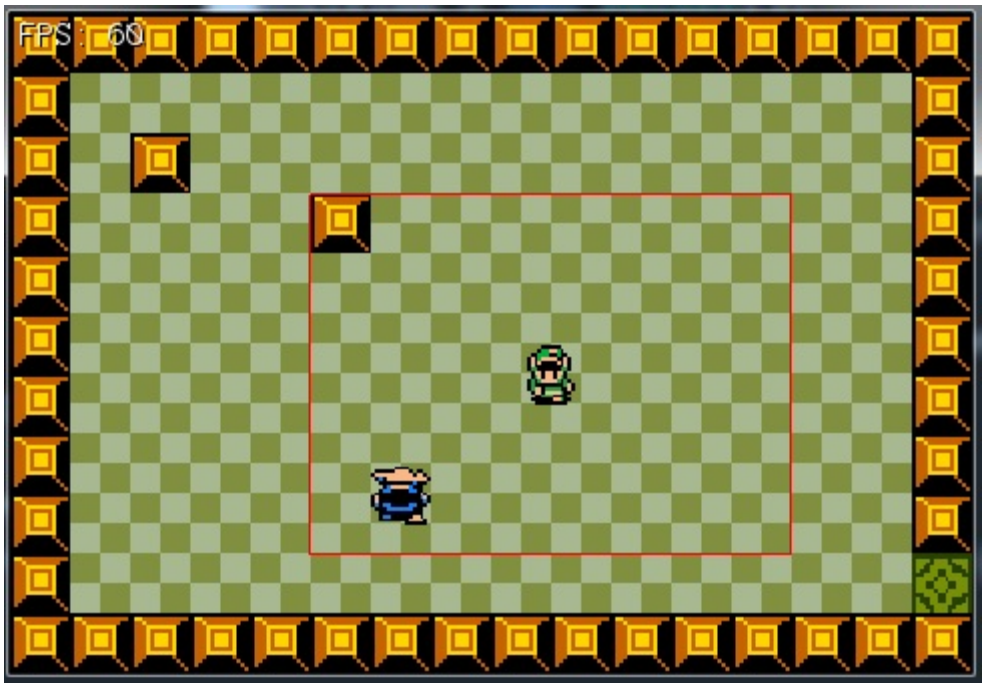
- Tile 0,1 is at position 0,1 in World Space
- The camera is located at positon 3, 2 in world space
- Relative to the camera, tile 0,1 is located at -3, -1
- Tile is located at -3,-1 in camera space

Why -3, -1? The camera has it's own reference point for where 0,0 is. This happens to be +3, +2 from world space. To go from world space to camera space, we must subtract the cameras world position -3, -2. From 0,1 if we take 3 away from the X and 2 away from the Y we are left with -3, -1.

That might sound complicated, but there is a simple formula to transform an object from world space to camera space:

```
Object Position In Camera Space = Object Position in World Space - Camera Position In World Space
```

This is what camera space for our game looks like (assuming a camera that is 8 tiles wide and 6 tiles tall):



The red square is camera space. Notice how the obstacle on the top left is at tile 5, 4 relative to the world, but at tile 0, 0 relative to the camera.... That's world space vs camera space! For us the camera is going to be centered around the character, it's position is tied to character position.

Screen Space, this is what actually appears on screen. Up until now we have seen the entire sceen! Really all **Screen Space** is just what you see on screen. For the most part we just want to see the bit of camera on screen. We want to chop off anything outside the camera view, and that's screen space. Here is a screenshot:

Implementation

Let's go ahead and actually implement scrolling. This should be pretty simple, much less complex than the theory that goes behind it.

The first thing to do is to **make the map bigger**. Go ahead and make room1 two, or maybe three times bigger. Running the game you will see that the screen resizes to fit your new room size. Go into the `Initialization` function of game and make the window size be only 8 by 6 tiles:

```
window.ClientSize = new Size(8 * tileSize, 6 * tileSize);
```

Cool, now you should be able to walk off screen. The rest of this section is rendering related. In **Game.cs** find your render function. The first thing we need to do is determine a world to camera space transformation. We want the

camera's to be centered around link. That means we want it's upper left to be 120 pixels (4 *tileSize) to the left and 90 pixels (3* tileSize) to the top. This will center link.

This world to camera transformation is going to be stored as a `PointF`. I'm going to call mine `offsetPosition`. Simply make a new `PointF` at the heros center and subtract half of the window width and height:

```
PointF offsetPosition = new PointF();
offsetPosition.X = hero.Position.X - (float)(4 * tileSize);
offsetPosition.Y = hero.Position.Y - (float)(3 * tileSize);
```

Now comes a bit of refactoring. Pass offsetPosition in to every render function that the **Game**'s render function calls. Next track down each of these functions, and add a `PointF offsetPosition` argument to them. Some functions call Render on other objects, Like the **Map** object calls **Render** on both EnemyCharacter and Tile. Add the argument to each of these as well. Every render function should take a PointF offsetPosition as an argument!

Inside each render function offset the X and Y positions of what is being rendered by the offsetPosition being passed in (subtract the camera offset from the x and y of what would be rendered). For example, this is what the **Render** function of `Character` becomes.

```
public void Render(PointF offsetPosition) {
    // This is where we would render in world space
    Point renderPosition = new Point((int)Position.X, (int)Position.Y);

    // Apply the camera offset, bring us to rendering in camera space
    renderPosition.X -= (int)offsetPosition.X;
    renderPosition.Y -= (int)offsetPosition.Y;

    // Draw the character
    TextureManager.Instance.Draw(Sprite, renderPosition, 1.0f, SpriteSources[currentSprite][currentFrame]);
}
```

Go ahead and implement this offsetting to every function. After adding this code to every render function **run the game**. Now the game should scroll around with link! This is scrolling in it's most basic form! If you have any questions, give me a call.

Optimization

There is one small optimization problem. Even tough we only see a small section of the map at a time ALL of it is rendered! Even the bits that are off-screen. You can confirm this by resizing the window, notice that ALL of the off-screen bits of the map are rendering here. This isn't an issue until we get HUGE maps, on the scale of thousands by thousands of tiles.

How can we optimize this? By rendering only what can be seen. This bit of code is going to go into the `Map` class, because after all the Map class is responsible for rendering tiles. Unfortunitaley, we don't have enough information to do this at the moment! We could make some assumptions and figure out the bounds of the render rect, but that's a lot of effort. We're just going to add function arguments.

Find the **Render** function of `Map`, add a new `PointF` argument, call it cameraCenter. Next, we need to make 4 integers inside the function, Camera Min and Max X and Y. We're going to figure out the minimum X and Y pixels that are visible on screen. This is a bit hard to explain, but you basically take the center point, subtract half of the tiles (plus one tile for good measure) and you have the min. Now convert these to tile coordinates. Do the opposite for max. Here is how i did it in my code:

```
public void Render(PointF offsetPosition, PointF cameraCenter) {
    // Find the visible corners of the screen in pixel position
    int minX = (int)cameraCenter.X - 4 * 30 - 30;
    int minY = (int)cameraCenter.Y - 3 * 30 - 30;
    int maxX = (int)cameraCenter.X + 4 * 30 + 30;
    int maxY = (int)cameraCenter.Y + 3 * 30 + 30;

    // Convert visible corners to tile indexes
    minX /= 30;
    minY /= 30;
    maxX /= 30;
    maxY /= 30;
```

Finally, right under this bit of code is nested a for loop. This for loop goes trough all the tiles and render them. Change this loop so instead of looping from 0 to the number of rows or columns it loops from Min to Max. It's worth noting that Min and MAx might be out of bounds, so be sure to add bounds checks. Here is what mine looks like:

```
for (int h = minY; h < maxY; h++) {
    for (int w = minX; w < maxX; w++) {
        // Lower bounds check
        if (h < 0 || w < 0) {
            continue;
        }
        // Upper bounds check
        if (h >= tileMap.Length || w >= tileMap[h].Length) {
            continue;
        }
        tileMap[h][w].Render(offsetPosition);
    }
}
```

Thats it! Now we're only rendering what is visible on screen! Want to confirm it? Resize your window! Make it larger and take not how only the area which is visible is being rendered. And that's it. We're done with basic scrolling.

Scrolling with limits

The way we have scrolling implemented right now works fine, but there is one small thing that might (or might not) bother you about it. The edge of the scren! There are three ways games deal with this:

- Add obstacles to edge of screen so there is no black space
- Render black (or other solid color) in blank space (what we do now)
- Stop camera at edge of screen

In this section we are going to try to stop the camera at the edge of the screen. This means you will be able to walk from wall to wall, and see no background!

The implementation of this is decievingly simple, it's all done in `Render` method **Game.cs** before anything is actually rendered! All we do is clamp the offsetPosition variable (camera space transformation) in four edge cases. When we calculate offet position:

- If the hero's world X position is less than 1/2 of the screen with
  - Clamp the xOFfset to 0
- If the hero's world Y position is les than 1/2 the screen height
  - Clamp the yOffset to 0
- If heros world X is greater than the world width minus 1/2 screen width
  - Clamp xOffset to world width minus screen width
- If heros world Y is greater than the world width minus 1/2 screen height
  - Clamp yOffset to world height minus screen height

As you can see, all we're doing is not moving the offsetPosition after the hero passes a given point. Because offsetPosition is what transforms world space to camera space this essentially pegs the camera in spots so no corner of the map is off-screen. This is what my math looks like:

```
// If the hero is less than half the camera close to the left or top corner
if (hero.Position.X < 4 * tileSize) {
    offsetPosition.X = 0;
}
if (hero.Position.Y < 3 * tileSize) {
    offsetPosition.Y = 0;
}

// If the hero is less than half the camera close to the bottom or right corner
if (hero.Position.X > (currentMap[0].Length - 4) * tileSize) {
    offsetPosition.X = (currentMap[0].Length - 8) * tileSize;
}
if (hero.Position.Y > (currentMap.Length - 3) * tileSize) {
    offsetPosition.Y = (currentMap.Length - 6) * tileSize;
}
```

If you **run the game** there is going to be a visual error. When you are near the edge of the screen one or more of the tiles might be missing! Like this:

This happens because of **Map.cs**, it limits which tiles are rendered and which are hidden. But it only lets 4 tiles to the left or right. We basically need to double the ammount of visible tiles, this is because we can now walk to the edge of the screen. So find the bit of code that hadnles how much is visible, this code:

```
// Find the visible corners of the screen in pixel position
int minX = (int)cameraCenter.X - 4 * 30 - 30;
int minY = (int)cameraCenter.Y - 3 * 30 - 30;
int maxX = (int)cameraCenter.X + 4 * 30 + 30;
int maxY = (int)cameraCenter.Y + 3 * 30 + 30;
```

And bump p the visible area to be the entire screen, like so:

```
// Find the visible corners of the screen in pixel position
int minX = (int)cameraCenter.X - 8 * 30 - 30;
int minY = (int)cameraCenter.Y - 6 * 30 - 30;
int maxX = (int)cameraCenter.X + 8 * 30 + 30;
int maxY = (int)cameraCenter.Y + 6 * 30 + 30;
```

Run the game now and you should be able to walk around flawlesly!

Depth

We want to add some depth to our game. Right now, having a purley top-down game depth isn't all that important. But as we experiment with different perspectives (Including isometric) depth is about to become very improtant.

As you will see, there are two ways of handling depth, with a z buffer or with the painters algorithm. Neither method is better or worse, for the most part which one you use is your preference.

I prefer to use z buffers myself, they requie a lot less code to write and is the method used when dealing with 3D graphics.

Adding depth to the tile based framework we have set up is going to be the hardest thing we've done so far (In my opinion) as such we're going to have to break this task into three sections. Each section will have it's own project.

First we're going to seperate out collision rectangles from rendering rectangles. This will allow us to draw character and tile sprites outside of their respective bounding boxes. If a tile is 45 pixels tall (instead of 30), the character will be able to walk behind the top 15 pixels.

Second we're going to implement sprite sorting using z buffers. This will require an update to the `GraphicsManager` class. z buffering is pretty simple to set up and takes very little code, this should be a quick fun exercise.

Third we're goin to implement the painters algorithm. This is going to build on the first section (Not the z buffered section). Implementing the painters algorithm is a bit difficult becuase it requires a shift in paradime for our code.

Moving forward

After having read all three of the sub sections for depth, which method you use moving forward (For things like isometric tiles) is going to be compleatly up to you.

Ideally, once sorting is set up the code shouldn't need to be touched so the mthod you choose should not have too much of an impact moving forward.

I would urge you to use z buffering for the simple reason that it requires a LOT less code than the painters algorightm. In my experience less code means easyer debugging.

Seperating Collision & Rendering

The first thing we need to do in order to add depth to our game is to seperate the collision rectangle of characters and tiles from their rendering. This will give us the unsorted illusion of depth. By the end of this chapter we want to achieve this effect (Note how links head is above the tile):

New Project

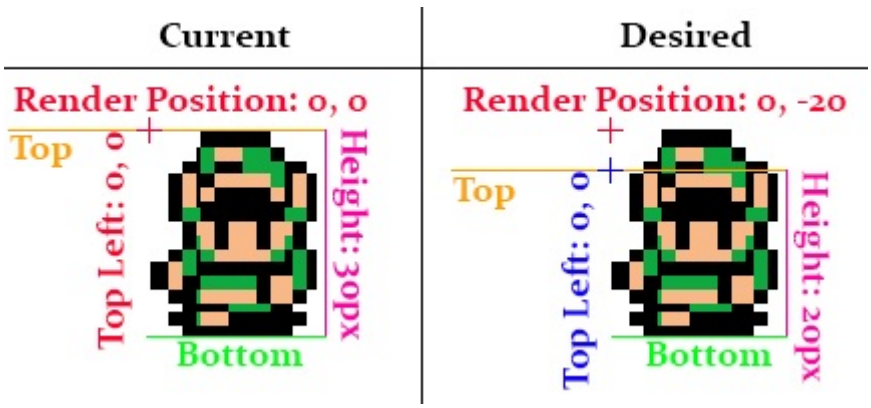
Let's make a new project, call it **DepthBase** and get this project up to par with the **LimitedScrolling** section of the writeup. We're going to work from here.

Character

We're going to start by changing the scrolling behaviour of the character class. Starting with the character is nice, we get the effects of it applied to both the player character and enemy character! We're only going to add depth on the Y axis for now, it's rare to see depth on the X axis.

We are going to hard code the height of the character to some constant number (In this example 20px). Whenever the corners or bounding rectangle of the character is requested we're going to change the return to have 20 hard coded to it. Then we're going to change the render function, we're going to change it to render the sprite at Position.Y - 20 instead of Position.Y.

These changes will cause a 20px gap at the top of our characters. This part of the sprite will still be drawn, but it won't be taken into collision calculations. I made a graphic atempting to describe what we will be doing:



Debug Rendering

Before we start coding anything, let's make our lives easyer by adding a little debug indicator to the `Character` class. At the end of the **Render** function, add this bit of code:

```
// Get a 3x3 rectangle, centered on the top left pixel
Rectangle positionLocator = new Rectangle((int)Position.X - 1, (int)Position.Y - 1, 3, 3);
// Apply camera offset
positionLocator.X -= (int)offsetPosition.X;
positionLocator.Y -= (int)offsetPosition.Y;
// Draw indicator
GraphicsManager.Instance.DrawRect(positionLocator, Color.Yellow);
```

This should draw a 3x3 yellow rectangle centered at the top left pixel of the character. If you run your game you should see this:

Applying Depth

First, we have to add a new member variable to the `Character` class. Make it a protected **float** and call it `Height`. Set it to **-1** by default. If the new variable is less than 0 we will not apply the depth effect to the character, if it is greater than 0 we will apply a depth effect.

Next, change the constructor of character, it should now take in a third argument, a float called height. Set the member variable accordingly. Since we changed the constructor, we need to change where the constructor gets called. Find the following code in both `PlayerCharacter` and `EnemyCharacter`:

```
: base(spritePath, pos) {
```

In `PlayerCharacter` pass **20** as the third argument, in `EnemyCharacter` pass **-1** as the third argument. By using constants as the last argument of the base constructor we can avoid changing the constructor of `PlayerCharacter` and `EnemyCharacter`. Needless to say, the player will have a depth effect, while the enemy will not.

Change the **Rect** getter in the `Character` class. If the new `Height` variable is >= 0, set the returned rectangles height to the varaible.

Change the **Corners** getter. Right now it uses this bit of code to add height to corners `SpriteSources[currentSprite][currentFrame].Height`, if the new `Height` variable is >= 0, use the `Height` variable, if it's less than 0 keep using what was already there.

Finally, change the **Render** function. After subtracting `offsetPosition` from `renderPosition`, if the new `Height` variable is >= 0 find the height difference of the frame height and the hard coded height:

```
int difference = SpriteSources[currentSprite][currentFrame].Height - (int)Height;
```

and subtract it from from `renderPosition.Y`. That should do it. **Run the game** and you should be able to walk up to walls and overlap them a bit. Links head should be above the yellow debug indicator. our game should look like this:

Notice how the moblins yellow indicator pixel is at his top left, while link's is slightly below his head. Also, links head slightly overlaps the obstacle. This is what we expected.

Tiles

The character having depth is awesome, but we're going to want the tiles to have depth too! Unlike character however we always know the height of a tile, it's 30! Because we already know the height of the tile, all we need to change is the `Render` function in **Tile.cs**.

This is what the current tile render function looks like (I've added comments to help readability):

```
public void Render(PointF offsetPosition) {
    // Find the world render position of this tile
    Point renderPos = new Point(WorldPosition.X, WorldPosition.Y);
    // Apply scale to tile offset
    renderPos.X = (int)(Scale * renderPos.X);
    renderPos.Y = (int)(Scale * renderPos.Y);
    // Move the tile into camera space
    renderPos.X -= (int)offsetPosition.X;
    renderPos.Y -= (int)offsetPosition.Y;
    // Draw the tile
    TextureManager.Instance.Draw(Sprite, renderPos, Scale, Source);
}
```

After moving the tile into camera space, make a new rectangle, call it renderRect and copy Source into is, then use this new renderRect to draw instead of Source (Change last argument). You can copy one rect into another with only two arguments like so:

```
Rectangle renderRect = new Rectangle(Source.Location, Source.Size);
```

After creating renderRect, check it's height. If renderRect.Height **is not equal to** 30 we will want to offset the tile render position.

To offset the render position, first find the `heightDifference` by subtracting 30 from renderRect.Height, i suggest storing this in a variable. Then, subtract `heightDifference` from `renderRect.Y`.

That's it! Tiles now have the same depth support that the character does. We will not be able to test this right now for two resons. First, we don't have any tiles that are taler than 30 pixels.

Second, because we draw all the tiles first, then the character on top, we would never see the character behind a tile. We're going to fix this in the next section.

Visual Tweaks

Before moving on to the next section i feel like we need to do some house cleaning. There are a few things i want to change in the code in order to add houses that have depth to them visually. Also, while reviewing your code i found a bug that was made because i gave bad instructions in the last tutorial section.

Do all the work for this section in the **DepthBase** project from the last section.

Let's start off by updating our **HouseTiles.png** to the following image:

Edit `spriteSources` in **Game.cs** to include some of the new tiles. I went ahead and measured these out for you:

```
protected Rectangle[] spriteSources = new Rectangle[] {
    /* 0*/new Rectangle(466,32,30,30),
    /* 1*/new Rectangle(466,1,30,30),
    /* 2*/new Rectangle(32,187,30,30),
    /* 3*/new Rectangle(32,280,30,40), // Top-left
    /* 4*/new Rectangle(63,280,30,40), // Top-middle
    /* 5*/new Rectangle(94,280,30,40), // Top-right
    /* 6*/new Rectangle(32,321,30,30), // Middle-left
    /* 7*/new Rectangle(63,321,30,30), // Middle-middle
    /* 8*/new Rectangle(94,321,30,30), // Middle-right
    /* 9*/new Rectangle(32,352,30,30),
    /*10*/new Rectangle(63,352,30,30)
};
```

Also, change the layout of room 1 to include the new building. Like so:

```
protected int[][] room1Layout = new int[][] {
    new int[] { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    new int[] { 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 },
    new int[] { 1, 3, 4, 5, 0, 0, 0, 1, 0, 0, 0, 0, 1 },
    new int[] { 1, 6, 7, 8, 0, 1, 0, 0, 0, 0, 1, 0, 1 },
    new int[] { 1, 9,10, 9, 0, 0, 0, 0, 0, 0, 1, 0, 2 },
    new int[] { 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1 },
    new int[] { 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1 },
    new int[] { 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1 },
    new int[] { 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1 },
    new int[] { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }
};
```

After having made these changes **Run the game**, you will notice two errors. First, the bannana is stuck in a building wall. Second, the roof of the building is rendered wrong! This is what it looks like for me:

The first one is easy, find where we add the bannana and move it's x index from 3 to 4. Fixing the rendering artifact is a little more involved as we have to build a mental model of what's wrong. First, open up **Tile.cs** and take a look at this section of the `Render` method:

```
Rectangle renderRect = new Rectangle(Source.Location, Source.Size);
if (renderRect.Height != 30) {
    int difference = renderRect.Height - 30;
    renderRect.Y -= difference; // This is a bug!
}
```

The line where we change renderRect.Y is inheritly wrong. This is my fault, i told you to change the wrong variable in the last tutorial. What you actually want to change is `renderPos.Y`. It makes sense when we thing about it, renderRect is not the position on screen, it's the rectangle to use from the sprite sheet. We don't want to mess with the source sprite, only the render position!

Running the game now, it's still messed up. The height and y of the tile is just not set properly! So let's track down where the height and Y are configured, take a look at the `constructor` of **Map.cs**. The issue is in this block of code:

```
for (int j = 0; j < layout[i].Length; j++) {
    Rectangle source = sources[layout[i][j]];

    Point worldPosition = new Point();
    worldPosition.X = (int)(j * source.Width);
    worldPosition.Y = (int)(i * source.Height);
    result[i][j] = new Tile(sheets, source);
```

```
            result[i][j].Walkable = false;
            result[i][j].IsDoor = false;
            result[i][j].WorldPosition = worldPosition;
            result[i][j].Scale = scale;
            foreach (int w in walkable) {
                if (layout[i][j] == w) {
                    tileMap[i][j].Walkable = true;
                }
            }
        }
```

The problem is where we calculate worldPosition.X and worldPosition.Y. We multiply i and j by the source tile width and height. This of course assumes that all tiles are going to be uniform sized, an assumption we just broke! Instead we need to hard code the tile size, like so:

```
 Point worldPosition = new Point();
worldPosition.X = (int)(j * 30);
worldPosition.Y = (int)(i * 30);
```

There is one last think i'd like to address. If you collect an item and close the game you get a yellow warning in the console about an unreleased texture. This only happens when you collect an item. That tells me that at the point when we remove the item from the map, you don't release it's texture. Go ahead and fix this.

Now we're good to go to the next section! If you run the game you will be able to walk on the top 10 pixels of one of the houses, this is expected behaviour. In the next section we're going to add proper depth sorting to make it look like you are behind the house. Until then, this is what your game will probably look like at this point:

Z-Buffer

The concept of a Z buffer is pretty simple. Everything in the game has thre component coordinates: x, y and z. The game has two buffers, a color buffer and a depth buffer. When you try to draw something, the first thing that happens is the z value is written to the depth buffer. The depth buffer is a black and white image, low z values are black, high z values are white.

So, the z value is written to the z buffer, the thing is, not every pixel might be written to the z buffer! If the object is partially occluded (if the buffer has lighter pixels over a part of the object), then those values are not written to the z buffer. This is called rejecting the pixel. It's also called a z test.

If a pixel fails the z test it is not written into the color buffer. If a pixel passes the z test, it is written into the color buffer.

This is the general concept of a z-buffer. It's a pretty important concept, if you are having trouble with it give me a call and we can talk it oer.

New Project

Let's make a new project, call it **DepthBuffer** and get this project up to par with the **VisualTweaks** section of the writeup. We're going to work from here.

Update Graphics Manager

GraphicsManager does not support all the functionality you need to get the project off the ground right now. You can fix this by grabing the latest manager from my github and pasting it into your file.

Existing Depth Buffer

The graphics frameworks is already using buffers under the hood. When you call `GraphicsManager.Instance.ClearScreen(System.Drawing.Color.CadetBlue);` in **Program.cs** the depth buffer is cleared to 0 (Pure white).

Every sprite you draw renders as a quad. Each time you call `GraphicsManager.Instance.Draw[String/Square/Line]` or `TextureManager.Instance.Draw` 0.0005f is added to the depth, and the quad is rendered into the depth buffer with the new depth. The closer we get to 1, the darker the depth buffer.

The depth test passes, and the colors of the quad are drawn to screen.

Right now the depth test will always pass! This is becuase each time a new quad is rendered it is rendered with a z valu 0.0005f higher than the last one. That is, nothing in the depth buffer will have a higher value (be darker) than the current thing being drawn.

Here is a screenshot of what our z buffer looks like. Notice how the tiles get darker not only going from top to bottom, but also left to right.

Just for fun, if you want to see the z-buffer for your self, you can add this code to **Game.cs**

```
if (InputManager.Instance.KeyPressed(OpenTK.Input.Key.Y)) {
    Image depthBuffer = GraphicsManager.Instance.GetDepthBuffer();
    depthBuffer.Save("depth_buffer.png", System.Drawing.Imaging.ImageFormat.Png);
}
```

With this code in place, every time you press the **Y** button the depth buffer will be saved next to the Asets directory and solution file.

Explicit Depth Buffer

The task at hand is to figure out how to sort tiles, enemies, characters and items; then to apply Z values directly. Right now we don't have controll of the z buffer directly, which is why i updated the Graphics manager with the following functions

```
public void SetDepthRange(float near, float far)
public void IncreaseDepth()
public void IncreaseDepth(float step)
public void SetDepth(float depth)
```

- **SetDepthRange** - Unlike the X and Y axis, our depth is not infinite! We need to define a range for it
  - Takes a near value and a far value. Configures depth buffer to use said values
- **IncreaseDepth** - This is what GraphicsManager has been using internally this whole time.
  - Increases current depth by 0.0005f
- **IncreaseDepth(float)** - The stock increase might not be enough. We may want to increase depth in smaller steps
  - Increases depth by the value of the argument provided
- **SetDepth** - Increase depth can only go forward, we may need to set depth to 3 after it was 5
  - Allows us to hard code the depth value for the next draw call

Out of all these functions, we only need SetDepth. Before rendering a tile or the character or anything we're going to explicitly set it's z-depth by calling **SetDepth**. This way we can be in total controll! The only question is, how to we figure out the depth of a thing (tile, character, item, etc...)??? Oh, and what resolution depth buffer do we need?

The image of the depth buffer we saw in the last section on this page actually gives us a pretty decent strategy for dealing with depth. The further down something is the higher it's depth. The further right something is, the higher it's depth!

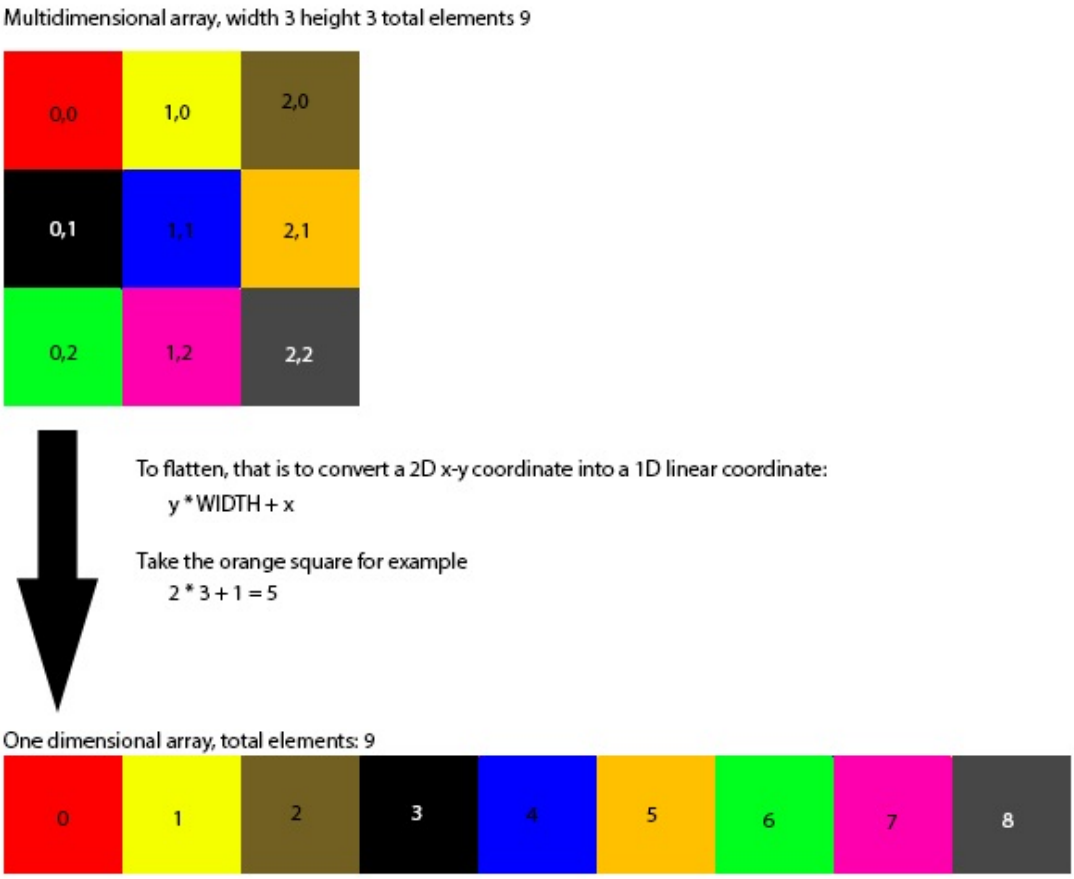
Here is what i propose. We will pick two numbers, one for the maximum height we might render and one for the maximum width we might render. For demonstration purposes, lets say height of 6, width of 7. Our depth range is

going to be height *width + 1 or 6* 7 + 1. The depth of each indevidual tile? **y * 7 + x**. 7 being the width. This ensures that each row starts on a multiple of 7, and we just increase on the way up.

Here is a visual demonstration using a height of 8 and a width of 10 (even tough only a width of 8 is displayed). It's ok to have extra tiles, a little saftey padding never hurt anybody.

0	1	2	3	4	5	6	7
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	32	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Here is another example of what we are trying to do:



Real World Values

So now that we know how to get the depth of each tile, and the depth range, what should it be for our game?!?!? Take a look at the **Render** function of **Map.cs** our width rendering ranges from -8 to +8, and our height rendering ranges from -6 to + 6. We also add one extra tile (the + 30) for padding. This means we have 17 tiles accross and 13 tiles up and down.

Let's round those numbers up to the nearest ten. Make the **depth range = 0, 21 x 21**. The depth function for each tile is going to be **y * 20 + x**. Wait, whats the extra 1 for? We did 21 x 21 but are only calculating with 20.... That extra 1 is padding. It's a good idea to add a little bit.

Set the depth range in the Initialize function of **Game.cs**

Implementation

Hint: all depth should be set based on **WORLD SPACE**, not camera space.

I'm going to leave implementation a little bit open ended. I'll make suggestions on where to call the **SetDepth** function, but you are welcome to call it anywhere.

For Tiles, set the depth for each tile in **Map.cs**, right before calling `tileMap[h][w].Render(offsetPosition)`. You can easily find the depth by doing `h * 20 + w`

For Characters, in the `Render` function of **Character.cs**. Set the depth based on the Bottom Right corner of the character. Remember, the bottom right is not measured in tiles, so you have to convert it to tiles.

Hint, Bottom right is misleading. When the players collision is corrected, technically his bottom right is one tile lower than what we want to be checking. Adjust Y by 1 pixel up, like so: `((int)Corners[CORNER_BOTTOM_RIGHT].Y - 1)` that way you get the exact foot tile!

Hint, add 0.5 to the depth to ensure that the player renders on top of the tiles. Like this `GraphicsManager.Instance.SetDepth(y_tile * 20 + x_tile + 0.5f)`, that tiny boost just makes sure that we are rendered on top of the current tile.

Run the game, you should be able to walk around. Notice how your character can walk below the roof of the house and above the obstacles:

For Items Do the same thing you did for the player. Instead of adding 0.5f to it, add an offset of 0.2f

For Projectiles, in **Game.cs** find where you render each projectile with `projectiles[i].Render(offsetPosition);` and before that set the depth to 19*19. We set it super high because these projectiles should be rendering above everything!

Lastly, in **Game.cs** before hte game over check (this line): `if (GameOver) {` make sure to set the depth to **20*20**, to

make sure that all game over text renders above the game

Isometric View

Isometric worlds are great, they can add a lot of depth and visual appeal to a simple tile based game. Just look at Diablo 2, or Starcraft 1!

There are some great resources online that really get into the nitty gritty of how isometric views work. They are worth a read.

Movement in an iso world doesn't have to be smooth. I think This demo fels awesome, and the character is snapped to a grid the whole time! Of course we are going to have smooth movement in our version.

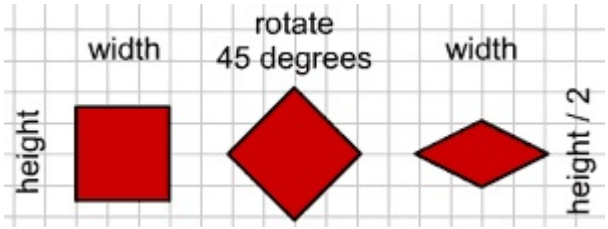
Depth buffering is very important for an isometric scene. The draw order really, really matters! A Lot!

New Project

Let's make a new project, call it **Isometric** and get this project up to par with the **DepthBuffer** section of the writeup. We're going to work from here.

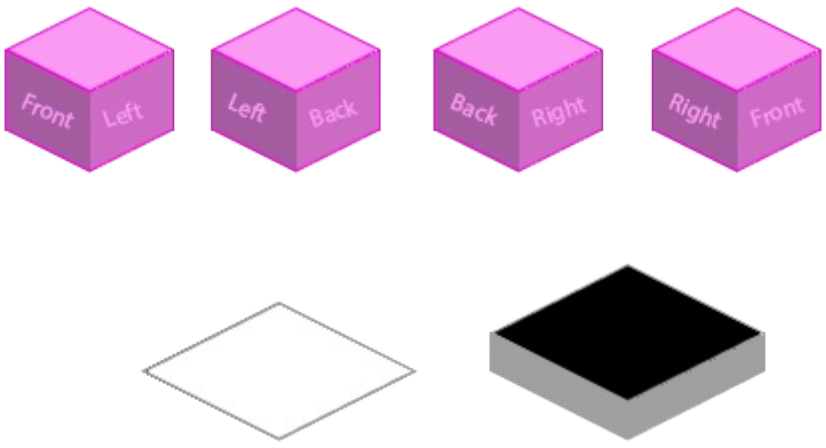
Tiles

So, how do we go from square tiles to isometric tiles? Simple, rotate by 45 degrees and scale height by 1/2 (0.5f). Like so:

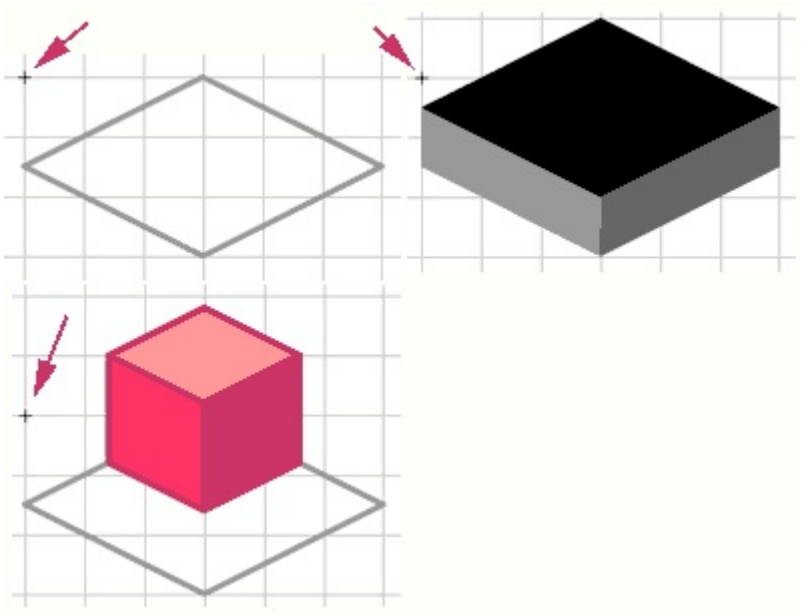


Take note that even tough the height of the tile changes, the width is untouched. We *could* do this in code, but we *should not*. Instead rely on artists to provide art assets!

Speaking of art assets, i made a new tile sheet, we are going to use this sheet for isometric tutorials. Save it to your "Assets" folder, make sure to call it **isometric.png**.



The positioning of the tiles on the source image is very, very important! The origin of each tile must line up. Remember how in the depth section we have a few tiles that where taller than 30? This is the same concept. Take a look:

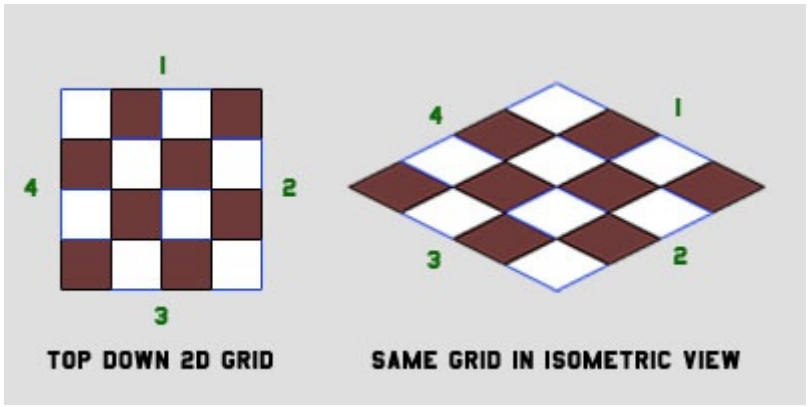


Notice how in the top, the registration point for both the flat and tall tiles are in the same spot? If we where to use a flat tile as the base of the tall tile, that's where the tip would be (On the y axis).

Same thing with player, we place him on top of the flat diamond, his registration point is where the tip of the tile would be (On the y axis).

Everything is set up relative to a base tile, in this case the base tile is just a flat outline. If you want to sprouce up the artwork you can follow some tutorials to figure out how to make iso graphics... Or just rip them from a Diablo game

So, when rotating these tiles, what direction becomes which direction? The way i think about it is that we are doing a counter clockwise rotation. Like so (Each side is numbered):



Coordinates

At the core of any isometric engine is being able to convert from cartesian coordinates to isometric coordinates. The top-down view we've been using up until now.

Atop down view grid is called cartesian coordinate system. The perspective view grid we are building is no longer using a cartesian coordinate system, it is going to be using an isometric one.

How to convert from **cartesian to isometric**?

- isoX = cartX - cartY;
- isoY = (cartX + cartY) / 2;

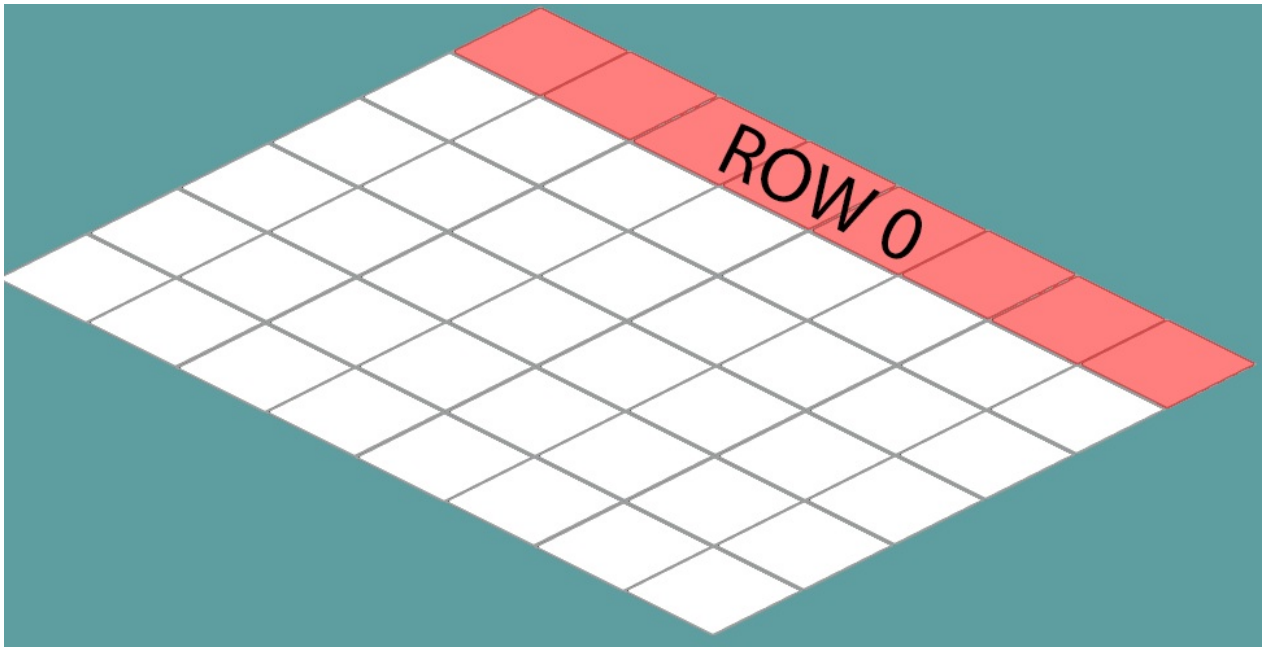
That's a nifty trick, how about going from **isometric to cartesian**?

- cartX = (2 * isoY + isoX) / 2;
- cartY = (2 * isoY - isoX) / 2;

Why do we need to convert both ways? I don't know if we *need* to at this point yet. But we are going to keep doing updates in world space. Only the view is changine to cartesian, not the logic!

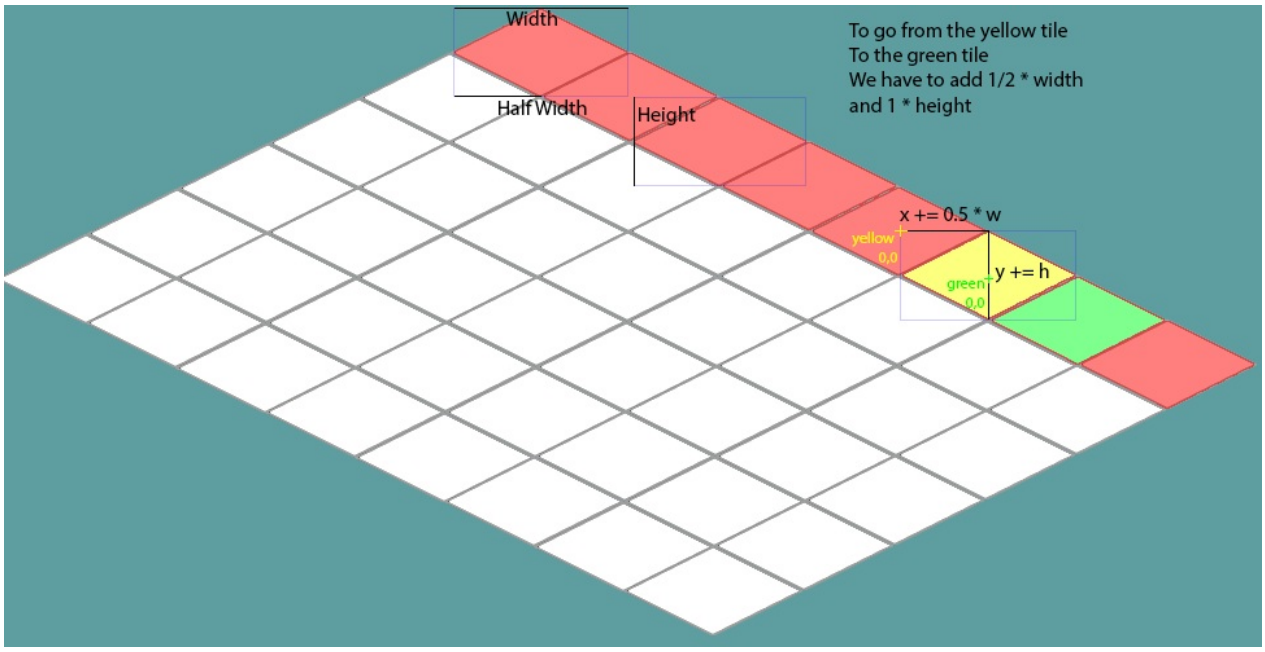
Layout & Size

It may not immediateley be obvious how iso tiles are rendered. As an example, let's see where ROW 0 is on an iso map:



The positioning of the tiles can also be misleading. This is because the registration points are not always where you expect them to be.

In order to draw the second tile of row 0, we had to go 1/2 the width of the first tile on the X and the height of the first tile on the Y. Like so:



This means that on an iso map, if a tile's source rect is `Rectangle(120, 166, 138, 70),` , then:

- Tile Height = 70
- Tile Width = 69
  - 138 / 2 = 69

On Your Own

Add these functions to **Map.cs**

```
public static PointF CartToIso(PointF cartesian) {
    // TODO
}

public static PointF IsoToCart(PointF isometric) {
    // TODO
}
```

Changine the map

Game.cs

Let's start by refactoring the map to render in an isometric fasion! Let's start by editing **Game.cs**:

- Change the value of `spriteSheets`
  - From: **"Assets/HouseTiles.png"**
  - To: **"Assets/isometric.png"**
- Change the value of `spriteSources` to
  - */*Index 0:*/* new Rectangle(120, 166, 138, 70),
  - */*Index 1:*/* new Rectangle(294, 147, 138, 90),
  - */*Index 2:*/* new Rectangle(120, 166, 138, 70)
- Change the value of `room1Layout` to
  - new int[] { 1, 1, 1, 1, 1, 1, 1, 1 },
  - new int[] { 1, 0, 0, 0, 0, 0, 0, 1 },
  - new int[] { 1, 0, 0, 0, 1, 0, 0, 1 },
  - new int[] { 1, 0, 1, 0, 0, 0, 0, 1 },
  - new int[] { 1, 0, 0, 0, 0, 0, 0, 2 },
  - new int[] { 1, 1, 1, 1, 1, 1, 1, 1 }
- Change where room1 doctor is assigned:
  - From: `room1[4][13].MakeDoor(room2, new Point(1, 1));`
  - To: `room1[4][7].MakeDoor(room2, new Point(1, 1));`
- Change where room2 is assigned:
  - From: `room2[1][0].MakeDoor(room1, new Point(12, 4));`
  - To: `room2[1][0].MakeDoor(room1, new Point(6, 4));`
- Hard code `offsetPosition` to **-200, 150** for now (We are not scrolling yet!)
- Make the client window size **990 x 550**

Tile.cs

Next we have a minor change to make in **Tile.cs**, remember how we hard coded the visual tile height to 30? Well we need to undo it. Find where in the `Render` code wecompare to and subtract 30 to give the illusion of depth, now change that to 70.

While we are in **Tile.cs**, let's update the tiles visual render position! After the offset has been applyed to render pos, update the render position by calling Map.CartToIso on the render position and assigning the return of the function back into render position.

Map.cs

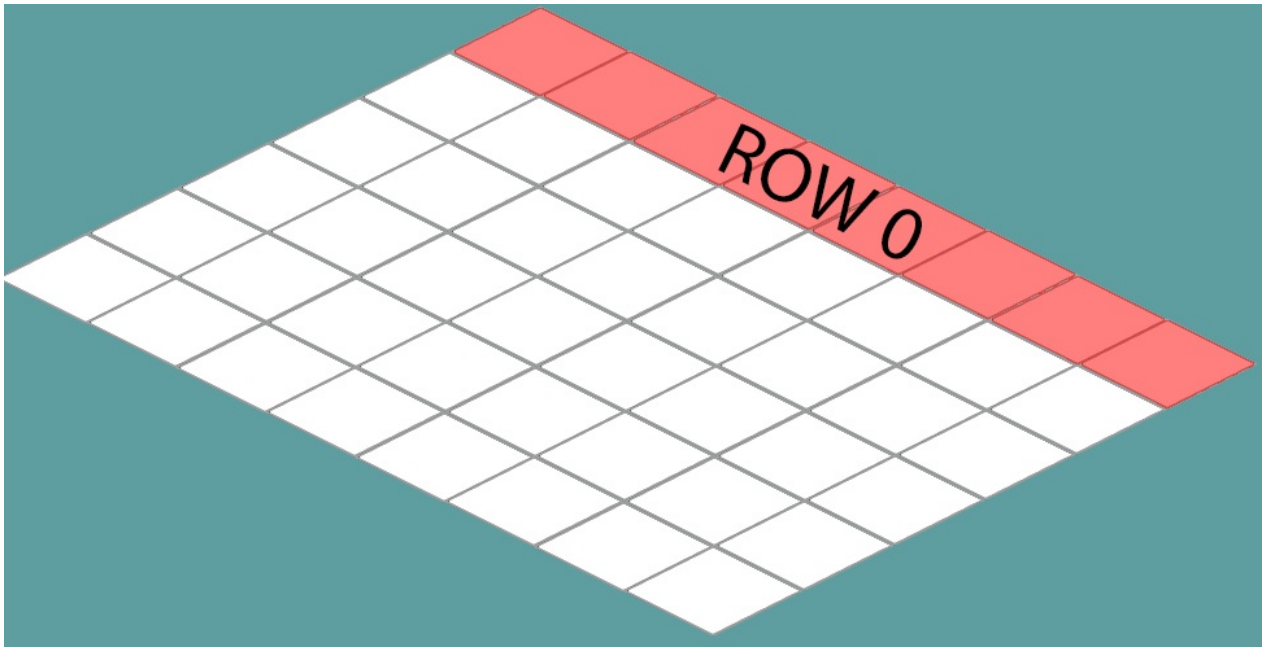
There is one more place where we hard-coded 30 as the tile size that we have to fix. Let's go into **Map.cs**. Where the world position of each tile is being set, you need to change this:

- `worldPosition.X = (int)(j * 30);`
- `worldPosition.Y = (int)(i * 30);`

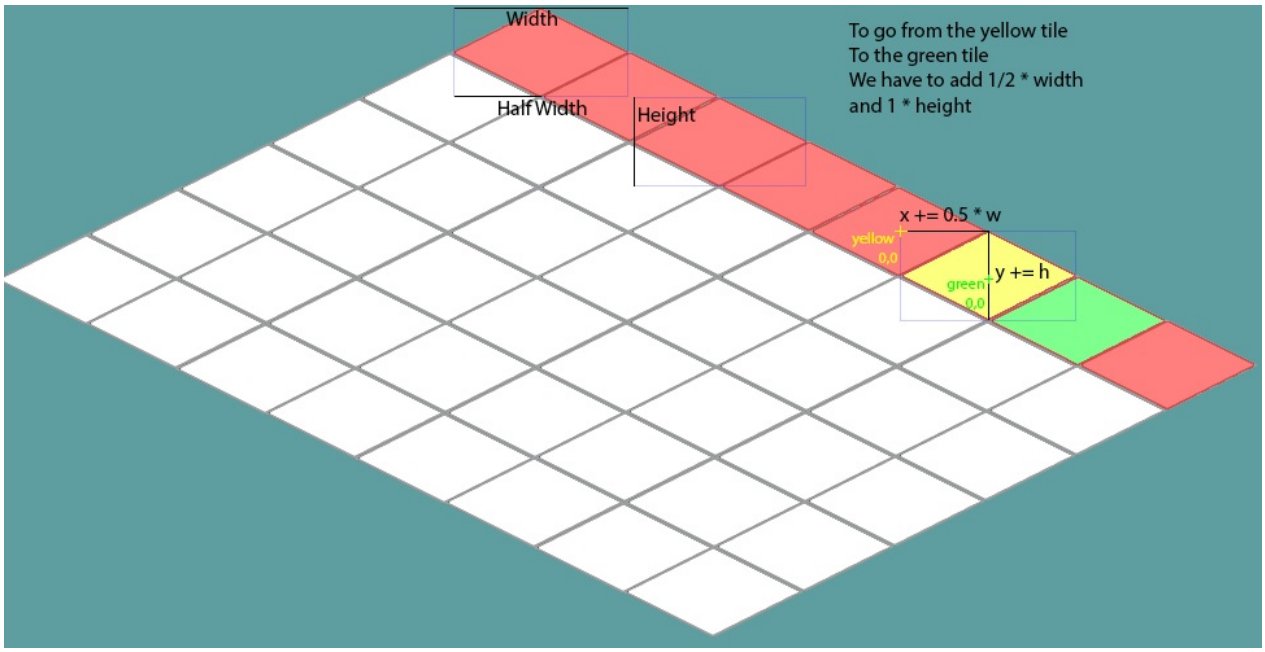
To This

- `worldPosition.X = (int)(j * (138 / 2));`
- `worldPosition.Y = (int)(i * (70));`

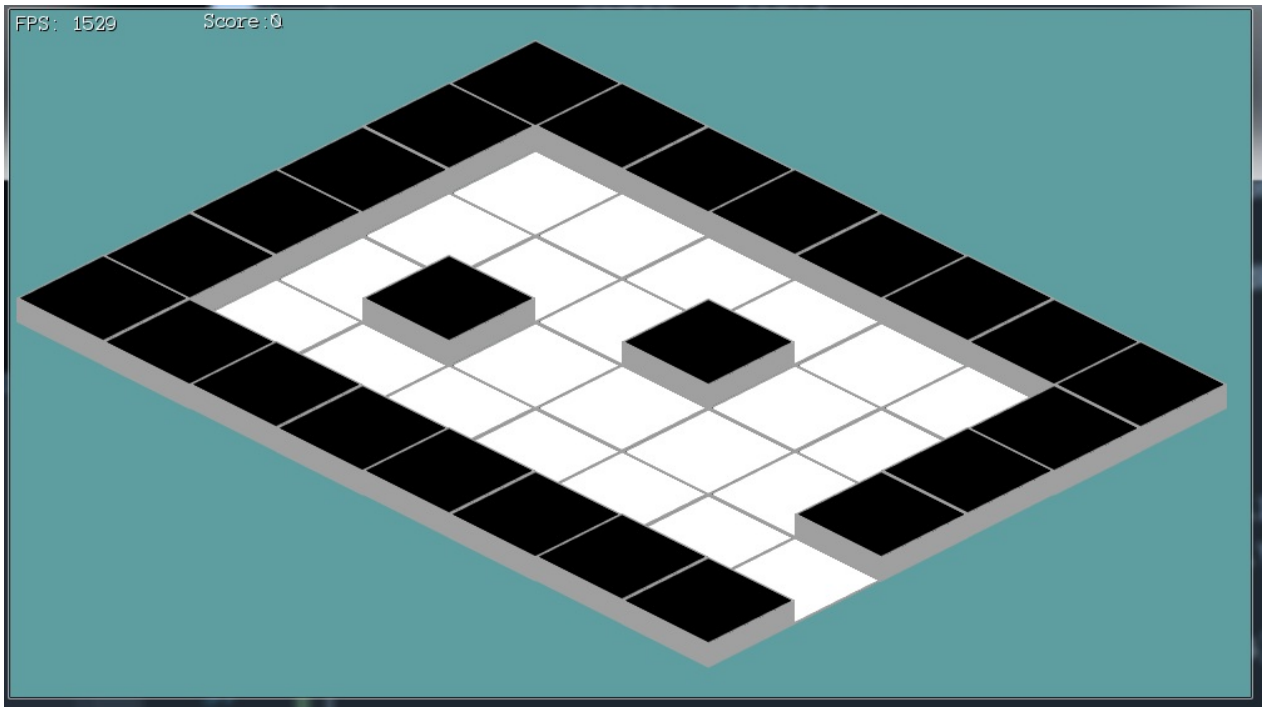
Why 138 / 2 and 70? Because the tiles are placed half the width and half width and half height apart. The width of the white base tile is 138, it's height is 70. Take a look at the image below, this is row 0:



Now how far away on the X and Y are each new tiles in row 0?



Run the game, you should see your first isometric map be rendered:


FPS: 1529
Score:0

Characters

Moving the characters into an isometric view might be a bit of a challenge. We're going to have to refactor far more code here than in the previous sections of the tutorial.

While doing all of this, keep in mind we are not doing collisions or any logic in an isometric space. Isometric is only a view, we still handle collisions and items in world space.

Recall there are three important spaces: world space, camera space and screen space. Applying an isometric projection only changes the camera space. "Applying an isometric projection" might sound fancy, but reality, that's what the `CartToIso` in **Map.cs** does.

Rendering World Space

Because we're doing all our logic in world space debugging will be a lot easyer if we can also see world space! If a collision looks wrong, it happens in an isometric view, but not in a top down view of world space then we know that the issue is render code and not collision code.

First, let's make a new variable in **Game.cs**. A public, static bool. Call it `ViewWorldSpace` and set it to false by default.

Update the `Update` function of **Game.cs**. Pick any key (I suggest 'U'), when that key is pressed toggle the `ViewWorldSpace` boolean.

In the `Render` function find where offsetPosition.X and Y are hard coded. Wrap that bit of code in an if statment, we only want to hard code the X and Y if `ViewWorldSpace` is **false**

Next, let's undo the isometric projection. In **Map.cs** find the `CartToIso` and `IsoToCart` functions. If `Game.ViewWorldSpace` is true, return the arguments without any modification.

Last, we need to make a slight modification to the rendering code of **Tile.cs**. If `Game.ViewWorldSpace` is true, Draw a rectangle. The Location of the rectangle is going to be `WorldPosition`, while it's size is going to be 69 x 70. (69 is 1/2 the width of each tile, 70 is the height of each tile). Thse are the numbers we set up in the constructor of **Map.cs** for the size and position of tiles. If `Game.ViewWorldSpace` is false, render everything as usual.

Further more, when `Game.ViewWorldSpace` is true and we render a rectangle, if the tile is walkable render the rectangle in `Color.LightSteelBlue`, otherwise render it in `Color.LightSlateGray`.

Run the game and toggle the display. Your non-isometric world view should look like this:

No more 30

Now that we can view world space just fine, we notice that the size of the world space tiles is MUCH larger than the size of our player and enemy tiles. Let's go ahead and fix this next.

The sizing issue happens because the map is rendering with our worlds current tile size, while the character an enemy have 30 hard coded for the tile size. That's no-bueno.

We have a global define in **Game.cs**, find the `tileSize` global variable and delete it. In it's place, add these two new defines:

```
public static readonly int TILE_W = 69; // 138 / 2
public static readonly int TILE_H = 70; // 70
```

We just caused a lot of compiler errors be removing `tileSize`, go ahead and fix these. Instead of `tileSize`, use `TILE_W` and `TILE_H` wherever appropriate. Appy H to Y values and W to X values.

Now it's time to replace EVERY OTHER INSTANCE of 30. Search the entire project *(Control + shift + F)* for **30**, replace every instance with TILE_W or TILE_H, depending on which one is appropriate.

Update the constructor of **Map.cs** to use the constants from `Game`, instead of hard coding 138 / 2 and 70.

In **Tile.cs** we hard coded the debug render size to 69x70. Switch these to be the constants in `Game.cs`, also, there is a height text of 70 hard coded in there, change that 70 to `Game.TILE_H`.

Run the game, if you switch into debug view, even tough link is the wrong size he can walk around the map. He just looks small. Confirm that collision and shooting work before moving on to the next section.

Display the Character

Let's take a little bit of time to refactor **Character.cs**. The first thing i'm going to do is remove the code that renders the debug yellow square at the characters registration point.

Next, we're going to updat the character sprites. In **PlayerCharacter.cs** and in **EnemyCharacter.cs**, change the sprite sources to:

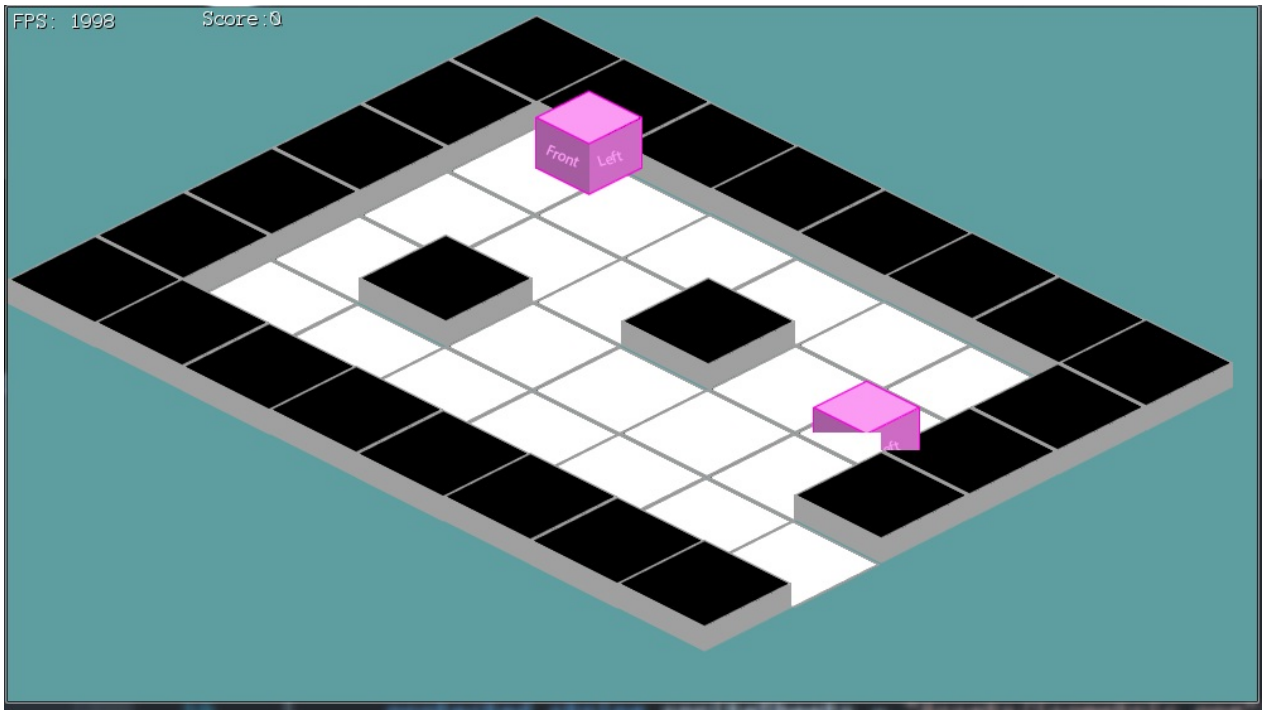
- AddSprite("Down", new Rectangle(52, 19, 85, 84));
- AddSprite("Up", new Rectangle(266, 19, 85, 84));
- AddSprite("Left", new Rectangle(155, 19, 85, 84));
- AddSprite("Right", new Rectangle(375, 19, 85, 84));

Back in **Character.cs**, let's go ahead and project these characters into an isometric space. In the `Render` method after the offset is applyed to `renderPosition`, but before the y correction takes place go ahead and call `Map.CartToIso`.

`MapCartToIso` will transform `renderPosition` from a cartesian space into an isometric space. Do this the same way you have already done it for **Tile.cs**

Lastly, back in **Game.cs**, go ahead and change `heroSheet` and `npcSheet` to point to the isometric sprite sheet.

Run the game, the isometric view should look pretty close to correct. Moving around is going to be broken, but visually it's close to good. We're going to just ignore z-fighting issues for now. This is what your game should look like:



Moving the character

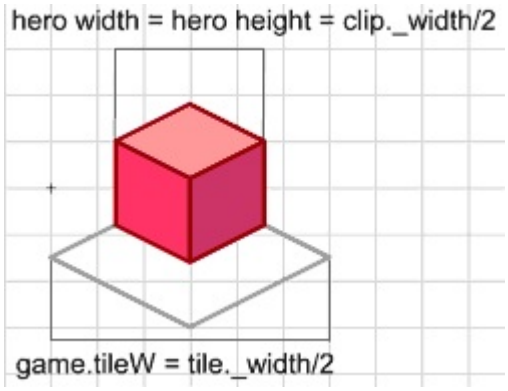
If the moving code is broken, then it's broken in a few places. Switch to the overhead view and see if movement is broken... Well it's not really broken, but it's also not correct. We should only see squares in this view, not sprites.

In **character.cs**, update the render method so if `Game.ViewWorldSpace` is true, we draw a square (use the Rect getter for it's rectangle), make it `Color.SteelBlue`. If `Game.ViewWorldSpace` is false, draw the texture as usual.

Looking at our new debug render, the errors are quiet obvious! The player is bigger (wider) than the floor tiles. That's a pretty big no-no. And how exactly does the size in this screen translate to the size of the iso sprite? After all the iso

sprite only takes up a part of each tile.

Just like the size of each tile is no longer the same as the size of it's sprite, the isometric size of the player tile is not the same as the size of its sprite. We have to apply the same transform to player as we do to tiles, so we are going to cut his width in half. How about the height? We will just make it the same as his width... For now.



`Character` alredy has a `height` variable. Delete it. Change the constructor to just do nothing with the height argument, ignore it.

Inside the `Rect` getter, make two new local intgers: `width` and `height`. `width` is going to equal the current sprites width, divided by two. `height` is going to equal `width`. Return a rectangle at `Positon`, with a size of` width `, height```. The if statement in here is no longer needed.

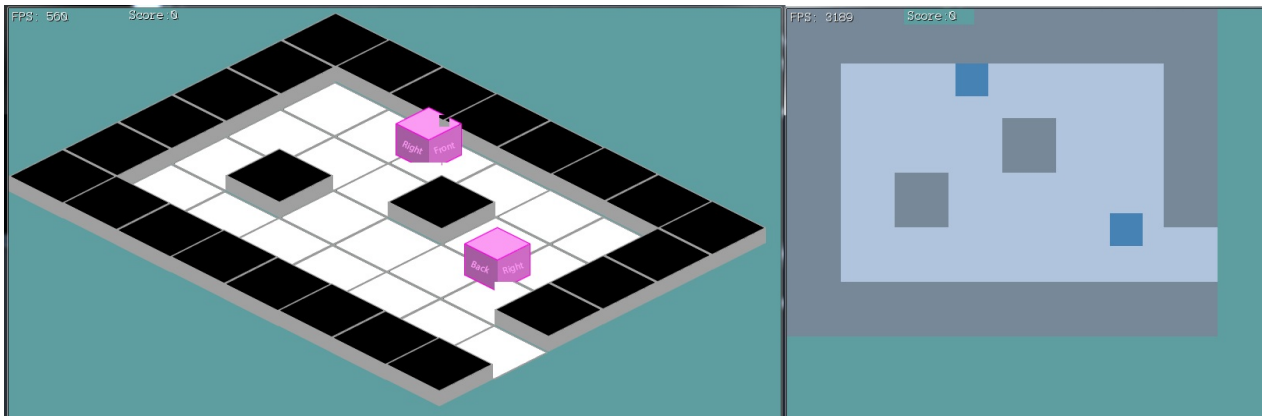
The `Center` getter already uses the `Rect` getter, there is nothing to change here.

In the `Corners` getter, change w to equal `Rect.Width` and h to equal `Rect.Height`

Lastly, there is a reference to the old `height` variable inside of the `Render` function. Change the line to subtract `Rect.Height` instead of the old member. Also, because the if statment this is in no longer makes sense, change it to:

```
if (SpriteSources[currentSprite][currentFrame].Height > Rect.Height) {
```

Run the game, walking around and collision should be working pretty well. Especially in overhead view:



Rendering artifacts

We have a few rendering artifacts going on. The most obvious of which is the bits of tile that render above the character. This happens becuase what gets written into the depth buffer is a square. Even if a diamond is rendered

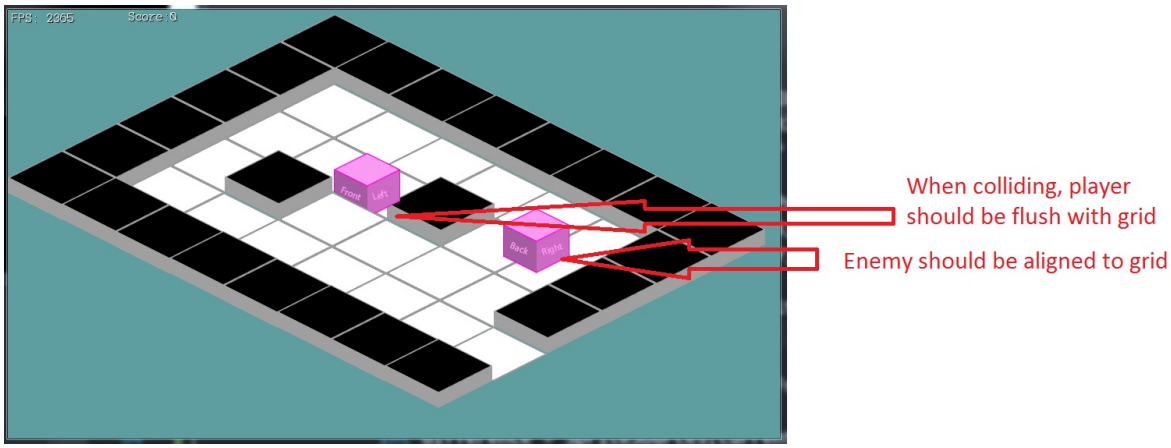
to the clor buffer, a square is rendered to the depth buffer.

This square ends up putting pixels where they are not supposed to go! Open up **GraphicsManager.cs**, go to the `Initialize` function, and there are two lines commetned out:

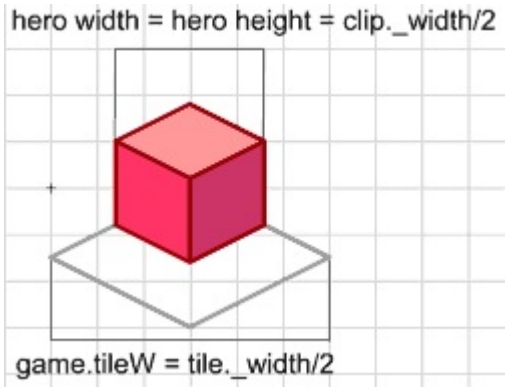
- `GL.AlphaFunc(AlphaFunction.Greater, 0.1f);`
- `GL.Enable(EnableCap.AlphaTest);`

Go ahead and comment these two lines back in, they will make it so alpha (transparent) pixels don't get written into the depth buffer.

The next issue is that the character does not line up to the grid!



This is not an easy one to fix. The issue is our player sprite uses it's middle left as a registration point right now. But in an isometric view, everything needs to be modeled relative to a tile. Look at this picture again:



The + on the left marks the registration point, it's the top left for the floor tile. BUT, it also marks the registration point for the player! Notice how the players registration point doesn't actually touch the sprite!

So how do we figure out how much space we need to offset the player by? Looking at the picture, it's `(TileCenter - TileW * 0.5) - (PlayerCenter - PlayerW * 0.5)` Doing the math with our known numbers, that comes out to about **25**.

You could calculate this at runtime, to support different size enemies, but we're just going to keep it simple. So, where to apply this offset? In **Character.cs**, the `Render` function. Find where the `renderPosition` is projected into isometric space. Directly after that add 25 to it's X component!

There is one last bug. If you walk into an obstacle from the X direction you will first be behind it, but then quickly pop in front of it! We have to offset the X position of the player by 1 pixel just like we did the Y position of the player. Look at the top of the `Render` function in **Character.cs**, and offset X by 1 the same way Y is offset by 1.

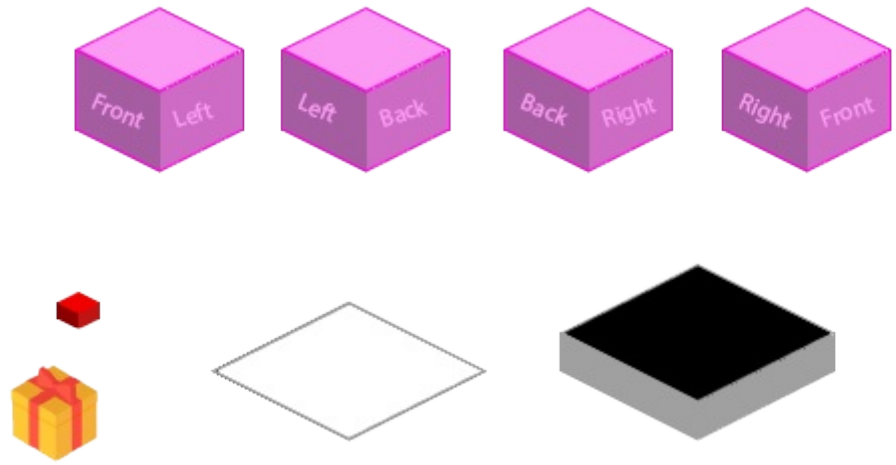
Run the game, everything should now work in an isometric environment. Your character can walk around, collide with walls and even enemies. As an adde bonus, the walls even have height to them!

Items and projectiles

Now that the player and enemy are both in isometric space, and 30 is no longer hard-coded anywhere in our code changing the items and the projectiles to be isometric as well is going to be super easy. You might even be able to do it without this guide.

First, we update the sprites to use isometric version. Next, we want to make sure that both projectiles and items work in our debug overhead view. Finally, we apply the isometric projection with the helper function that lives in **Map.cs**. And that's pretty much it.

Here is the updates isometric.png sheet that has items and bullets on it:



New projet?

You can either do this in a brand new project, or keep working int he lat isometric one. I'll leave it up to you.

Updating tile sprites

The source rectangle for an item is defined in the constructor of the `Item` object. We make new items (call the constructor) in **Map.cs** in the `AddItem` function, which just forwards the `source` argument. The `AddItem` function is actually called in **Game.cs**. Find the following lines in **Game.cs**

```
room1.AddItem(spriteSheets, new Rectangle(350, 255, 16, 16), 10, new Point(4 * TILE_W + 7, 2 * TILE_H + 7));
room1.AddItem(spriteSheets, new Rectangle(381, 256, 13, 15), 20, new Point(5 * TILE_W + 7, 4 * TILE_H + 7));
room2.AddItem(spriteSheets, new Rectangle(412, 256, 16, 15), 30, new Point(4 * TILE_W + 7, 2 * TILE_H + 7));
```

Let's update the source rectangle, all items are going to use the same image. The new source rectangle is: **Rectangle(20, 198, 44, 49)**. While editing these, go ahead and remove all the **+ 7** parts. 7 is the number we hard coded to put each item in the middle of a tile, now that tile sizes are different, 7 holds no meaning.

Also, this line is not valid:

```
room1.AddItem(spriteSheets, new Rectangle(20, 198, 44, 49), 10, new Point(4 * TILE_W, 2 * TILE_H));
```

This is because i accidentally changed where one of the obstacles are. instead of the obstacle being at tile 5, 2 it's at 4, 2. Meaning this items is directly inside an obstacle! Update it's position to be at 3, 2.

Go int **Tile.cs** and let's add an if statement to the `Render` function to render the debug view of the item. Add an if statement, if `Game.ViewWorldSpace` is true we render a debug square, if it's false we render the texture like normal.

The debug rectangle should be at the position specified by the `Position` member variable. It should be of color `Color.DarkSeaGreen`. As for size, both the width and the height are going to be **Source.Width / 2**. The size of the item works the same as the size of the player, so width is sprite width / 2 and height is width.

While we are in the render function, find where we offset the y pixel position of the tile: `int yTile = (Position.Y - 1) / Game.TILE_H;`, get rid of the -1 offset. Because isometric tiles use a different registration point than cartesian tiles, we no longer need to apply this offset.

Run the game, you should now be seeing green item squares on the debug map. So long as your looking at the top down debug map you can even move around and collect the items!

Adding a shooting sprite

Shooting already works in the debug view. Go ahead, try it. The thing is, bullets never had a sprite to begin with, so we must provide them with one!

Let's start by adding two new variables to **Bullet.cs**. First, an integer, name it `spriteSheetHandle`, next a Rectangle, name it `sourceRect`. The initial value of sourceRect will be **43, 161, 22, 20**

Update the `Rect` getter to return a rectangle at the location of the `Position` variable, with a width and height of **source width / 2**. The same logic that dictates the player width / height also dictates the bullet width / height.

Change the constructor to take a third argument, the new argument is an integer and is a sprite sheetreference. In the constructor, set `spriteSheetHandle` equal to this new value.

In the `Render` function, change ``Rectangle renderRect `to` PoinF renderPoint `. If` Game.ViewWorldSpace `is true, draw a red rectangle, use the` Rect``` getter for the actual rectangle object to draw.

If `Game.ViewWorldSpace` is flse, draw the new bullet sprite. The spriteSheet is `spriteSheetHandle`, the render poisition is `renderPoint`, scale is 1 and the source rectangle is `sourceRect`

Trying to compile the game now you will get some compiler errors. This is because we changed the constructor, but not where it gets called.

In **Game.cs** make a new member integer, let's call it `spriteSheetInstnace`. In the Initialize function, load the isometric sprite sheet into this. In the Shutdown function, don't forget to unload the image. Finally, where the new bullet is added, pass `spriteSheetInstnace` in as the last argument.

Run the game, everything should more or less work as before. The bullet being shot in overhead view might look a little off center, but that's ok.

Isometric projection

Now that both bullets and items work in a 2D verhead view (that is, they work in world space) lets make them work in an isometric view by applying the proper projections and offsets!

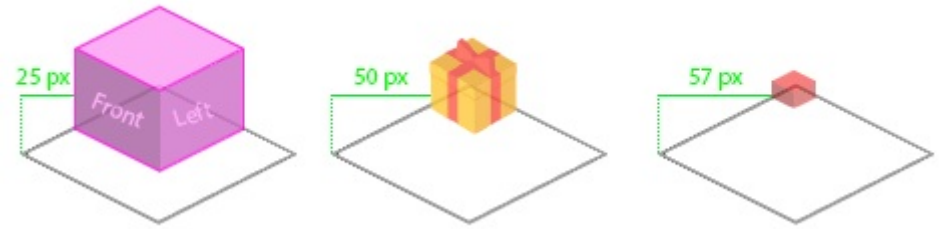
In **Bullet.cs** take the `renderPoint` variable and project it into isometric space using `Map.CartToIso`. Of course, only do

this after all offsets have been applyed. If you look where we first set the position, we subtract 5 from the X and Y. Remove the -5. It worked when we where in a top down view, but we don't need it anymore (right now i should say)

In **Item.cs** move the `renderPosition` into isometric space.

Run the game, shoot some bullets, and check out where the items are. You will notice that both bullet shooting and item positions are way off from where we expect them to be. But why?

Remember how the player's registration point needed to be the same as the base tiles registration point? Well that holds true for all objects! Here is a picture:



Just like we offset player by 25, we're going to offset the present by 50 and the bullet by 57. The above image shows both bullet and present as having some height (y offset), don't let that fool you we're only going to apply an X offset.

In **Bullet.cs**, after you have converted `renderPoint` into isometric space, add 57 to it's X value

In **Item.cs**, after you have converted `renderPosition` into isometric space, add 50 to it's X value.

That's it! **Run the game** and you should have an awesome, interactive isometric world ready to go!

Mouse To Move

Using the mouse to move will involve some rudimentary pathfinding / AI. We're going to implement the simplest path-finding possible. Usually, when doing a mouse to move you would use the A* algorithm to find the shortest path, however i have that lesson planned for later.

New Project

Let's make a new project, call it **MouseToMove** and get this project up to par with the **OpenTheDoor** section of the writeup. We're going to work from here.

We don't care about items, enemies or bullets. We just want to be able to walk around with a mouse. And i'm hoping this means typing a little less boiler plate code.

Tile Size

We are starting off from a state before we did the tile size refactor in the isometric section (obviously), but i don't like having 30 all over the place. I made a new `public static readonly` variable in **Game.cs** called `TILE_SIZE` and refactored the rest of the project to use this instead of the hard coded 30

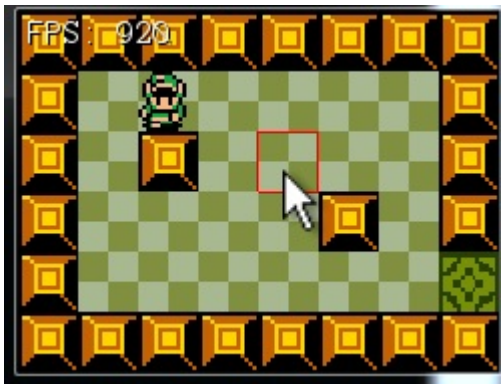
You can keep hard coding 30, or do this refactor too, it's up to you. For the remainder of this article i'm going to be refering to the size of the tile by this new constant in **Game.cs**

Mouse Indicator

Clicking on tiles is usually obvious enough, but having a mouse indicator is always a nice touch. We're going to figure out which tile the mouse is over and highlight it with a red outline.

In **Game.cs**, make a new `Point` variable, call it `cursorTile`. In Update, we're going to set this. Set it to the tile that the mouse is on. Because our tiles are based on a grid of 30, if a mouse is at 45,67 it's tile would be 2,3.

In the render function, go ahead and draw a red rectangle around the selected tile. Remember to bring the tile from "tile space" back into "world space". We don't have a method to draw a square outline, you will have to do it with 4 lines. Here is what this should look like:



Note: Right now you probably just divided the mouse position X and Y by tile size, that should work very well for this demo. Be aware however, if you add scrolling, you will have to account for the scrolling offset in the mouse position!

Target location

Inside **PlayerCharacter.cs** make a new variable:

```
protected Point targetTile = new Point(2, 3);
```

This variable represents the tile we want to walk to (in tile space). Publicly, we don't need any way to see the value of the variable, but we do need a way to set it. Let's add a setter function:

```
public void SetTargetTile(Point target) {
    targetTile = new Point(target.X, target.Y);
}
```

Next, in the `Update` method, after getting a local input manager instance, find the tile that the player is currently in:

```
Point currentTile = new Point((int)(Position.X) / Game.TILE_SIZE, (int)Position.Y / Game.TILE_SIZE);
```

Broad phase movement

At this point we know the current tile of the player and the target tile of the player. We also have code to move the player on screen with the keyboard. What we need to do is to change the movement conditions.

Instead of looking at the keyboard, the movement conditions should look at the current and target tiles of the player.For example, if player tile is 5, but target tile is 2 the player should just move left!

Change this line:

```
if (i.KeyDown(OpenTK.Input.Key.A) || i.KeyDown(OpenTK.Input.Key.Left)) {
```

to this:

```
if (targetTile.X < currentTile.X) {
```

And this line

```
else if (i.KeyDown(OpenTK.Input.Key.D) || i.KeyDown(OpenTK.Input.Key.Right)) {
```

to this:

```
else if (targetTile.X > currentTile.X) {
```

Chang the up and down if statements on your own. We call this broad phase movement, because it gets the player to the correct tile, not the correct pixel. We will add narro phase movement soon.

Hook it up

In **Game.cs**, set the players target tile if a tile is clicked. We only want to set this target if the tile being clicked is walkable:

```
if (InputManager.Instance.MousePressed(OpenTK.Input.MouseButton.Left)) {
    if (currentMap[cursorTile.Y][cursorTile.X].Walkable) {
        hero.SetTargetTile(cursorTile);
    }
}
```

Run the game, clicking around the map should cause the hero to walk to the selected tile! There are two problems. First, when you are behind a blocking tile and you click straight ahead the player will just forever walk into the obstacle. This is just dumb ai. We are not going to fix that in this lesson.

The second, and perhaps more glaring issue is that the player is ok walking right or down. However walking left or up the player stops just one tile short of the target. This is because our registration point is in the top left of the character and the top left pixel is in the target tile already.

Narrow phase movement

The only way to solve our hero stopping one tile short is to add more fine grained collision controls. The `if` / `else if` for x movement, we're going to add another `else if` to that.

This new else if checks how far the player is from the target x / y position (pixel, not tile). If he is close enough, he will be snapped to the pixel, if not he will just continue moving towards it.

```
else if (Position.X != currentTile.X) {
    // More than two pixels away, walk
    if (Math.Abs(Position.X - currentTile.X * Game.TILE_SIZE) > 2) {
        SetSprite("Left");
        Animate(deltaTime);
        Position.X -= speed * deltaTime;
    }
    // Two pixels away, snap!
    else {
        Position.X = currentTile.X * Game.TILE_SIZE;
    }
}
```

Dont forget to do the same thing for the y axis! **Run the game**, clicking around, link should always walk to the right tile. Evenif he takes a dumb route.

Doors

Now clicking aroudn moves us correctly, but what happens if you walk into a door? You try to walk to the door tile on the next map! This needs to get fixed.

Find the code that makes the hero walk trough a door and shows the new room. In that code, set the heros target tile to be right outside the door (the specified target position). That should cuase the player to just switch rooms.

I purposley left the instructions for that last part very vague. I want you to think about this, look at your code and try to solve the problem yourself. If you need any help or guidance i'm aways here.

More Tiles

In all of our examples we have loaded tiles by hand. We did breafly look at how to use tools to generate some understandable data, but it only goes so far. Let's leave off with an easy to parse map format. In this chapter we're going to load maps.

Define a format

I want you to come up with a map format! Each map will contain ONLY 1 room. Door tiles can go to other maps, how you represent those maps is up to you (By number, by string, or by whatever).

Here is what a format *might* look like:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 3 3 3 0 0 0 0 0 0 0 0 0 1
1 0 3 3 3 0 3 3 3 3 3 3 3 0 1
1 0 0 0 0 0 3 3 0 0 0 0 3 0 1
1 0 3 3 3 0 3 3 3 0 3 3 3 0 1
1 0 3 3 3 0 0 0 0 0 0 0 0 0 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

T Assets/TileSheet.png
R 0 23 87 98
R 1 23 87 98
R 2 23 87 98
R 3 23 87 98
W 0 2
D 2 Assets/NextRoom.txt
S 2 1
```

I assume NO HEIGHT in my map format. In this hypothetical format, blank lines are skipped. Any line that begins with a number records a row. Any line that starts with a letter starts some meta-data. Here is my meta data format:

- **T** [Path]
  - Specifies where the texture for this sheet is loaded
- **R** [index] [x] [y] [w] [h]
  - Specifies the source rectangle
  - index - which tile are we specifying the source of (0, 1, 2...)
  - x, y, w, h - rectangle size
- **W** [index list]
  - Specifies which tiles are walkable
  - Takes a list of integers, any integer in the list is walkable
- **D** [Index] [Path]
  - A door tile!
  - The first argument is the index. In this map, tile 2 is a door
  - The second argument is another map file! Load this map file when the door is entered
- **S** [x] [y]
  - Specifies the starting tile for this map.
  - When the map first loads, place the hero here.

There is no requirement for this. You can include as many or as few items in your map as you want (You don't *have* to have doors unless you want to). Feel free to use my proposed format as a jumping off point.

Loading the map

Right now we are hard coding the map layout into **Game.cs**, this needs to change. Change the constructor in **Map.cs** to take a single string for it's argument. This string should be the path of the map file to load. All map info needs to be in this file!

You should craft a map file by hand, then modify the constructor of `Map` and try loading that file.