XNA Development: Tutorial 5

By Matthew Christian (Matt@InsideGamer.org)

Code and Other Tutorials Found at http://www.insidegamer.org/xnatutorials.aspx

Let's make a HUD! A HUD (heads-up display) is the typical screen items used in most games to clue the player in on things like how much life the character has left, character position/direction with a small map, and any weapon or point numbers. Another name for the 2D objects placed on the screen is sprites. All these items will be basic 2D and have different effects used with them to provide a functional HUD.

Directory Structuring

Following the first three tutorials you should be able to use the same project, otherwise make sure to start a new project and create it with the Windows Game (2.0) template. Then, follow XNA Development: Tutorial 3 on the GameModel class. The GameModel and GameCamera class are flexible enough that with a little push or pull they should work under both the Windows and Xbox 360 architectures. As normal, tutorial files will be named Game1.cs, GameModel.cs, and Program.cs. You can change the names of the files but for ease of reference this tutorial uses default file names.

We're going to introduce a new type of game content, .PNG files. These are 2D files that allow transparency layers so anything behind the image will stilll show up. Currently, it would get really messy quick if we just started adding images and models into the Content directory of the solution. To solve this, we can add folders in the Solution Explorer and move all the content into the respective folder.

Right click on the Content Folder and select Add > New Folder. Name the new folder, 'Models'. Add another folder within the content folder named 'Sprites'. The next part is tricky if you have any models in your scene. Drag the model files into the model folder (no this is not the tricky part). You've successfully moved the model files into their proper sub folder **but** you haven't moved the textures related to that model and compiling will most likely come back with an error saying "Missing Asset", then lists the placement of a texture file. Open an explorer window and browse to your solution directory. Copy every texture file related to the model(s) into the Models subfolder. Make sure you leave GameThumbnail.png where it is.

Modifying GameModel Initialization

Now that the model data is in a different subdirectory, GameModel needs to be told where to find the models. Look at the following code found in the Initialize() method of the Game1 class:

```
myHouse = new GameModel(content, "Models\\houseModel");
```

The only portion being changed is the string passed in. Previously, only the model name was sent and XNA would look in the solution's Content directory, but now that the model isn't there that wouldn't be correct. XNA is now being told to look for the model 'houseModel' in the 'Content\Models\' subdirectory. We use '\\' because a single backslash is used as an escape character in programming and two are used to represent a single backslash.

GameSprite Class Design

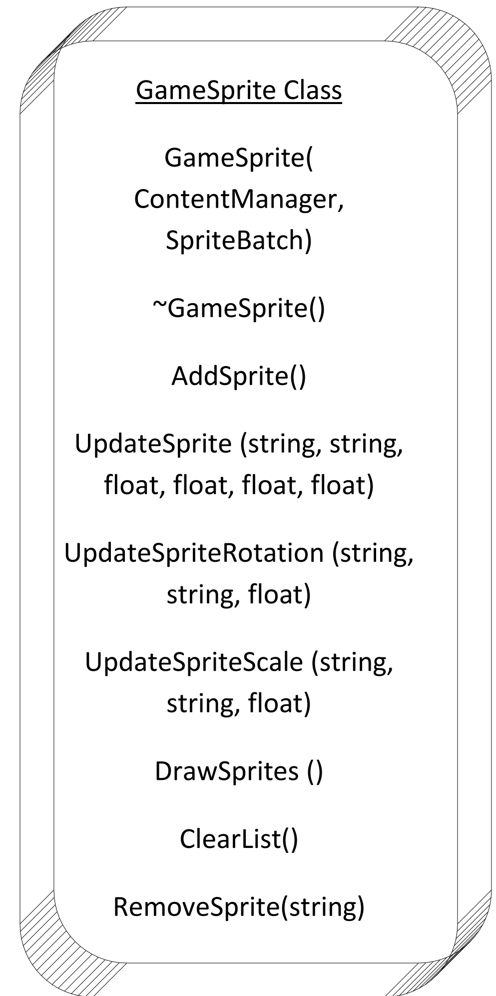

Diagram 1 –GameSprite Class

The GameSprite class is designed a bit differently due to some of the technical aspects used when drawing sprites. Sprites use and object called a SpriteBatch to collect all the sprites that need to be rendered at once (sort of like a grouping) and runs that batch to the graphics card instead of one at a time (which saves calls to an from the CPU to the GPU). This means the GameSprite class will only need one SpriteBatch but, a new object of the class for each sprite would have a new SpriteBatch. To fix this, a small version of a resource manager is implemented by creating a list and adding new sprites to it.

As normal the constructor of the class will initialize some basic components. GameSprite also contains an AddSprite() method, a few update methods, a DrawSprites() method, and some methods to clean out sprites. Don't worry, we'll go over all this next.

Programming the GameSprite Class

To start, create a new class file named GameSprite.cs and add the following in place of the `using` statements at the top:

```
#region Using Statements
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;
#endregion
```

Within the GameSprite class a list of sprites is needed but each sprite has multiple values associated with it like position, texture, rotation, scale, etc… What we will do is create a list of items using structs. A struct is like a user-defined type that has multiple values associated with it. For example, remember how Vector3 variables in the past tutorial could access each item such as myVar.X and access the X element? X is simply a property value associated with the Vector3 variable. Believe me, it's much easy to see the actual code to understand it.

```
//New sprite Struct
public struct sprite
{
    public string resourceName;
    public Vector2 screenPos;
    public Texture2D spriteTex;
    public float rotation;
    public float scale;
};

//Game Batch
private SpriteBatch gameBatch;

//List of game sprites
```

```
public List<sprite>gameSprites = new List<sprite>();

//Game content manager
private ContentManager gameContent;
```

Place this code within your GameSprite class as these will be class global. First a struct is created which has the values resourceName, screenPos, spriteTex, rotation, and scale. These values are used for uniquely identifying the sprite, setting the screen position, defining which texture to use, specifying the amount of rotation, and specifying the amount to scale the sprite, respectively. The struct has been named 'sprite' (notice how the word sprite comes at the end of the definition and is light blue). This is only the definition, when we actually create an object of this type you'll understand it more.

```
private SpriteBatch gameBatch;
```

Then the game's SpriteBatch object is created, the functionality of the SpriteBatch object was described above.

```
public List<sprite>gameSprites = new List<sprite>();
```

Next comes the List being used to store all game sprites. It might seem weird since List requires < and > but all that tells the computer is what kind of items will be stored in the list. The list will store objects of type 'sprite' (the type of the struct) so the list has <sprite>. Finally, each time we add a new sprite we need to load it from the game's content manager but we don't want to send the content manager each time so we'll send it one (in the constructor in this case) and store it for use later.

```
private ContentManager gameContent;
```

GameSprite()

The initialization routine within the class's constructor is relatively straight-forward.

```
public GameSprite(ContentManager content, SpriteBatch batch)
{
      //Set game content manager and create new game batch
      gameContent = content;
      gameBatch = batch;
}
```

The constructor takes two parameters, a ContentManager and a SpriteBatch. First, the ContentManager is stored for later use. In XNA 2.0, the ContentManager is built directly into the project and can be accessed in Game1.cs with the 'this' operator. Then, the SpriteBatch object is set to the game's SpriteBatch. XNA 2.0 automatically builds a SpriteBatch object when creating a new project so all that's need is a reference to it.

~GameSprite()

This is the first time we've actually had something in the destructor and this really isn't necessary but is nice to have just as a 'clean-up'.

```
~GameSprite()
{
      gameSprites.Clear();
}
```

Remember, gameSprites is the list of all game sprites. The call to Clear() removes everything within the list, simple as that. Since the object of the class is being destroyed, the list isn't needed anymore so it can be cleared out.

AddSprite()

AddSprite() is the most vital method of the GameSprite class other than DrawSprites(). It's arguable that this is even more important since without adding sprites to the list, there won't be any sprites to draw! Here's the full code:

```
public void AddSprite(string handle, string spriteName, float x, float y, float
      rotation, float scale)
{
      //Create a new sprite
      sprite newSprite = new sprite();
      newSprite.resourceName = handle;
      newSprite.spriteTex = gameContent.Load<Texture2D>("Sprites\\" + spriteName);
      newSprite.screenPos.X = x;
      newSprite.screenPos.Y = y;
      newSprite.rotation = rotation;
      newSprite.scale = scale;

      //Add new sprite to list
      gameSprites.Add(newSprite);
}
```

AddSprite() opens taking in 6 arguments which is a lot but are necessary. Remember, the struct is defined as the following:

```
public struct sprite
{
    public string resourceName;
    public Vector2 screenPos;
    public Texture2D spriteTex;
    public float rotation;
    public float scale;
};
```

Looking at the struct it should make sense the AddSprite() method takes 6 arguments (the struct only has 5 properties but we are passing in a float for the X and Y positions which are then set into a Vector2 in the struct). Before filling out the struct, we need to create a new object of that struct (of type sprite).

```
sprite newSprite = new sprite();
```

This is how a new variable with the properties of the struct is initialized. It's the same basic principle used to create GameModel and GameCamera objects in the main file in past tutorials. As mentioned above, a struct is basically a small class only containing properties. Now, each property of the newly created struct can be accessed with the dot (.).

```
newSprite.resourceName = handle;
newSprite.spriteTex = gameContent.Load<Texture2D>("Sprites\\" + spriteName);
newSprite.screenPos.X = x;
newSprite.screenPos.Y = y;
newSprite.rotation = rotation;
newSprite.scale = scale;
```

For each item of the struct, we set each part of the struct to the passed in arguments. You might ask, "What is each property of the struct used for?" First, the resourceName is a unique name passed into the class so each sprite can be easily referenced. If we were using any type of unmanaged code (XNA and C# is managed) we could use pointers and reference the direct place in memory that sprite is located but we can't so we'll leave it up to the call to AddSprite() to worry about having multiple sprites of the same name. (Note, we could have some type of checking around this method to make sure we don't add two items with the same

resourceName). SpriteTex is the actual sprite being used for this sprite object so we call Load<Texture2D> using the stored ContentManager (the ContentManager was stored in the constructor). Next, each piece of the screen position is filled using the two floats, x and y, which were passed in. Finally, rotation and scale are set to some basic float numbers, you'll see how this gets applied while rendering later.

```
gameSprites.Add(newSprite);
```

After the new sprite is filled out with all the values, it gets added to the list of all our game sprites by calling the List class's Add() method.

Notice that this List is a very quick and simple implementation of a sprite manager. If you have any experience with arrays or lists you know you can access elements in whichever type by their position such as:

```
scale = gameSprites[1].scale;
```

The reason this tutorial is using a unique sprite name instead of list position is because it's usually harder to reference something by just knowing a number and knowing you're getting the correct sprite back. Currently, we will be able to say "I want the map background sprite" and get it, instead of saying "I want sprite[0]" and getting something back which may not be the map background sprite.

UpdateSprite()

UpdateSprite() is used to update our sprite while it's still in the list which makes for a big larger code than normal (even more since we need to fill out the struct). Plus, this method has two similar methods, both of which are just as large. Here is the full first method:

```
public void UpdateSprite(string handle, string spriteName, float x, float y,
    float rotation, float scale)
{
    //Default variables
    int position = 0;
    bool found = false;

    //See if the sprite exists
    for (int i = 0; i < gameSprites.Count; i++)
    {
        if (gameSprites[i].resourceName == handle)
        {
            //Found the sprite
            found = true;
            position = i;
        }
    }

    //Can't update if we don't have a currently existing one!
    if (found)
    {
        //Create new sprite based on values
        sprite newSprite = new sprite();
        newSprite.resourceName = handle;
        newSprite.spriteTex = gameContent.Load<Texture2D>("Sprites\\" +
             spriteName);
        newSprite.screenPos.X = x;
        newSprite.screenPos.Y = y;
        newSprite.rotation = rotation;
        newSprite.scale = scale;

        //Swap new sprite into sprite position we want to update
        gameSprites[position] = newSprite;
```

```
        }
    }
```

The trick used in the UpdateSprite() method is to create a new sprite based on new values and swap out the old for the new. The way it's coded, the method first finds the sprite being swapped out and (assuming it's found) will update it with new values. If it is found, the method needs to store the position to update with the new. The position and determining if the sprite is found is stored in the first two variables of the method which are first initialized and then set if found.

```
int position = 0;
bool found = false;
```

Each item in the list is then run through to see if it matches the handle (or unique sprite name) given to it. If so, store position and set found to true.

```
for (int i = 0; i < gameSprites.Count; i++)
{
      if (gameSprites[i].resourceName == handle)
      {
            //Found the sprite
            found = true;
            position = i;
      }
}
```

It would be impossible to update a sprite that hasn't yet been added to the list, hence the use of our found boolean. Now that it's found, the method can swap out the sprite at that position with a new sprite.

```
if (found)
{
      //Create new sprite based on values
      sprite newSprite = new sprite();
      newSprite.resourceName = handle;
      newSprite.spriteTex = gameContent.Load<Texture2D>("Sprites\\"spriteName);
      newSprite.screenPos.X = x;
      newSprite.screenPos.Y = y;
      newSprite.rotation = rotation;
      newSprite.scale = scale;

      //Swap new sprite into sprite position we want to update
      gameSprites[position] = newSprite;
}
```

This code is quite similar to the add except it is passed into a specific spot of the list instead of just added to the end of the list.

Maybe the 'unattractiveness' of this idea is apparent to you now. Using the unique method name allows room for error where two sprites could be named the same. The way it is currently implemented, it would always update the last instance of that name and never the first (or any before it) because our for loop would always overwrite the position variable with the next found position. A work-around would be to pass in another integer number stating which instance of that sprite the method should access and then keep track in the for loop of which number is being found. In this case, the calling method would need to know which order each sprite was entered in the list anyways and be accessing by a number (similar to accessing list position by number and not name). This is all 'food for thought' and the reasoning behind using a string as a unique identifier. The HUD created in this tutorial won't utilize multiple HUD elements of the same name so we won't have to worry about that but check the tutorial exercizes for a question on adding a element number.

UpdateSpriteRotation()

UpdateSpriteRotation() does exactly what the method sounds like it does, it updates the rotation of a specific sprite within the list, found by checking the resourceName. Here's the full method:

```
public void UpdateSpriteRotation(string handle, string spriteName, float rotation)
{
      //Default variables
      int position = 0;
      bool found = false;

      //Look for sprite in list
      for (int i = 0; i < gameSprites.Count; i++)
      {
          if (gameSprites[i].resourceName == handle)
          {
              //Found sprite!
              found = true;
              position = i;
          }
      }

      //Can't update without an old sprite
      if (found)
      {
          //Create a new sprite and old sprite holder
          sprite newSprite = new sprite();
          sprite oldSprite = new sprite();

          //Store what we want to update
          oldSprite = gameSprites[position];

          //Set values of new sprite
          newSprite.resourceName = handle;
          newSprite.spriteTex = gameContent.Load<Texture2D>("Sprites\\" +
              spriteName);
          newSprite.screenPos.X = oldSprite.screenPos.X;
          newSprite.screenPos.Y = oldSprite.screenPos.Y;
          newSprite.rotation = rotation;
          newSprite.scale = oldSprite.scale;

          //Swap in new sprite
          gameSprites[position] = newSprite;
      }
  }
```

Notice the method begins similar by declaring a few variables and finding the instance of the sprite. Again, an update is not possible if the original doesn't exist so a sprite needs to be found before the update code actually happens. Instead of simply creating a new sprite and swapping it in, this method only updates the sprites rotation so the method needs to take the values of the old sprite and apply it to the new sprite in conjunction with the new rotation value. First, two sprite objects are created to hold the old and new sprites:

```
sprite newSprite = new sprite();
sprite oldSprite = new sprite();
```

Next, oldSprite is set to the sprite at the position we found the sprite to update in:

```
oldSprite = gameSprites[position];
```

Then, the method goes through setting each element of the newSprite either as a passed in value or as the value of the old sprite. Notice spriteName needs to be passed in because the oldSprite contains a Texture2D in spriteTex and we can't be sure the resourceName is the sprite file so the method requires the sprite file again unfortunately. This could easily be fixed by adding a string to store the file name in the sprite struct but to keep the size of the struct to a minimum, this tutorial doesn't implement that (see the final exercizes for a question dealing with this subject).

```
sprite newSprite = new sprite();
sprite oldSprite = new sprite();

//Store what we want to update
oldSprite = gameSprites[position];

//Set values of new sprite
newSprite.resourceName = handle;
newSprite.spriteTex = gameContent.Load<Texture2D>("Sprites\\" + spriteName);
newSprite.screenPos.X = oldSprite.screenPos.X;
newSprite.screenPos.Y = oldSprite.screenPos.Y;
newSprite.rotation = rotation;
newSprite.scale = oldSprite.scale;

//Swap in new sprite
gameSprites[position] = newSprite;
```

UpdateSpriteScale()

UpdateSpriteScale() is the same as UpdateSpriteRotation() but instead of updating the rotation of the sprite, this method updates the scale.

```
public void UpdateSpriteScale(string handle, string spriteName, float scale)
{
      //Default variables
      int position = 0;
      bool found = false;

      //Find sprite
      for (int i = 0; i < gameSprites.Count; i++)
      {
          if (gameSprites[i].resourceName == handle)
          {
              //Found sprite!
              found = true;
              position = i;
          }
      }

      //Can't swap if we don't have an old one
      if (found)
      {
          //Create a new sprite and old holder
          sprite newSprite = new sprite();
          sprite oldSprite = new sprite();

          //Store old sprite
          oldSprite = gameSprites[position];

          //Fill out new sprite
          newSprite.resourceName = handle;
          newSprite.spriteTex = gameContent.Load<Texture2D>("Sprites\\" +
spriteName);
```

```
            newSprite.screenPos.X = oldSprite.screenPos.X;
            newSprite.screenPos.Y = oldSprite.screenPos.Y;
            newSprite.rotation = oldSprite.rotation;
            newSprite.scale = scale;

            //Swap in new sprite
            gameSprites[position] = newSprite;
        }
    }
```

This tutorial won't go into detail about this method because of its similarities to the UpdateSpriteRotation() method. Simply notice the change when setting newSprite.scale.

DrawSprites()

DrawSprites() is the other most commonly used method of this class (the first being the AddSprite() method). It's purpose is to run the game's SpriteBatch and draw each element in the list. Here is the DrawSprites() method.

```
    public void DrawSprites()
    {
        //Begin our gameBatch using properties
        gameBatch.Begin(SpriteBlendMode.AlphaBlend, SpriteSortMode.Deferred,
            SaveStateMode.SaveState);
        for (int i = 0; i < gameSprites.Count; i++)
        {
            //Draw each item on the list
            gameBatch.Draw(gameSprites[i].spriteTex, gameSprites[i].screenPos, null,
                Color.White, gameSprites[i].rotation, new
                Vector2(gameSprites[i].spriteTex.Width / 2,
                gameSprites[i].spriteTex.Height / 2), gameSprites[i].scale,
                SpriteEffects.None, 0.0f);
        }

        //Finish game batch code, end
        gameBatch.End();
    }
```

DrawSprites() only works with the list and the SpriteBatch already stored in the class so no arguments are required for it to run. In order to begin drawing the SpriteBatch the method has to call the Begin() method which is part of the SpriteBatch class. Passed in are three properties that defined how the sprites are shown on the screen. These are severly important and without them the drawing on the screen would mess up and even effect the way any 3D models are rendered so make sure you get each of these in there.

```
    gameBatch.Begin(SpriteBlendMode.AlphaBlend, SpriteSortMode.Deferred,
        SaveStateMode.SaveState);
```

Between the Begin() and End() methods of the SpriteBatch object is the 'beefiest' few lines of code contained in this class. First, each sprite needs to be rendered so an obvious choice is to perform a for loop through the list. For each sprite in the list the SpriteBatch needs to call Draw() which takes 9 arguments (in the way this tutorial uses it). Before going through each item, take a look at the definition of the Draw() method:

```
    public void Draw ( Texture2D texture, Vector2 position, Nullable<Rectangle>
        sourceRectangle, Color color, float rotation, Vector2 origin, float scale,
        SpriteEffects effects, float layerDepth )
```

Since the sprite struct already contains the sprite texture, position, rotation, width, height, and scale all the method needs to do is pass them in the correct spots. The first two parameters are the texture and position

properties of the struct (since each sprite is being passed, each sprite can be accessed with gameSprites[index] where 'index' is the position in the list).

```
gameSprites[i].spriteTex, gameSprites[i].screenPos,
```

Third, the Draw() method accepts a Nullable<Rectangle> which tells the SpriteBatch which portion of the sprite to draw. This would be extremely useful if a single .PNG file contained many sprites (as many games usually have). Since each file created for this tutorial was created seperately that argument is set to 'null' which then draws the entire file.

```
gameSprites[i].spriteTex, gameSprites[i].screenPos, null,
```

Next, Color.White is passed to the Draw() method. If there was a sprite that needed a color overlay, for instance, needed to be drawn with a red tint, the Draw() will do this assuming the correct color is passed in. Note, the tint will apply to the full drawn sprite, not only certain colored parts.

```
gameSprites[i].spriteTex, gameSprites[i].screenPos, null, Color.White,
```

Rotation is passed in before the next argument which is a Vector2, accepting an origin point used to rotate the sprite around. This tutorial sets the origin to the center of the sprite by dividing the sprite height and width by 2.

```
gameSprites[i].spriteTex, gameSprites[i].screenPos, null, Color.White,
gameSprites[i].rotation, new Vector2(gameSprites[i].spriteTex.Width / 2,
gameSprites[i].spriteTex.Height / 2),
```

The final three arguments are pretty simple. Scale simply takes the scale to scale the file by (if no scale is applied, make sure to pass 1.0f and not 0.0f!). Effects allows SpriteEffects to be passed which consist of flipping the sprite horizontally or vertically, in this case no effect is used. Finally, layerDepth specifies whether to sort the sprite behind the rest or in front. 0 specifies the front which won't matter too much since we'll put the sprites on the list in the order we want them to draw.

```
gameSprites[i].spriteTex, gameSprites[i].screenPos, null, Color.White,
gameSprites[i].rotation, new Vector2(gameSprites[i].spriteTex.Width / 2,
gameSprites[i].spriteTex.Height / 2), gameSprites[i].scale, SpriteEffects.None,
0.0f
```

Add in the following and the DrawSprites() method is done.

```
gameBatch.End();
```

The DrawSprites() method is quite complex at first glance but looking at all the pieces it really only uses only what it needs and each piece is very simple.

ClearList()

ClearList simply removes all the objects from the list, similar to the cleanup code added in the destructor of the class.

```
public void ClearList()
{
    gameSprites.Clear();
}
```

It might not be apparent what this is used for currently but will be used as a major component in the next tutorial.

RemoveSprite()

RemoveSprite attempts to remove a sprite from the list using the unique handle stored in the sprite object.

```
public void RemoveSprite(string handle)
{
    //Default variables
    int position = 0;
    bool found = false;

    //Find sprite
    for (int i = 0; i < gameSprites.Count; i++)
    {
        if (gameSprites[i].resourceName == handle)
        {
            //Found sprite!
            found = true;
            position = i;
        }
    }

    //Can't swap if we don't have an old one
    if (found)
    {
        gameSprites.RemoveAt(position);
    }
}
```

Very similar to the UpdateSpriteRotation() and UpdateSpriteScale() methods, RemoveSprite searches through the list for the requested sprite and removes it from the list using the index (index and position in this sense can be used interchangeably).

Implementing the GameSprite Class

Back in the Game1 class create the following under the rest of your objects:

```
GameSprite gameSprites;
```

The GameSprite constructor will handle creating the SpriteBatch and store the game's ContentManager. Place this line within the LoadContent() method of the Game1.cs file. The spriteBatch object is initialized within the LoadContent() method as well, place the following line just below it.

```
gameSprites = new GameSprite(this.Content, spriteBatch);
```

Next, before rendering the sprites, they need to be added to the list in our class. First, make sure to add all the sprite files you want rendered to the Sprites folder added to the solution at the beginning of this tutorial. (The simple way is to right click on the folder, select "Add -> Existing Item…" and select the files). Since everyone might not have fresh .PNG's for their game, I've created some really basic (and ugly) ones you can find here: (http://www.insidegamer.org/documents/Sprites.zip). Now to initialize them and add them to the list, add the following below the previous line within the LoadContent() method of the Game1.cs file:

```
gameSprites.AddSprite("RadarBG", "RadarBG",
    graphics.GraphicsDevice.Viewport.Width - 100, 100, 0.0f, 0.7f);

gameSprites.AddSprite("RadarChar", "RadarChar",
    graphics.GraphicsDevice.Viewport.Width - 100, 100, 0.0f, 0.7f);
```

```
gameSprites.AddSprite("4Buttons", "4Buttons",
    graphics.GraphicsDevice.Viewport.Width - 90,
    graphics.GraphicsDevice.Viewport.Height - 90, 0.0f, 1.0f);

gameSprites.AddSprite("HealthBG", "HealthBG", 140,
    graphics.GraphicsDevice.Viewport.Height - 100, 0.0f, 1.0f);

gameSprites.AddSprite("HealthBorder", "HealthBorder", 140,
    graphics.GraphicsDevice.Viewport.Height - 100, 0.0f, 1.0f);
```

Remember, AddSprite() is passed the unique handle, the sprite filename, the x and y position, the rotation, and the scale. The screen is referenced with graphics.GraphicsDevice.Viewport.Height/Width and since all HUD elements are on the basic viewport the screen is the base reference used to place each object. Each AddSprite won't be dissected and talked about but be sure to notice this point, the X and Y values passed to the list are the coordinates **relative the the middle of the sprite** because Draw() was told to place the sprite by the middle. If the Y coordinate is set to graphics.GraphicsDevice.Viewport.Height, only half of the sprite will show up. To compensate, the radius of the sprite is subtracted from the screen width/height. Finally, since only one of each sprite is used, each sprite's unique name is set to the filename.

Add a global variable towards the top of the Game1 class under your objects:

```
float previousRotation = 0.0f;
```

Lastly, in the Draw() method of the Game1 class add the following lines:

```
if (previousRotation != camera.GetRotation())
{
      gameSprites.UpdateSpriteRotation("RadarBG", "RadarBG", camera.GetRotation());
      previousRotation = camera.GetRotation();
}

gameSprites.DrawSprites();
```

The code above detects a change in the rotation of the camera and if it changes, the background of the radar rotates as it does in many first-person games. Then previousRotation is set so next run doesn't update the rotation when it doesn't need to. Finally, all sprites are drawn with the DrawSprites() method.

Updating the GameCamera Class

There's one quick and simple method to add to the GameCamera class, the GetRotation() method used above when drawing the sprites. It is as follows:

```
public float GetRotation()
{
      return turnAmt;
}
```

All that it is doing is passing back the turnAmt float stored in the class.

Conclusion

Creating the basics of a simple HUD introduced 2D elements into XNA programming. Traditionally, programmers begin making 2D games and even after this tutorial a good idea might be to go back and create a basic game using 2D and the SpriteBatch items. Eventually, instantiated classes could be made for, example, the map and all the little 'quest points' we could place on it. The GameSprite class provides lots of options for later improvements including menus.

Suggested Exercises

1) Using a photo editor make a simple crosshairs and place them in the center of the screen. Also try making some numbers for the health portion and place them in appropriately (Allow numbers from 00 – 100).

2) Update the struct to store the file name of the sprite passed in the AddSprite() method. Then, change the UpdateSpriteRotation() and UpdateSpriteScale() to only accept the handle and rotation/scale arguments. [HINT: Use oldSprite.fileName]

3) Update the UpdateSprite() method to accept an int named 'elementNum' which will access a specified instance of a sprite with a similar name. (For example: the list contains 5 sprites with resourceName = "buttonBorder", if elementNum = 3 the method will access the 3rd instance of the sprite and not the final instance). Make sure to include updates to the UpdateSpriteRotation() and UpdateSpriteScale() methods.

4) Create a new method that does the work of both the UpdateSpriteRotation() and UpdateSpriteScale() methods. Call it UpdateSpriteRotationScale() and pass it three arguments. [**Note:** If you haven't completed Exercise 2 this method will require four arguments instead of three]

5) Update the RemoveSprite() method to accept an int named ‘elementNum’ which removes an instance of the specified sprite object.