Dungeon Master Database Creator

This project is designed to create a database of all the data needed for the Monogame version of Dungeon Master.

It uses the Nuget package sqlite-net-pcl by SQLite-net.

To create a sqlite3 database:

```
1. using SQLite;
2. SQLiteConnection db;
3. db = new SQLiteConnection(dbPathAndFile);
```

If the file already exists it will be opened.
If not a new database will be created.

This package uses the idea of working with classes to populate and read the database.
For example: Source Rectangles:

```
public class DbSourceRectangle
{
    [PrimaryKey]
    public string Name { get; set; }
    public string Spritesheet { get; set; }
    public int X { get; set; }
    public int Y { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }
    public DbSourceRectangle() { }
    public DbSourceRectangle( string name, string spritesheet, int x, int y, int w, int h )
    {
        Name = name;
        Spritesheet = spritesheet;
        X = x;
        Y = y;
        Width = w;
        Height = h;
    }
}
```

This class is used for creating and storing source rectangles from a spritesheet.
There are 2 string properties used for the name of the source rectangle, eg “Wall.C3” and the name of the spritesheet it can be found eg “WallsLayer3+4.png”.

There ar 4 integer properties for the X, Y coordinates and the Width, Height of the rectangle.

For the database to work properly you must supply a default constructor without parameters

```
public DbSourceRectangle() { }
```

Optionally add a working constructor to use when creating real objects.

Once this class has been added, you can add a table to the database using this class:

```
db.CreateTable<DbSourceRectangle>();
```

That's it.

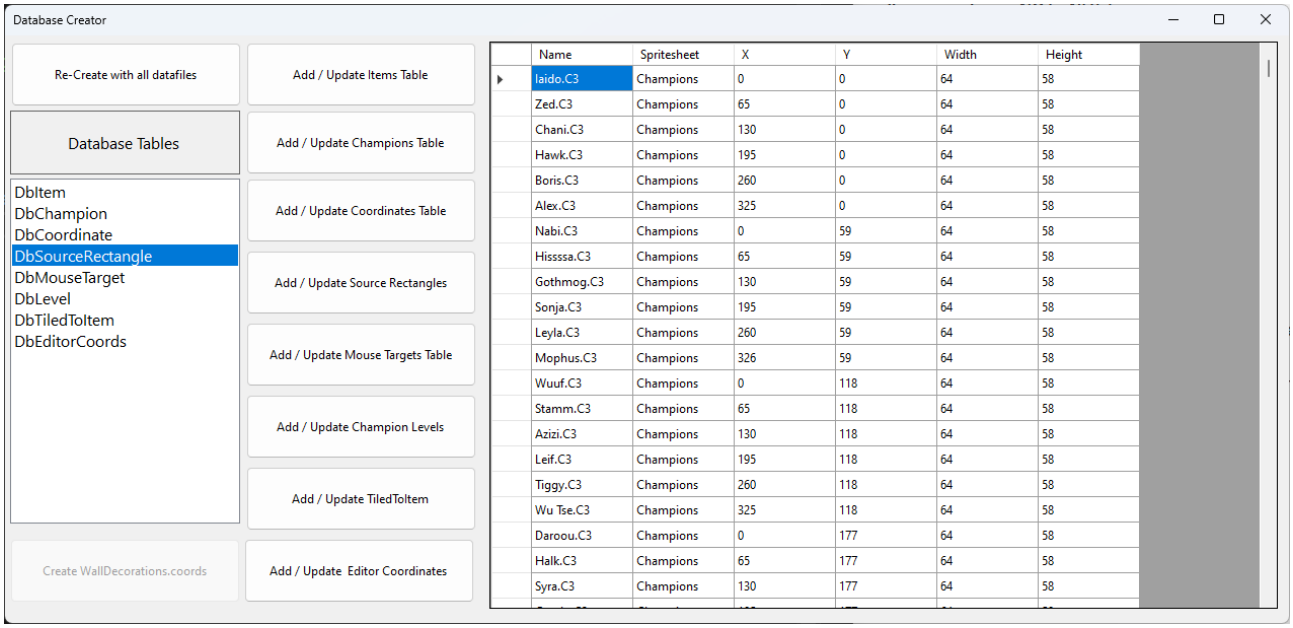
This has the same effect as using this SQL statement

```
CREATE TABLE "DbSourceRectangle" (
   "Name"  varchar NOT NULL,
   "Spritesheet"  varchar,
   "X"  integer,
   "Y"  integer,
   "Width" integer,
   "Height"  integer,
   PRIMARY KEY("Name")
);
```

To add a new SourceRectangle to the database use:

```
db.Insert(new DbSourceRectangle(name, spritesheet, X, Y, Width, Height));
```

This is what the project looks like when started, and a table selected:



DBCreator Tutorial

Start a new C# Windows Forms Desktop project.

Name it DBCreator

Add the Nuget package sqlite-net-pcl by SQLite-net.

Add a new Directory called Models (All the DB classes go here)

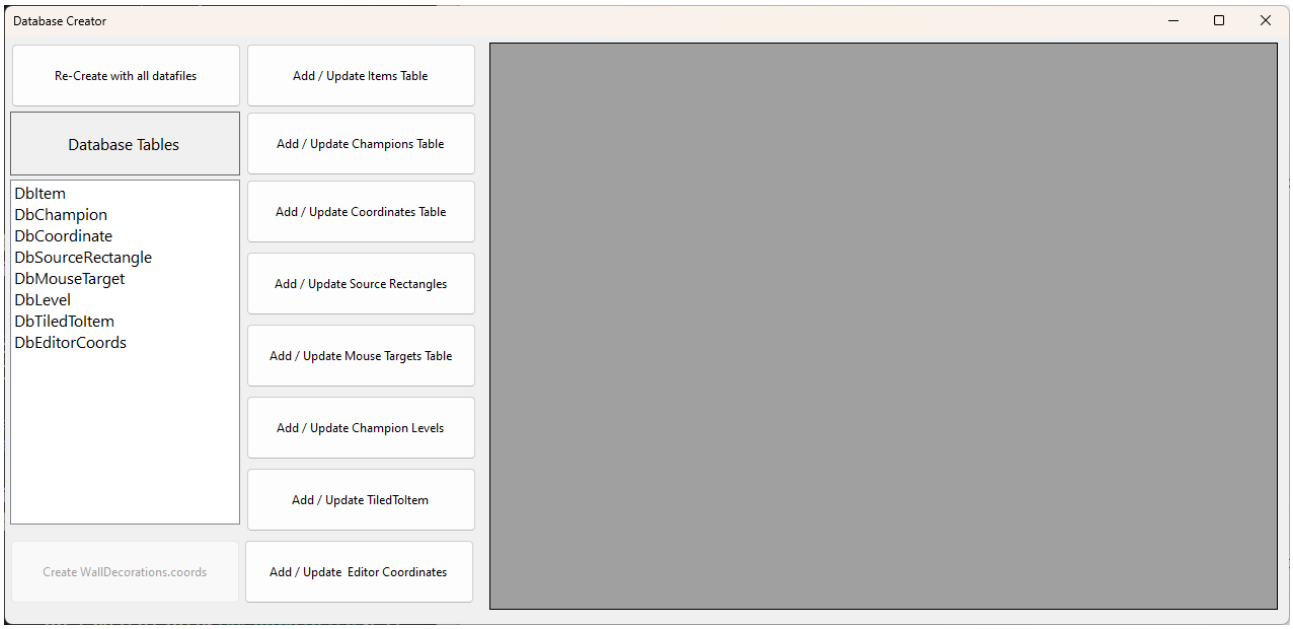
Over-write the frmMain.cs, frmMain.Designer.cs and frmMain.resx with the files from Github:

https://github.com/Inksaver/MonogameDungeonMaster/tree/main/DBCreator/DBCreator

Add all the files in the Models folder

Check Namespace matches on all files (DBCreator)

frmMain loads and reaches the Shown() event:



The subroutine Initialise() is called:

This checks for the existence of a file called “IOPaths.txt”
If it exists the paths for locating the text files used to create the database and the location of the database itself are configured.

If it does not exist CreateConfig() is called which uses a FolderBrowserDialog to find the correct source and destination folders and write them to a text file called IOPaths.txt.

The final part of Initialise() is to check for the existence of the database defined in the file using CheckDatabase()

```
private void CheckDatabase()
{
    if (!File.Exists(dbPathAndFile))
    {
        db = new SQLiteConnection(dbPathAndFile);
        db.CreateTable<DbItem>();
        db.CreateTable<DbChampion>();
        db.CreateTable<DbCoordinate>();
        db.CreateTable<DbSourceRectangle>();
        db.CreateTable<DbMouseTarget>();
        db.CreateTable<DbLevel>();
        db.CreateTable<DbEditorCoords>();
        db.CreateTable<DbTiledToItem>();
    }
    else
        db = new SQLiteConnection(dbPathAndFile);

    FillListBox();
}
```

If the file exists, the database is opened. If not a new database is created then opened.
Creating tables is very simple using the class name as a type eg:

```
   db.CreateTable<DbItem>();
```

If the databse is new the Database tables listbox is empty. The screenshot above shows a populate database, with the Tables listed via `FillListBox();`

```
private void FillListBox()
{
    lstTables.Items.Clear();
    if (File.Exists(dbPathAndFile))
    {
        if (TableExists("DbItem"))
            lstTables.Items.Add("DbItem");
        if (TableExists("DbChampion"))
            lstTables.Items.Add("DbChampion");
        if (TableExists("DbCoordinate"))
            lstTables.Items.Add("DbCoordinate");
        if (TableExists("DbSourceRectangle"))
            lstTables.Items.Add("DbSourceRectangle");
        if (TableExists("DbMouseTarget"))
            lstTables.Items.Add("DbMouseTarget");
        if (TableExists("DbLevel"))
            lstTables.Items.Add("DbLevel");
        if (TableExists("DbTiledToItem"))
            lstTables.Items.Add("DbTiledToItem");
        if (TableExists("DbEditorCoords"))
            lstTables.Items.Add("DbEditorCoords");
    }
}
```

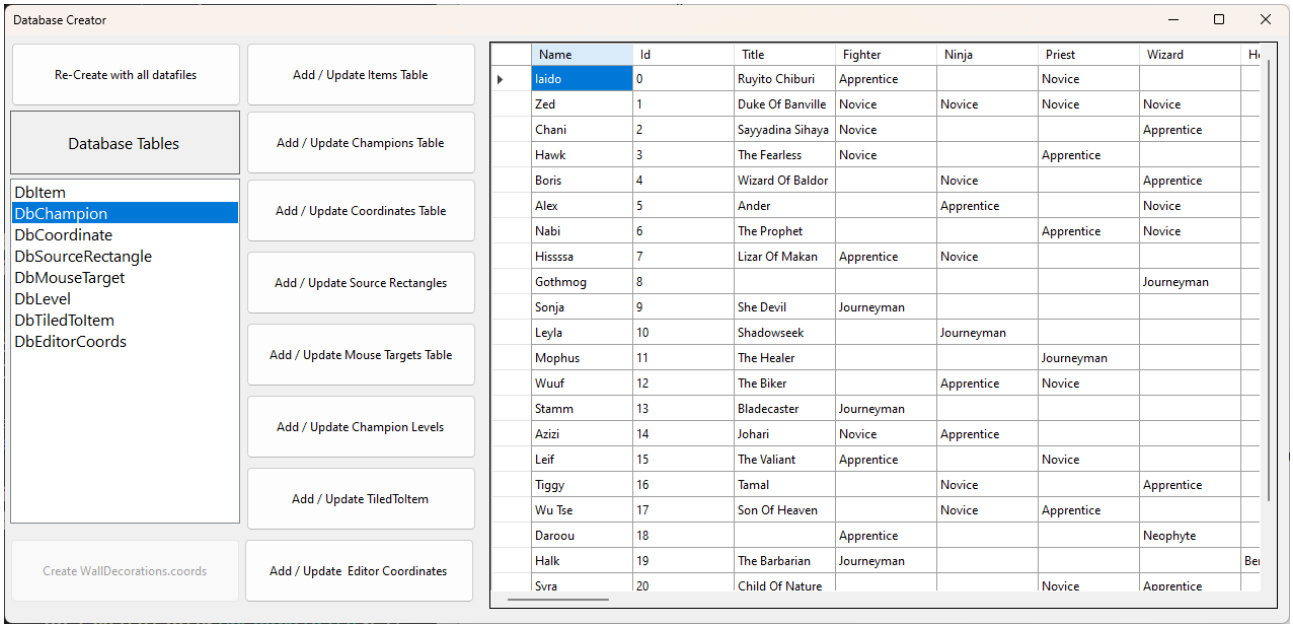
This calls TableExists() in a selection statement and adds to the ListBox if the table exists.
The table names are hard-coded.

```
public bool TableExists(string tableName)
{
    /// Checks the database to see if the table exists
    TableMapping map = new TableMapping(typeof(SqlDbType)); // Instead of mapping to a specific table
just map the whole database type
    object[] ps = new object[0]; // An empty parameters object

    int tableCount = db.Query(map, "SELECT * FROM sqlite_master WHERE type = 'table' AND name = '" +
tableName + "'", ps).Count; // Executes the query from which we can count the results
    if (tableCount == 0)
        return false;
    else if (tableCount == 1)
        return true;
    else
        throw new Exception("More than one table by the name of " + tableName + " exists in the
database.", null);
}
```

Selecting one of the tables populates the Datagrid via the event handler:

```
private void lstTables_Click(object sender, EventArgs e) { DisplayTable(); }
```




```
private void DisplayTable()
{
    string tableName = lstTables.Text;
    if (tableName == "DbChampion")
        dgvDMData.DataSource = db.Table<DbChampion>().ToList<DbChampion>();
    else if (tableName == "DbCoordinate")
        dgvDMData.DataSource = db.Table<DbCoordinate>().ToList<DbCoordinate>();
    else if (tableName == "DbEditorCoords")
        dgvDMData.DataSource = db.Table<DbEditorCoords>().ToList<DbEditorCoords>();
    else if (tableName == "DbItem")
        dgvDMData.DataSource = db.Table<DbItem>().ToList<DbItem>();
    else if (tableName == "DbLevel")
        dgvDMData.DataSource = db.Table<DbLevel>().ToList<DbLevel>();
    else if (tableName == "DbMouseTarget")
        dgvDMData.DataSource = db.Table<DbMouseTarget>().ToList<DbMouseTarget>();
    else if (tableName == "DbSourceRectangle")
        dgvDMData.DataSource = db.Table<DbSourceRectangle>().ToList<DbSourceRectangle>();
    else if (tableName == "DbTiledToItem")
        dgvDMData.DataSource = db.Table<DbTiledToItem>().ToList<DbTiledToItem>();
}
```

The selected table is used to set the Datagrid datasource to the Table with the same name.

Tables can be populated individually by using the corresponding button. This is useful if a text file has been edited and the data needs to be re-built.

Using the button Add Source Rectangles as an example (Event handler) :

```
private void btnAddSourceRects_Click(object sender, EventArgs e) { AddSourceRectangles(true); }

private void AddSourceRectangles(bool replace)
{
    this.Cursor = Cursors.WaitCursor;
    if (replace)
    {
        if (TableExists("DbSourceRectangle"))
            db.DropTable<DbSourceRectangle>();
        db.CreateTable<DbSourceRectangle>();
    }
    foreach (string fileName in ImagePaths)
    {
        if (File.Exists(Path.Combine(dataSource, $"{fileName}.data")))
        {
            List<string> lines = ProcessTextFile(Path.Combine(dataSource,
$"{fileName}.data"));
            foreach (string line in lines)  // eg Wood.C3,0,0,202,213
            {
                string[] data = line.Split(",", StringSplitOptions.RemoveEmptyEntries);
                if (int.TryParse(data[1], out int X) &&
                    int.TryParse(data[2], out int Y) &&
                    int.TryParse(data[3], out int Width) &&
                    int.TryParse(data[4], out int Height))
                {
                    Insert(new DbSourceRectangle(data[0], fileName, X, Y, Width, Height));
                }
            }
        }
    }
}
```

If the replace flag is set, the original table is deleted and re-created.

The foreach loop iterates the relevant source text files with the extension .data, reads them in via

`ProcessTextFile()`

```
private List<string> ProcessTextFile(string filename)
{
    List<string> retValue = new List<string>();

    string[] lines = File.ReadAllLines(filename);
    foreach (string line in lines)
    {
        if (!line.StartsWith('#') && line.Trim().Length > 0)
        {
            retValue.Add(line);
        }
    }
    return retValue;
}
```

to strip out comments and empty lines returning a List<string>

This list is then returned to `AddSourceRectangles()`
from which the name(key), X, Y , Width and Length are extracted and a new DbSourceRectangle object created and inserted into the database with `db.Insert(obj);`

A similar process is used on all the tables and text files. Each one is written to deal with the specific file that is being read.

To create or re-create the whole database, use the button “Re-Ceate with all datafiles”
This will delete the original database and re-load all files in the source directory, with a confirmation dialog between each table added.