

# Blocks MonoGame Tutorial Part 2

## Contents

|  |    |
|--|----|
| Blocks MonoGame Tutorial Part 2 .....                            | 1  |
| Object-Oriented Programming in 1 Minute.....                     | 1  |
| Part 2 – Content Pipeline: Load Sounds and Images .....          | 2  |
| Image Files.....   | 3  |
| Sound Files .....  | 3  |
| Add Content .....  | 3  |
| If you see a text file after double clicking Content.mgcb: ..... | 4  |
| Add Fonts.....   | 6  |
| GameContent Class: Loading Images and Sounds .....               | 8  |
| Assignment Submission.....                                       | 11 |

Time required: 60 minutes

## Object-Oriented Programming in 1 Minute

In this series of tutorials we are diving head first into object-oriented programming without a life jacket, a paddle, or a net.

**Object-oriented programming (OOP)** is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

A class is a blueprint from which we can create as many objects as you like.

A class contains data and behavior (methods).

**Class:** House

**Data:** Address, Color, Area

**Methods:** Open door, close door

**Class:** Car

**Data:** Color, Brand, Weight, Model

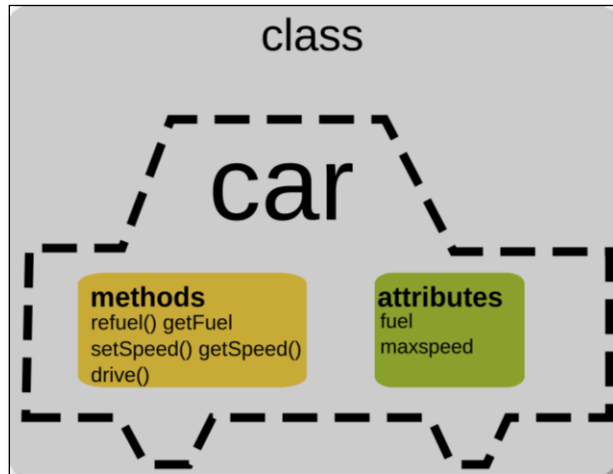
**Methods:** Brake, Accelerate, Slow Down, Gear change

When we create an object (MyCar) from a class, we can use it to store data and do actions.

Create Object: MyCar

```
MyCar.Color = Red    // Store the color of my car
```

```
MyCar.Brake()        // Put on the brakes
```



We can create as many objects as needed by the program without rewriting a bunch of code. We can use the object without having to understand what is inside the object, we just have to know how to use it.

## Part 2 – Content Pipeline: Load Sounds and Images

In previous section, we created a new MonoGame project, and looked at what was created for us to start with. Let's take a look at the MonoGame Content Pipeline, and why we need it for our game.

Although we could load image and sound files directly from the file system in MonoGame, that's not the best way to do it. The content pipeline provides several advantages:

1. The pipeline tool converts files to a format that MonoGame can use at run-time. For example, MonoGame can't process ".wav" sound files at run-time, we use the Pipeline tool to convert the file to the ".xnb" format that MonoGame knows how to use.
2. The pipeline tool will reformat the files into a format appropriate for the target platform. Remember, even though this tutorial is targeting Windows, you can also use MonoGame to target iOS, Android, Mac OS, and Linux.

3. The pipeline tool will also strip out information from the file that isn't needed by the run-time. This reduces the file size on disk and can reduce load time.

### Image Files

- Ball.png (our game ball)
- Block.png (image of the Blocks on our gameboard)
- Paddle.png (the game paddle used to hit the ball)
- Pixel.png (a file with a single white pixel. We'll use this for drawing shapes. We'll talk about this later)

### Sound Files

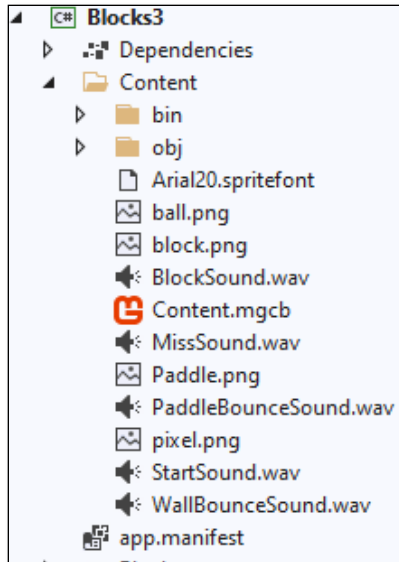
- StartSound.wav (played when we serve a new ball)
- WallBounceSound.wav (played when the ball bounces off a wall)
- Blocksound.wav (played when a block is hit and destroyed)
- MissSound.wav (played when the player misses the ball and it falls out of play)
- PaddleBounceSound.wav (played when the ball bounces off of the paddle)

---

## Add Content

**BlocksAssets.zip** is attached to the assignment.

1. Extract the image and sound files.
2. Copy and paste them into your project **Content** folder:



We will working with the **Content.mgcb** file to add the assets to the program.

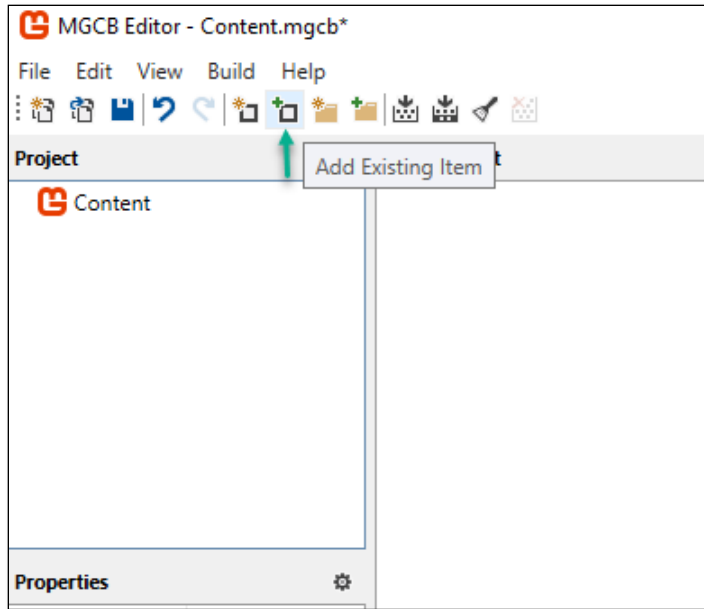
**Content.mgcb** is an XML file that the pipeline tool uses to store information about your project file assets. To use assets (sound and graphics) in a MonoGame Project, we need to add the files and build them using the MGCB (Monogame Framework Content Builder) Editor.

1. Double-click **Content.mgcb** to open the MGCB Editor.

**If you see a text file after double clicking Content.mgcb:**

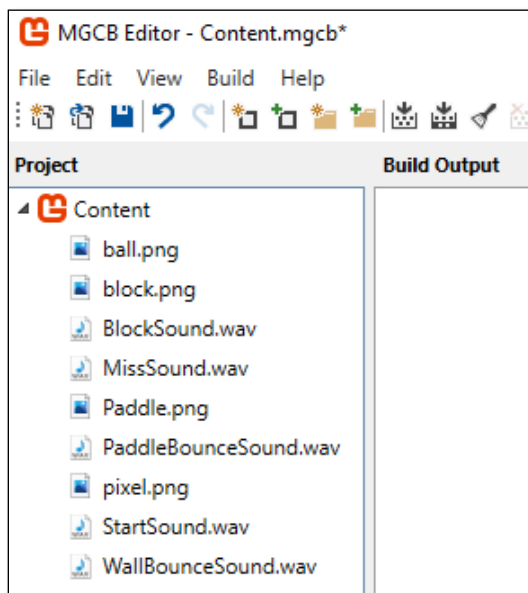
1. Right Click **Content.mgcb** → **Open with**
2. Find and select **mgdb-editor-wpf**
3. Click Set as Default. Click OK.
4. Double Click **Content.mgcb**

Your MGCB Editor tool should now look like this:



5. Click the **Add Existing Item** button as shown above.
6. Select all of the ".PNG" and ".WAV" files you copied into your Content folder.  
NOTE: You can select multiple items, by holding down the **Ctrl** key while clicking on each file name.
7. Click **Open**.

Your MGCB Editor should now look like this:



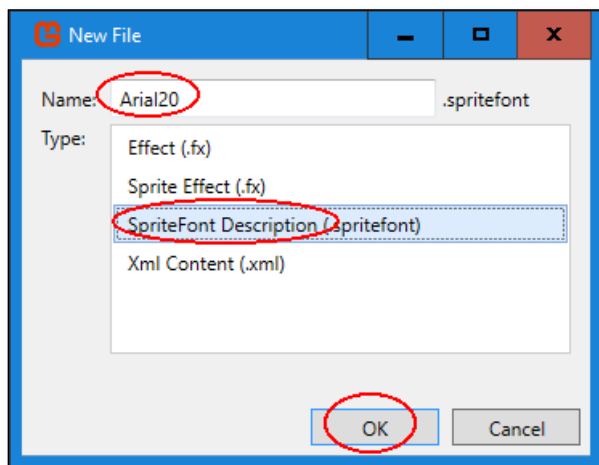
---

## Add Fonts

MonoGame doesn't write text to the screen the way a normal windows program does. It uses a bitmap font. You use the MonoGame MGCB Editor to create a bitmap image of each character in the character set, as it will look on the screen. It then draws those bitmap images on the screen for you.

That has some implications for you. You will need to add a font asset for each font family and font size you will use in your program. For this program, we'll be using the "Arial" font, with a point size of "20". Fortunately, adding a font is easy.

1. Double Click **Content.mgcb** to open the MGCB Editor.
2. Click the **Add New Item** button.
3. In the **New File** Dialog, enter **Arial20** as the name.
4. From the list of **Types**, click on **Sprite Font Description**. Click **OK**.



MonoGame added a file called **Arial20.spritefont** to your Content folder.

MonoGame will default to "Arial" when we add a font. We'll need to manually change the font size. We will need to edit the file we just created using a text editor. Normally in Windows we would use Notepad.

The problem with windows Notepad is that it doesn't recognize the "Newline" characters in the file, so it will look jumbled. We will use **Visual Studio Code**.

The file will be in the **Content** folder of your project, and will be called "Arial20.spritefont".

1. Open the file with **Visual Studio Code**. (Or another text editor.)

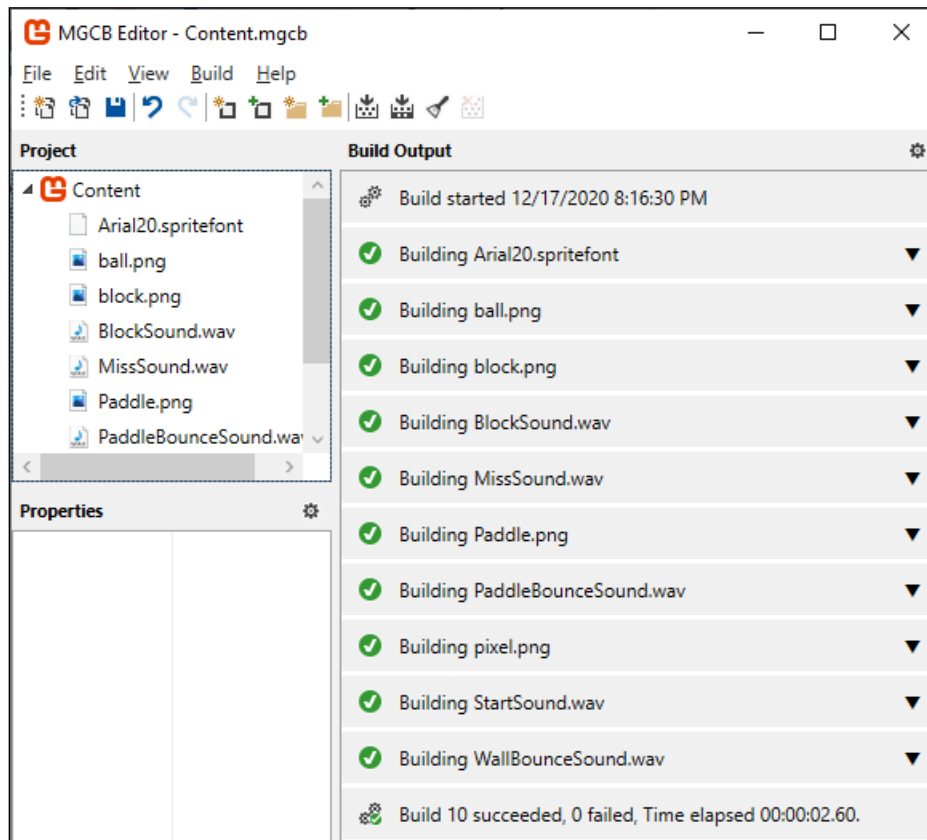
2. The file is a small XML file. The tag we need to change is the **Size** tag. Change the value of the tag to **20**. Save the file.

```
<!--  
Size is a float value, measured in points. Modify this value to change  
the size of the font.  
-->  
<Size>20</Size>  
  
<!--
```

The last step is to convert the asset files to “.xnb” files, so they can be used in MonoGame.

3. In the **MGCB Editor**, select the **Build** button.

After the build, your screen should look like this:



There are two things to remember about content.

1. If you add any assets to your content folder, you must go back to the MGCB Editor and add them. You must do a **Build** again.

2. If you make any changes to any of the assets you previously added, do a **Build** again. The program isn't working with the image you edited, it's working with the **.xnb** file that **Build** creates from that image.

Now that we've gotten the files into a format that MonoGame can use, let's see how we actually load them into our program.

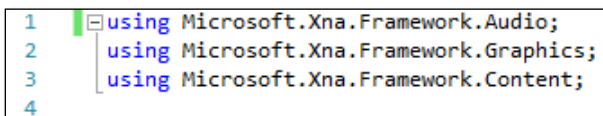
---

## GameContent Class: Loading Images and Sounds

Before we can use images and sounds in our game, we have to load them. An advantage of using the content pipeline and content loader, is that all of our images and sounds will be cached in memory, and can be used quickly when we need them.

Let's create a new class called **GameContent**. This class will do all of the content loading into memory for our game.

1. Right click on the **Blocks** project in the **Solution Explorer** → **Add** → **Class** from the drop-down menus.
2. In the **Add New Item** Dialog, enter **GameContent.cs** for the file name and click the **Add** Button.
3. Visual Studio will open the empty class in an edit window
4. Add the following using statements to the top of the class. You can remove any other statements.



```
1 using Microsoft.Xna.Framework.Audio;
2 using Microsoft.Xna.Framework.Graphics;
3 using Microsoft.Xna.Framework.Content;
4
```

These statements add the MonoGame Framework to this class.

Wait a minute! Aren't we using MonoGame, not XNA? MonoGame is fully compatible with XNA 4. To make sure that existing code would work with MonoGame, the MonoGame creators decided to use the Namespace names that XNA used. This also means that as you work with MonoGame and come across XNA examples, you should be able to use them just fine in MonoGame.

5. Add the following reference variables/properties to the class. This allows our game to access the images and sounds we added earlier to the Content.mcgb. Don't change your namespace.



```

1  using Microsoft.Xna.Framework.Audio;
2  using Microsoft.Xna.Framework.Graphics;
3  using Microsoft.Xna.Framework.Content;
4
5  namespace Blocks2
6  {
7      2 references
8      class GameContent
9      {
10         // Create reference variables/properties for each content asset
11         1 reference
12         public Texture2D imgBlock { get; set; }
13         1 reference
14         public Texture2D imgPaddle { get; set; }
15         1 reference
16         public Texture2D imgBall { get; set; }
17         1 reference
18         public Texture2D imgPixel { get; set; }
19         1 reference
20         public SoundEffect startSound { get; set; }
21         1 reference
22         public SoundEffect blockSound { get; set; }
23         1 reference
24         public SoundEffect paddleBounceSound { get; set; }
25         1 reference
26         public SoundEffect wallBounceSound { get; set; }
27         1 reference
28         public SoundEffect missSound { get; set; }
29         1 reference
30         public SpriteFont labelFont { get; set; }
31     }
32 }

```

6. In the GameContent class, create a class constructor as shown. This loads all the game content into the ContentManager and into memory.

```

21 // Constructor loads content into ContentManager
22 1 reference
23 public GameContent(ContentManager Content)
24 {
25     // Load images into the reference variables
26     imgBall = Content.Load<Texture2D>("Ball");
27     imgPixel = Content.Load<Texture2D>("Pixel");
28     imgPaddle = Content.Load<Texture2D>("Paddle");
29     imgBlock = Content.Load<Texture2D>("Block");
30
31     // Load sounds into the reference variables
32     startSound = Content.Load<SoundEffect>("StartSound");
33     blockSound = Content.Load<SoundEffect>("BlockSound");
34     paddleBounceSound = Content.Load<SoundEffect>("PaddleBounceSound");
35     wallBounceSound = Content.Load<SoundEffect>("WallBounceSound");
36     missSound = Content.Load<SoundEffect>("MissSound");
37
38     // Load fonts into the reference variables
39     labelFont = Content.Load<SpriteFont>("Arial20");
40 }
41 }

```

We put **Content.Load** statements in our **GameContent** class constructor. The game assets will all be loaded into memory when a **GameContent** object is created.

We pass a reference to the **ContentManager**, because all of the assets we are loading will be managed by that class. The rest of the method uses the **ContentManager Load** method to load all the asset files.

**Note:** We must tell MonoGame the type of file we are loading:

- Images: Texture2D
- SoundEffects: SoundEffect
- Font: SpriteFont

The name we pass as the argument matches the file names we added in the pipeline tool (minus the file extension). In our **Game1** class **ContentLoad** method, we'll create a new instance/object of the **GameContent** class, which will load all of the assets we need into the **ContentManager**.

1. In the **Game1** class, add a new variable/reference called **\_\_gameContent**. This allows our main game class to access the content.

```
1reference
public class Game1 : Game
{
    // Create reference variables
    private GraphicsDeviceManager _graphics;
    private SpriteBatch _spriteBatch;
    private GameContent __gameContent;
```

2. Create an instance/object of **GameContent** in the **Game1 LoadContent** Method. This allows the game to use a **GameContent** object to load, manage, and use the game content.

```
protected override void LoadContent()
{
    // Create a SpriteBatch object which is used to draw textures.
    _spriteBatch = new SpriteBatch(GraphicsDevice);

    // Create a GameContent object that references the content assets
    __gameContent = new GameContent(Content);
}
```

With these changes, our program has now loaded all of the assets we'll need into the **ContentManager**.

Run your program now, just to make sure it still brings the game up with the blue-background window. If you get an error, you probably forgot to add a file to the pipeline tool, or you forgot to do a Build using the pipeline tool. If you get an error, go back and make sure you've done all of the steps correctly.

If your game still runs properly, we're ready to actually start doing some drawing. Let's tackle that in the next tutorial.

---

## **Assignment Submission**

Zip up the pseudocode and the project folder. Submit in Blackboard.