MonoGame Blocks Tutorial

Contents

Mission Impossible

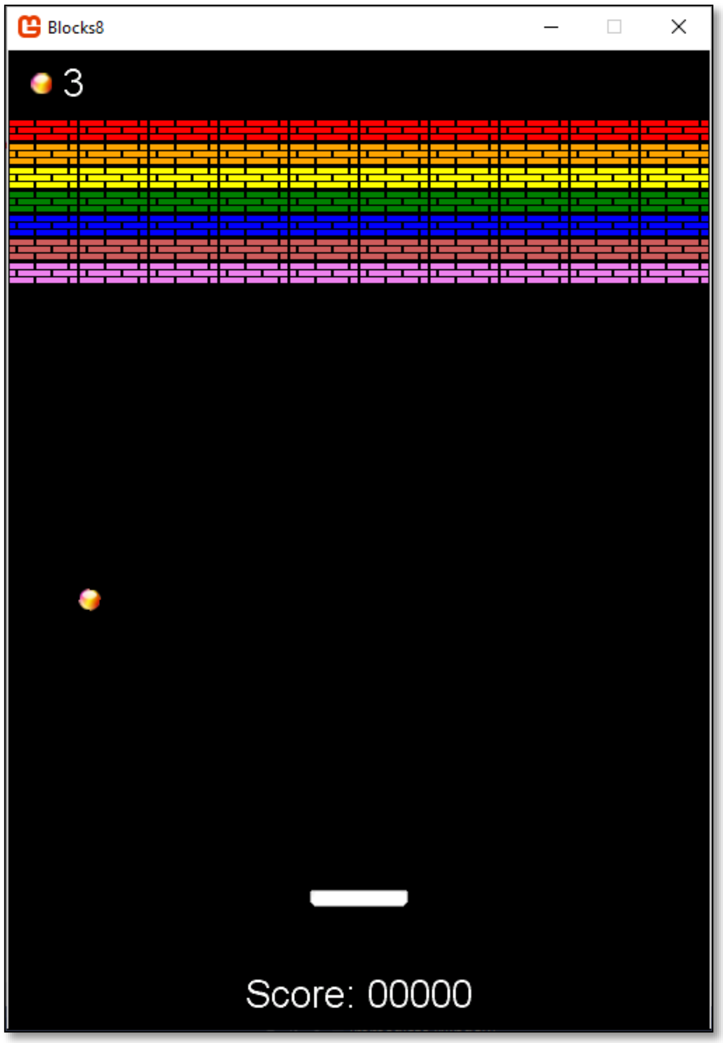
The mission if you choose to accept it:

Save the earth from a Blocks invasion. You must create a computer game to destroy the deadly Blocks invaders before it is too late!

The fate of the world rests in your hands.

Game Preview

This is what the finished product will look like.



Part 1 – MonoGame Tutorial

Time required: 60 minutes

Why and What is MonoGame?

Microsoft C# and .Net are excellent development tools for creating Windows desktop and web-based applications. C# falls short in game development. Using the .Net API's, it is not possible to get the high performance required for gaming.

DirectX allows low-level access of the Video Card, which is fastest method of drawing on the screen for the PC. DirectX, which is used by virtually all professional games on Windows, requires the use of the C++ language. C++ is a difficult language to use.

In 2006, Microsoft provided a solution: Microsoft XNA Framework. XNA allows C# developers to access DirectX from C#. C# is now a very credible solution for gaming development. It is used by many commercial games today.

Microsoft discontinued XNA development in 2013.

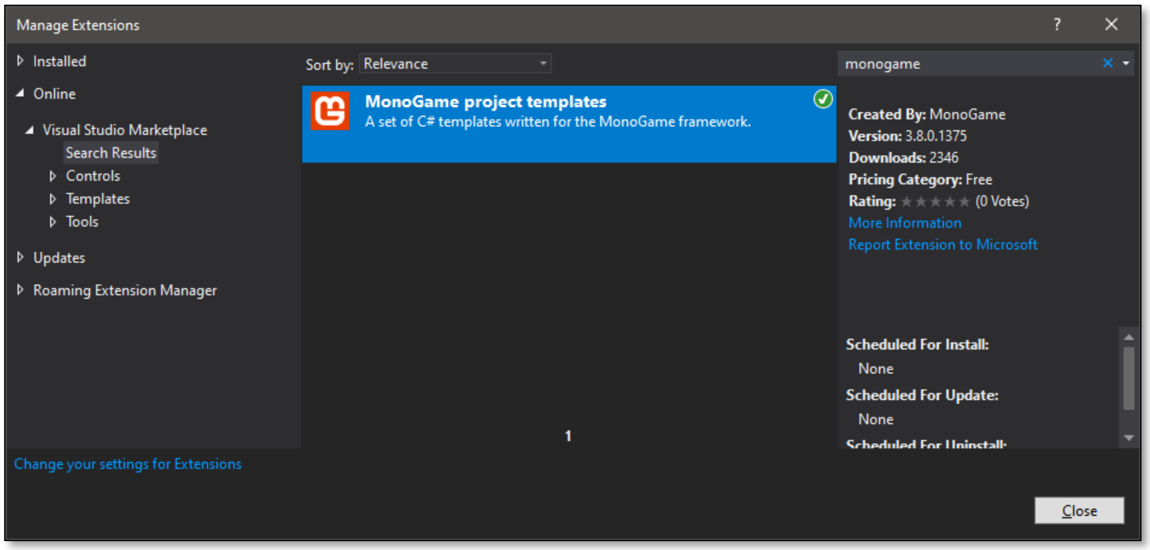
MonoGame is a compatible open source version of XNA. It is still being supported. MonoGame is cross-platform. You can develop games in C# for iOS, Android, Mac OS X, Linux, and Windows.

Go to www.monogame.net for more information.

Install MonoGame project templates for Visual Studio 2019

It is time to get started with your game development career. Let’s install the Visual Studio 2019 MonoGame project templates extension.

1. In **Visual Studio**, Click **Continue without code**.
2. Go to the menu bar to **Extensions → Manage Extensions**
3. Search for **MonoGame**. Click **MonoGame project templates**. Click **Download**.
4. Finish the installation by Closing Visual Studio.
5. You will be asked to Modify Visual Studio. Click **Modify**.



MonoGame project templates
A set of C# templates written for the MonoGame framework.

Created By: MonoGame
Version: 3.8.0.1375
Downloads: 2346
Pricing Category: Free
Rating: ★★★★★ (0 Votes)

Scheduled For Install:
None
Scheduled For Update:
None
Scheduled For Uninstall:

Setup the MGCB Editor

MGCB Editor is a tool for editing .mgcb files, which are used for building content.

Register the MGCB Editor tool with Windows and Visual Studio 2019 by running the following 2 commands from the Command Prompt. These should both be successful.

```
dotnet tool install --global dotnet-mgcb-editor
mgcb-editor –register
```

The Blocks Game

Breakout is one of the earliest video games ever developed. It offers interesting game play and the mechanics are relatively simple. It is a good subject for a beginning MonoGame tutorial.

In the game, there is a wall of several rows of Blocks at the top of the screen. A ball is bounced off a movable paddle at the bottom of the screen. It travels across the screen. If the ball hits the side or top walls of the screen it bounces. If it hits a block, the block is destroyed. If the player fails to hit the ball with the paddle, the ball flies off the bottom of the screen and is lost. The player gets 3 balls. Play continues until all three balls are lost. After all Blocks are cleared, a new set of Blocks is displayed, and play continues.

Tutorial Road Map

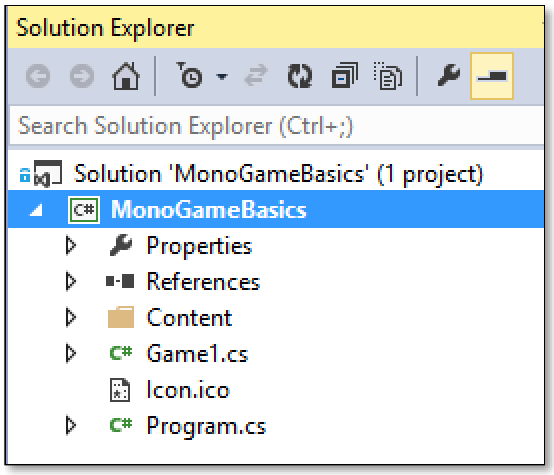
In this series of tutorials, we will cover the following topics:

1. Overview and Setting up the Environment
2. Basic structure of a MonoGame Program (The game loop)
3. Content Pipeline Tool (How we get images and sound effects usable by MonoGame)
4. Loading Images and Sounds
5. Drawing Images on the Screen
6. Drawing basic shapes on the screen
7. Getting Keyboard and Mouse Inputs
8. Moving and Rotating Images
9. Playing sound effects
10. Drawing Text

Getting Started with Blocks

We start by creating a new Visual C# MonoGame project called “Blocks” in Visual Studio.

1. In Visual Studio, Click the **File** Menu → **New** → **Project**:
2. This will open the **Create a new project** dialog.
3. Search for and select **MonoGame Windows Desktop Application (Windows DirectX).** Click **Next**.
4. In the **Project name** field: enter **Blocks.**
5. Under **Location**: Browse to where you are saving your C# projects.
6. Click the **Create** button to create the project.
7. This creates a blank MonoGame project. It should look like to this.



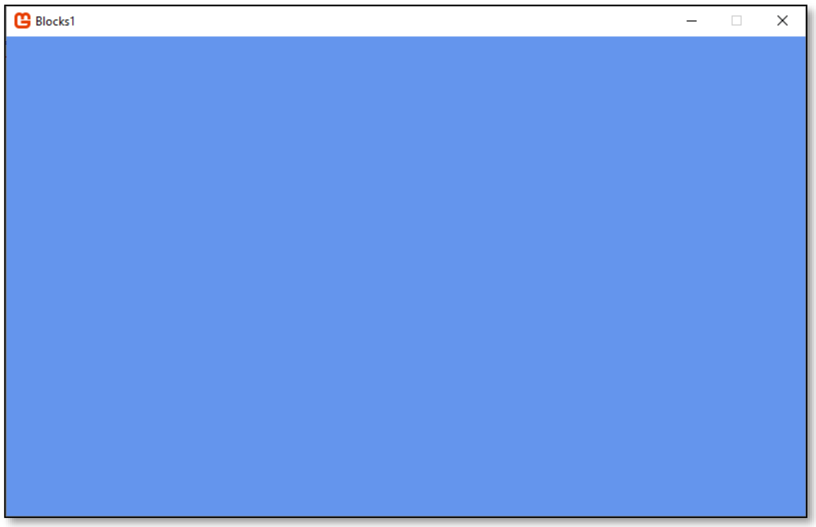
Solution Explorer

Search Solution Explorer (Ctrl+;)

```
Solution 'MonoGameBasics' (1 project)
  MonoGameBasics
    Properties
    References
    Content
    Game1.cs
    Icon.ico
    Program.cs
```

8. Press the **F5** button to run the application.

You should see a new window with the title **Blocks.** Press the **Esc** key (or click the **X** button in the upper right of the window), to close the game.



Congratulations! You have created one of the world's most boring video game!

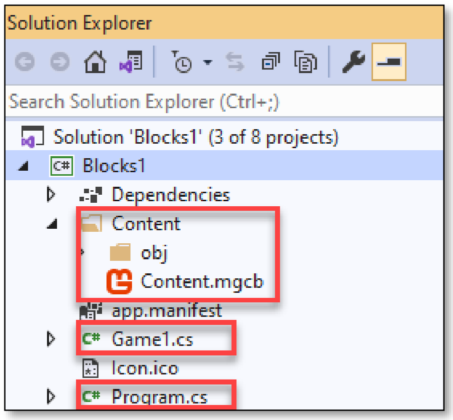
This is all we want to accomplish with this tutorial. We wanted to install and test the MonoGame development environment in Visual Studio.

This is the framework for the entire game. We are ready to get inputs from the keyboard or mouse, and draw on the screen in the next tutorial.

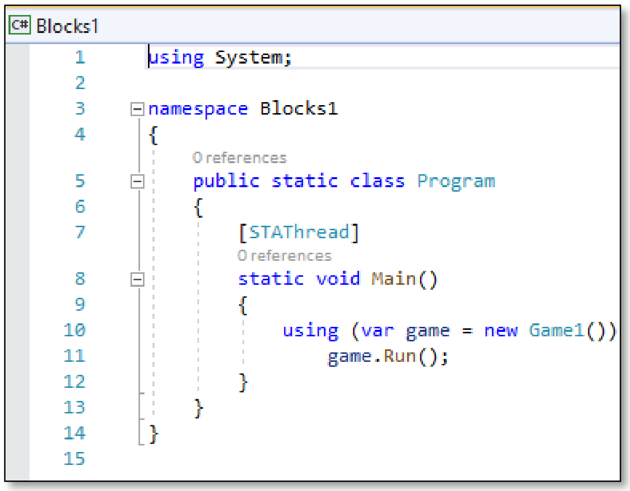
Basic Structure of a MonoGame Program AKA: The Game Loop

Before we go to the next tutorial, let's take a look at the structure of a MonoGame program.

In the Visual Studio Solution Explorer (upper right of the window), you will see a summary of the project that has been created. The three highlighted files are the ones we'll be looking at in this section:



The file "Program.cs" has the main entry point that Windows uses to launch our game. That's all it does. It gets loaded when our program runs, and calls the game Run method. We will never need to modify this file.



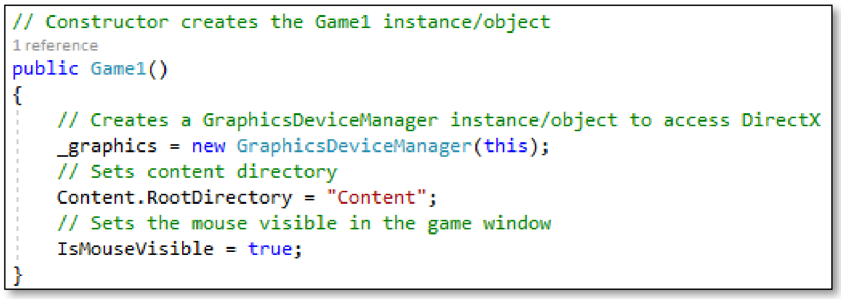
Blocks1

```
using System;

namespace Blocks1
{
    0 references
    public static class Program
    {
        [STAThread]
        0 references
        static void Main()
        {
            using (var game = new Game1())
                game.Run();
        }
    }
}

```

The **Content.mgcb** manages the Content Pipeline. The Content Pipeline modifies the images, sounds and other assets into the correct format for MonoGame. It also is used to help preload those assets into memory for fast access by the game.

The **Game1.cs** file contains the **Game1** class. This contains the main game loop for the project. Let's look at the methods in this class to see what they do. Everything we add to the project will be called from the methods in this class.

Game Constructor



```
// Constructor creates the Game1 instance/object
1 reference
public Game1()
{
    // Creates a GraphicsDeviceManager instance/object to access DirectX
    _graphics = new GraphicsDeviceManager(this);
    // Sets content directory
    Content.RootDirectory = "Content";
    // Sets the mouse visible in the game window
    IsMouseVisible = true;
}
```

This short method provides two critical things needed by our game.

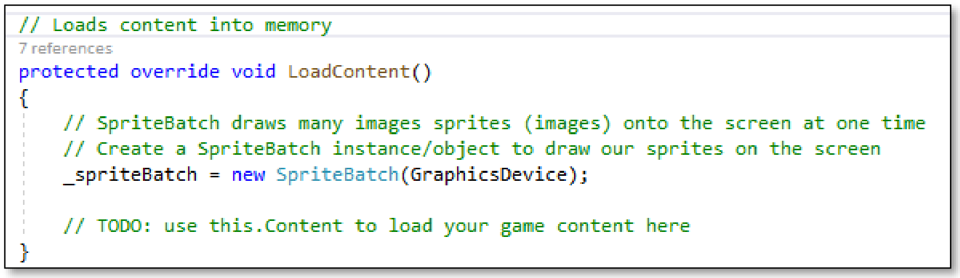
1. The first line creates a **GraphicsDeviceManager** instance/object that C# uses to interact with DirectX. This exposes or allows access to the graphics card for MonoGame.

2. The second line tells our program where to look for content (images, sounds, etc.) that will be used in our game. By default, this will be a folder called "Content" under the root folder of our project.

3. The 3rd line sets the window to display the mouse pointer.

The **Initialize** method is called once at startup. This isn't used for loading graphics and sounds, and it isn't needed for this game.

The **LoadContent** method caches or preloads all of our image and sounds assets into memory. They can be used quickly while the game is running without access the disk drive.

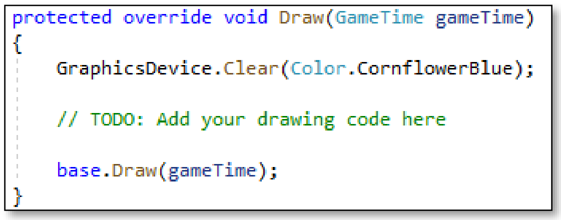


```
// Loads content into memory
7 references
protected override void LoadContent()
{
    // SpriteBatch draws many images sprites (images) onto the screen at one time
    // Create a SpriteBatch instance/object to draw our sprites on the screen
    _spriteBatch = new SpriteBatch(GraphicsDevice);

    // TODO: use this.Content to load your game content here
}
```

The **SpriteBatch** class allows MonoGame to draw a whole bunch of sprites (images) onto the screen at one time. This is more efficient than drawing them one at a time.

There is also an **UnloadContent** method. You can use this method to unload assets that you didn't load using the **ContentManager**. We will be using **ContentManager** to load our assets, we won't need to use **UnloadContent** in our program.

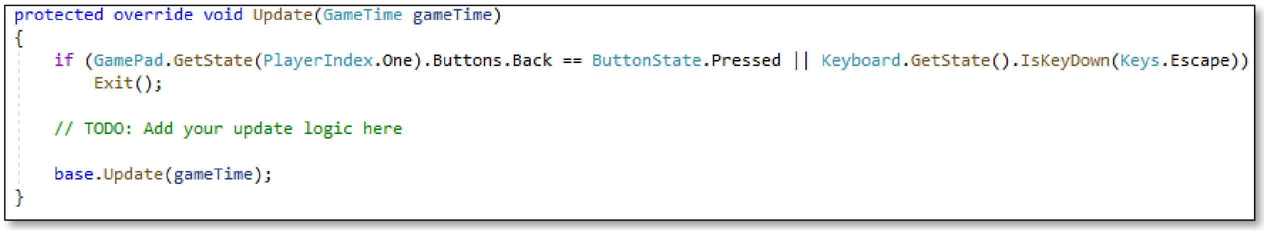
We'll use the remaining two methods the most: **Draw** and **Update**. MonoGame will continuously loop and call these two methods. We use the **Draw** method whenever we want to draw something on the screen.



```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.CornflowerBlue);

    // TODO: Add your drawing code here

    base.Draw(gameTime);
}
```

The **gameTime** property provides the elapsed time since the last update. This allows us to determine if the game is running at too slow a frame rate, so we can take action. It is available for more complex games. The **GraphicsDevice.Clear** statement is what causes the game to display the **CornflowerBlue** colored background in the game window.

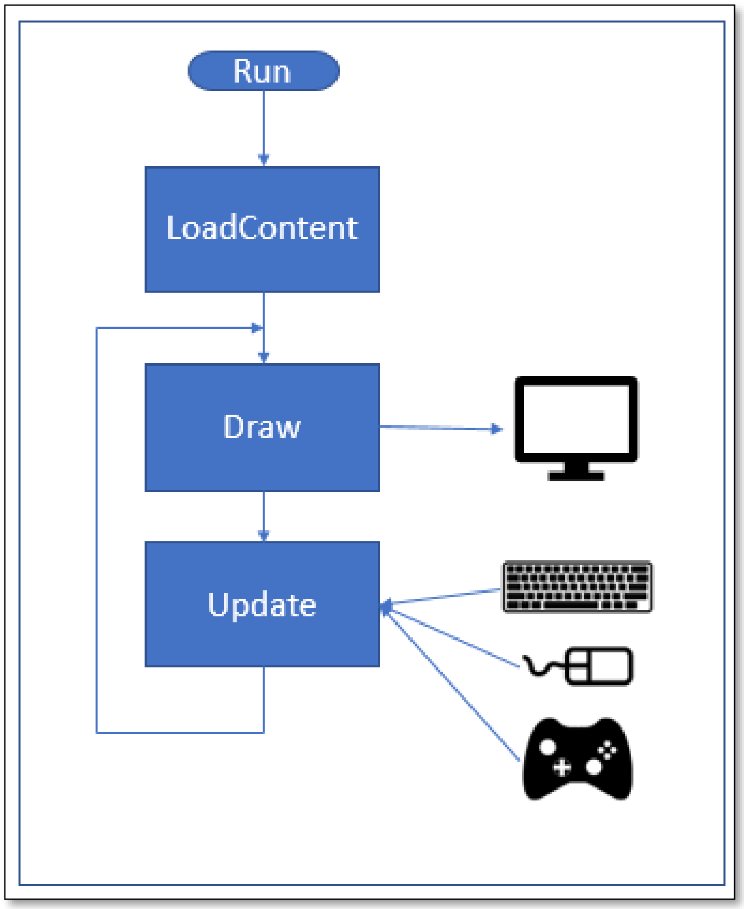
The **Update** method will allow us to get inputs from the mouse and keyboard (and a controller, if your game supports one).



```
protected override void Update(GameTime gameTime)
{
    if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed || Keyboard.GetState().IsKeyDown(Keys.Escape))
        Exit();

    // TODO: Add your update logic here

    base.Update(gameTime);
}
```

This **Update** method checks to see if the Back button on a game controller, or the "Escape" key is pressed on the keyboard. If it is, the program exits. That's why the program terminated when we pressed escape earlier.

Anything we want to display on the screen we must include in the **Draw** method. We'll get inputs and take actions whenever the **Update** method is called. The following diagram shows the flow of a MonoGame program.



Now that we have the basic framework under our belts, let's look at how MonoGame handles sound and graphics assets in the next tutorial.