CHAPTER 20

DungeonTrap Project Setup

Before I begin the chapter I want to list some general project notes to help make sure you have things setup correctly and can address the most common issues quickly. The notes are as follows:

Java:

1. The game project should be configured with a working directory set to './dist' in the project's 'Run' settings.

Java and C#:

1. The ENGINE_CONFIG_FILE field of the static main class should point to the game project's config file in the 'cfg' directory with a relative path from the project's working directory.
2. The game engine config file should have a NAME entry with a value that is the same as its associated project and that project's resource folder.
3. To turn off the gamepad 1 input add this line to your game engine config file:

```
<entry key="GAMEPAD_1_ON" val="false" type="bool"
from="GameSettings" />
```

This chapter will be somewhat similar to Chapter 16, but it's nuanced and specific to the DungeonTrap game project. In this chapter, I will walk you through setting up a new game development project for the DungeonTrap game jam in Java or C#. You can use this chapter as a reference for either version of the game engine and as a general guide for future game projects. We'll also take care of setting up the game's resources including images, sounds, and class config files used by the game.

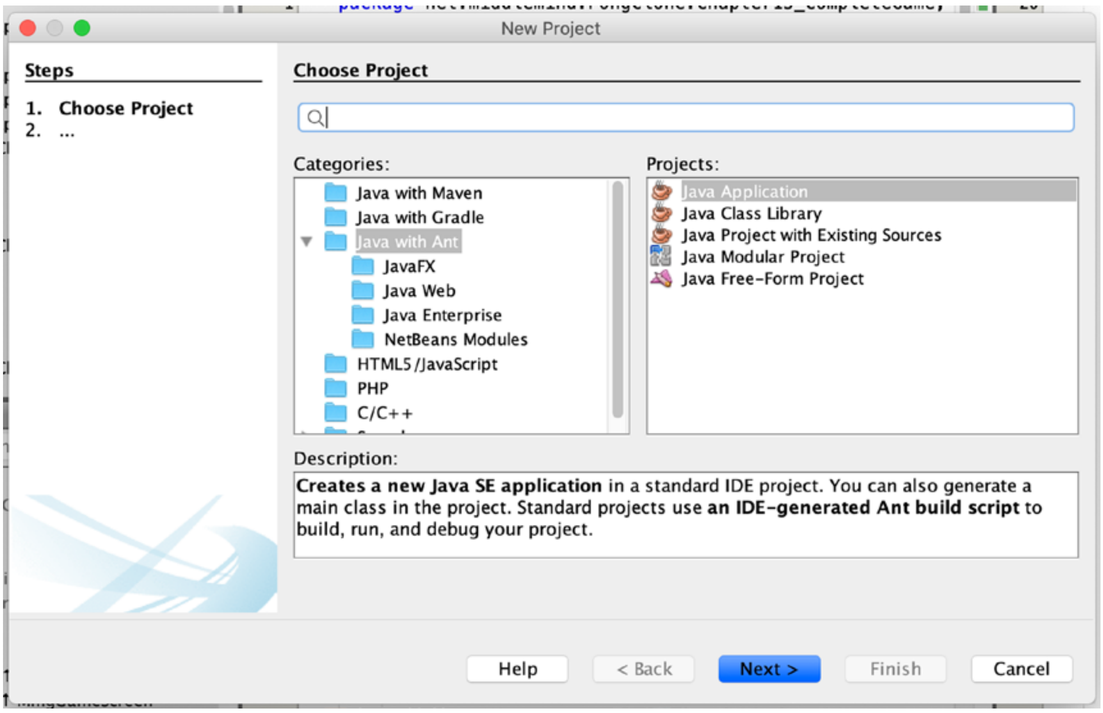
We're going to approach this game development project in steps. This chapter represents our first step. The goal is to have something functional at the end of a development step. As such, I've designed the development process so that you'll have a program that will compile and run at the end of this chapter. I should mention that this might not be the case in the real world, and that's perfectly normal. Don't make the mistake of thinking that video game development is always as cut and dried as the games we're building here. Let's get to it, shall we?

Setting Up a New Project: Java/NetBeans IDE

In this section, we'll review the steps necessary to create a brand-new Java game development project that uses the game engine API you've been learning throughout this text. First up, we'll outline the process for setting things up in the NetBeans IDE. Fire up the NetBeans IDE, and let's get to work! Make sure you have the MmgGameApiJava project opened in the IDE.

If you're working on a Mac and encounter an error regarding the library "libjnidispatch-440.jnilib" not being properly signed or a similar error for that matter, just take a moment to look up the error online for your particular version of OSX. You should be able to accept a security exception in the "System Properties" application under the "Security" section to prevent the error message from appearing when the IDE starts. You might also encounter this issue when running a game project in or outside of the IDE. The solution for this issue should be the same.

Once the IDE has loaded and you've opened the game engine project, select "File" from the menu and choose the "New Project ..." option. You'll need to create a new project in the same directory where you keep your NetBeans projects. Select the "Java with Ant" category from the "New Project" popup. From the "Projects" list, select the "Java Application" option and click "Next >."

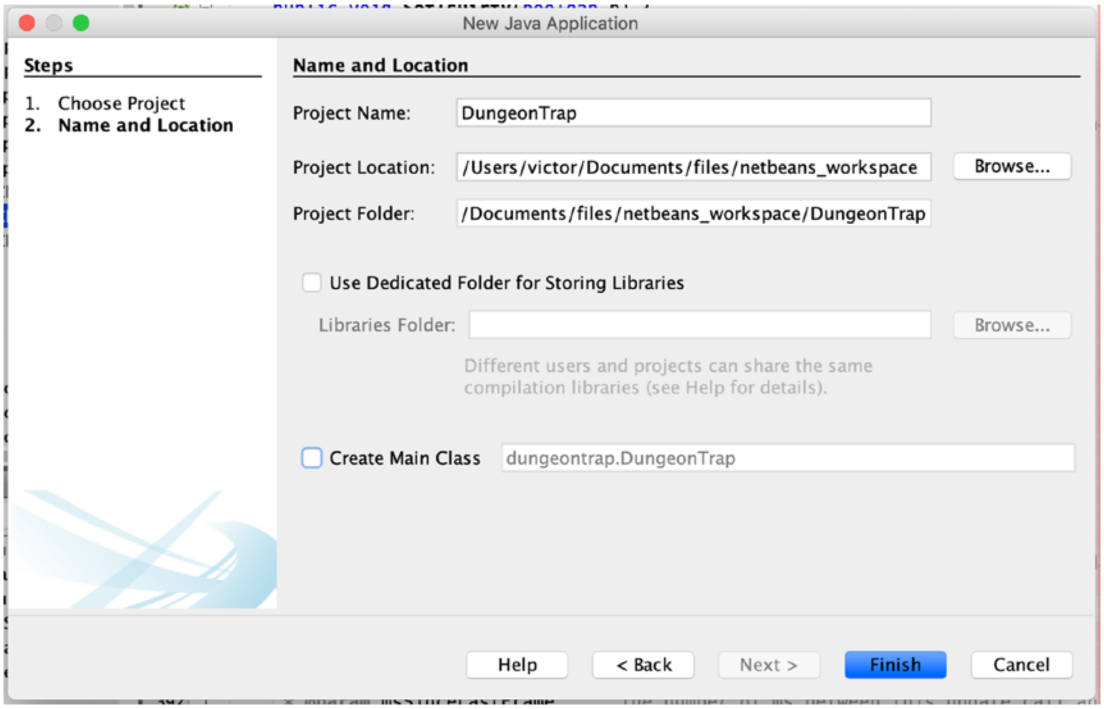


Creates a new Java SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

this figure will be printed in b/w

Enter in the name "DungeonTrap" in the "Project Name" field. Make sure that the "Project Location" and "Project Folder" fields have the correct values. Keep the "Use Dedicated Folder for Storing Libraries" and "Create Main Class" options unchecked and click the "Finish" button.

this figure will be printed in b/w



Project Name: DungeonTrap

Project Location: /Users/victor/Documents/files/netbeans_workspace

Project Folder: /Documents/files/netbeans_workspace/DungeonTrap

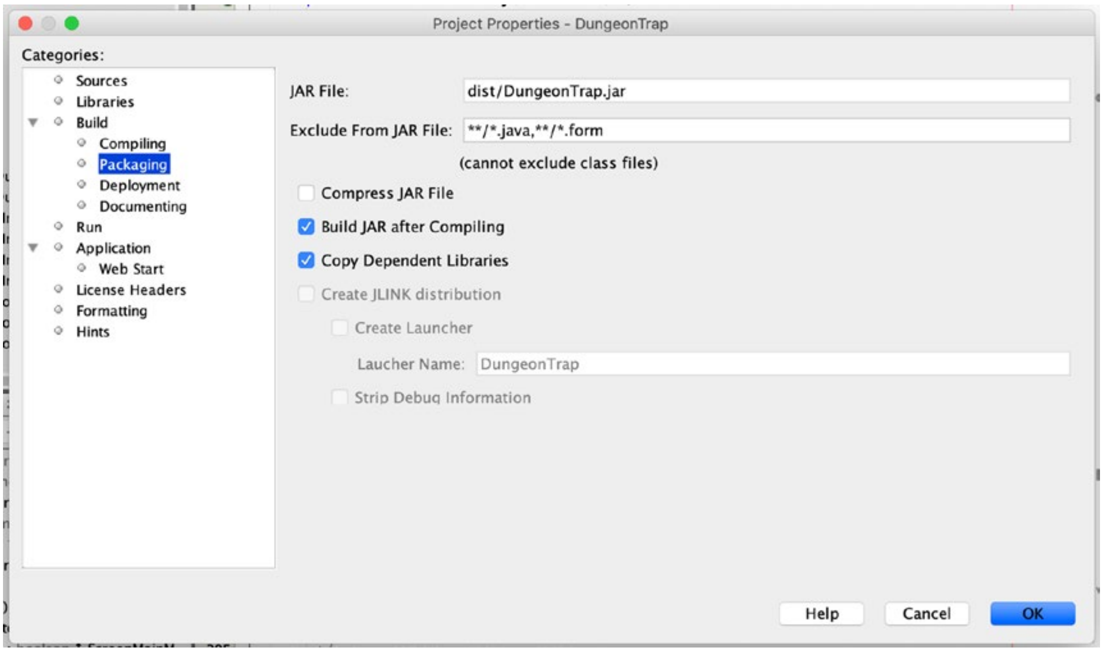
☐ Use Dedicated Folder for Storing Libraries

Libraries Folder:

Different users and projects can share the same compilation libraries (see Help for details).

☐ Create Main Class dungeontrap.DungeonTrap

You should now have a new empty project in the IDE’s project list, DungeonTrap. Next, we’ll look into configuring the project. Right-click your new project and select the “Properties” option from the context menu. You should see a “Project Properties” popup with a list of categories on the left-hand side of the screen. Select the “Packaging” option under the “Build” category.



JAR File: dist/DungeonTrap.jar

Exclude From JAR File: **/*.java,**/*.form

(cannot exclude class files)

☐ Compress JAR File

☑ Build JAR after Compiling

☑ Copy Dependent Libraries

☐ Create JLINK distribution

☐ Create Launcher

Laucher Name: DungeonTrap

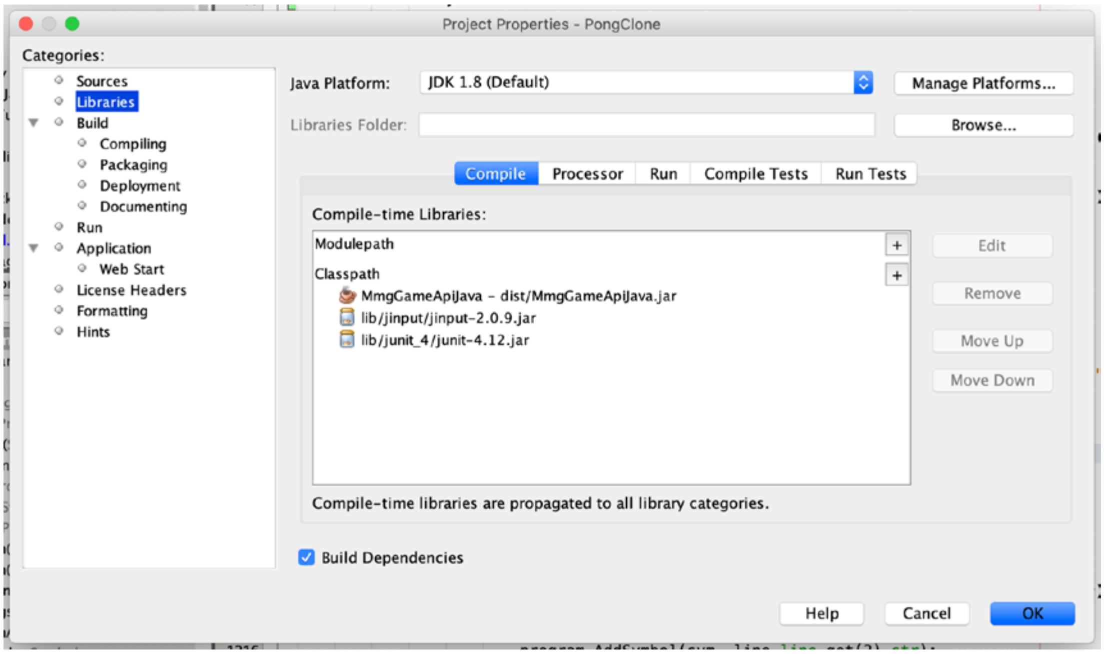
☐ Strip Debug Information

Make sure that your project's "JAR File" field is set to "dist/DungeonTrap.jar." Make sure your project has the working directory set to "./dist/" under the project's 'Run' settings. Also, make sure the "Build JAR after Compiling" and "Copy Dependent Libraries" options are checked as shown in the preceding. Now we need to set up our project resources. Again, you'll need to have the MmgGameApiJava project loaded in the IDE if it hasn't been already. Click the "Files" tab near the project listing. If you don't see it there, you can also access the "Files" view by clicking the "Window" main menu entry and selecting the "Files" option.

Open the "MmgGameApiJava" folder and locate a folder inside named "cfg." Open the "cfg" directory and locate the "asset_src" folder. In this folder, there should be a folder called "dungeon_trap_resources"; open it and find the folder inside named "cfg." Copy this folder and paste it into the root of the DungeonTrap project folder. Your new game project now has a full set of resources ready to use.

We're just about ready to start writing some code for our DungeonTrap game, but there are a few things we must attend to first. We need to get the project libraries configured properly. First, let's add a reference to the game engine API. Open up the DungeonTrap project's properties popup again. This time select the "Libraries" category from the left-hand side list of categories. Under the "Compile" tab, locate the "Classpath" section shown in the following.

this figure will be printed in b/w



Java Platform: JDK 1.8 (Default)

Libraries Folder:

Compile-time Libraries:

Modulepath

Classpath

MmgGameApiJava - dist/MmgGameApiJava.jar

lib/jinput/jinput-2.0.9.jar

lib/junit_4/junit-4.12.jar

Compile-time libraries are propagated to all library categories.

☑ Build Dependencies

Click the "+" button and select the "Add project" option. Choose the "MmgGameApiJava" project from the list, and you should now see the game engine project listed under the "Classpath" section. Close the Project Properties popup. Reopen the "Files" view. Find the game engine folder again and open it. You should see a folder named "lib" in the project's root directory. Copy the folder and go back one directory to the project listing. Find the "DungeonTrap" folder and paste the "lib" folder into the project's root directory. We have just two more libraries to register, and the library configuration is done.

Next, we'll need to open the DungeonTrap project's properties popup again. We need to set up a couple of project libraries. Select the "Libraries" category and find the "Classpath" section. Click the "+" button and select the "Add JAR/Folder" option. Navigate to the "DungeonTrap" project folder and open the "libs" directory. Open the "jinput" folder and select the file "jinput-2.0.9.jar"; click the "Choose" button and then click the "Ok" button.

Repeat this process except this time find and open the "junit_4" folder and select the "junit-4.12.jar" file. Complete the operation by clicking the "Choose" button and then clicking the "Ok" button to close the popup. That's it. The base project is set up and configured. We're ready to start working on the Java project, but first I'll cover setting up the C# version of this project.

Setting Up a New Project: C#/Visual Studio IDE

In this section, we'll review the steps necessary to create a brand-new game development project, in C#, that uses the game engine API. You'll need to open up a terminal window for this next part. Please refer to Chapter 1 for details on finding the terminal/console program for your particular operating system or search online. Once you're set up and ready with your terminal program, we'll need to run the following command to ensure that you have the necessary Monogame project templates installed. Again, refer to Chapter 1 for more information on how to set up the C#/Monogame environment:

```
dotnet new -i MonoGame.Templates.CSharp::3.8.0.1641
```

Once that's done, navigate to the directory where you keep your Visual Studio projects. If you are unfamiliar with your operating system's terminal program, take a moment to look up online how to change directories in your terminal program. You should have the MmgGameApiCs project in the same folder that the new game project will reside. Run the following command to create a new Monogame project for the DungeonTrap game:

```
cd [Place Your Visual Studio Directory Path Here]
dotnet new mgdesktopgl -o DungeonTrap
```

When the command is done running, you should have a new project folder in your Visual Studio project directory named "DungeonTrap." In the next few steps, we'll be working on getting the game project properly configured. First off, let's get a copy of the game's resource folder and add it to the project. Using your preferred folder navigator, locate and open the "MmgGameApiCs" folder in your Visual Studio project directory. Find the "cfg" folder inside and open it. You should see a folder named "asset_src"; open it and locate the folder named "dungeon_trap_resources." Open that directory and copy the "cfg" folder found inside.

Now, navigate to the "DungeonTrap" project folder you just created. Paste the "cfg" directory into the game project's root folder. All the game's resources are now ready to go! Next up, we have to configure the DungeonTrap project and add some libraries to it that will give us access to the game engine. For this next part, we must make sure that the game engine project has been built so we can reference the resulting .Net library.

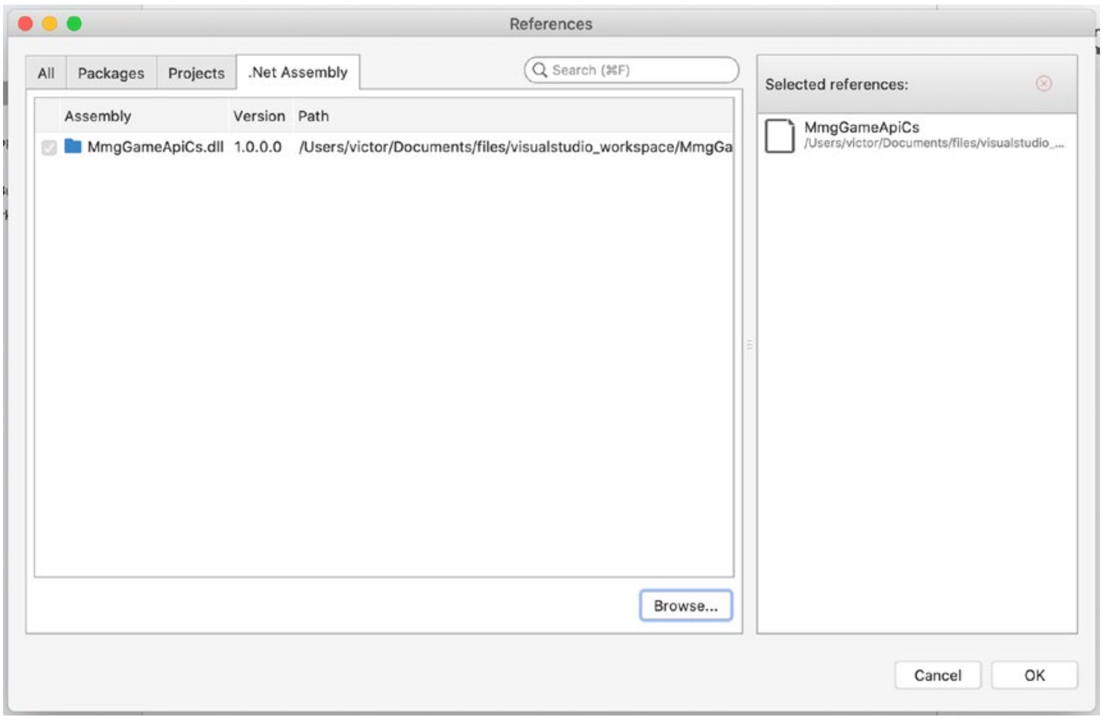
Open the MmgGameApiCs project, and once it's done loading, select the "Build" option from the IDE's menu and then choose "Rebuild" from the context menu. Allow the project to compile and make sure that you see the successful build results in the IDE's "Build Output" view. Now that we have a library to reference, close the MmgGameApiCs project and open the DungeonTrap project you just created.

When it's done loading, right-click the "Dependencies" folder in the "Solution" view and select "Add Reference …" from the context menu. From the "References" popup, click the ".Net Assembly" tab and click the "Browse …" button. Navigate to the "MmgGameApiCs" folder in your Visual Studio project directory. Find the following file and add it to your project's references. Click the "Ok" button to close the popup.

From the root of the game engine's project directory, the .Net library can be found at the following location:

```
./MmgGameApiCs/bin/Debug/netcoreapp3.1/MmgGameApiCs.dll
```

Your project "References" popup should look like the following screenshot.

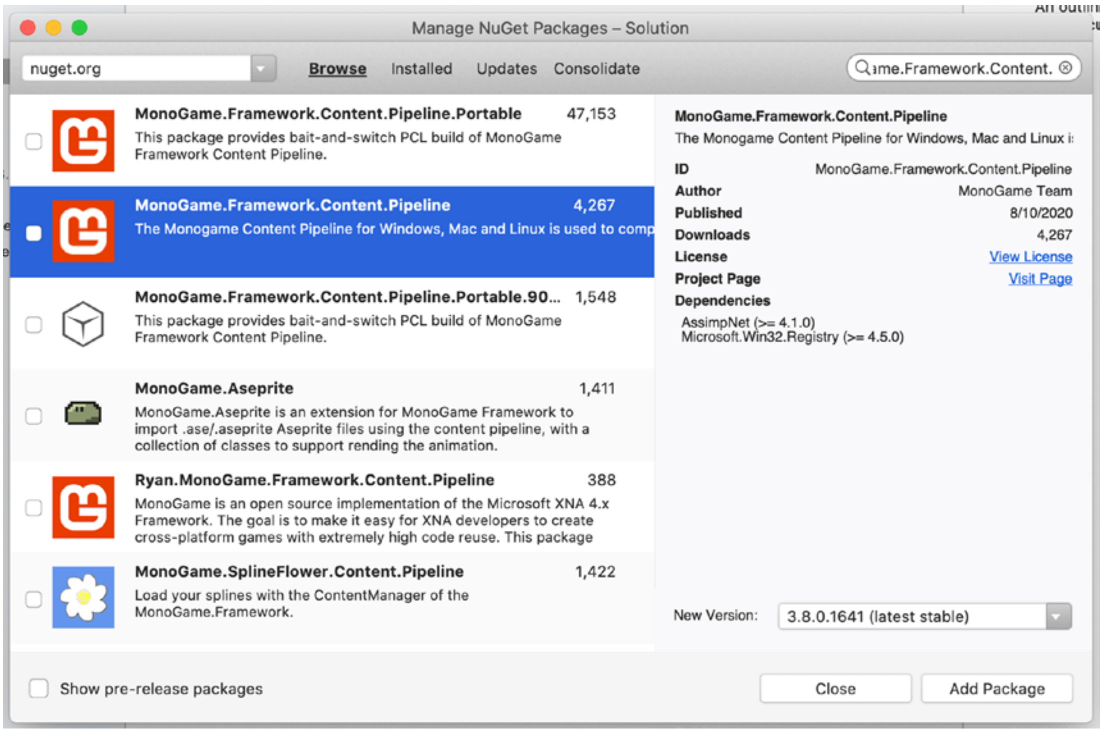


	Assembly	Version	Path
✓	MmgGameApiCs.dll	1.0.0.0	/Users/victor/Documents/files/visualstudio_workspace/MmgGa

MmgGameApiCs
/Users/victor/Documents/files/visualstudio_...

We have a few NuGet packages to add to our project before we can start coding. Select "Project" from the IDE menu and click the "Manage NuGet Packages …" option. When the "Manage NuGet Packages" popup is done loading, search for the package "MonoGame.Framework.Content.Pipeline" by using the search bar in the top right-hand corner of the popup.

Look for the package in the listing on the left-hand side of the popup and check the box next to the package listing. Next, search for the package named "MSTest.TestFramework." Check the box next to that package listing as well and click the "Add Package" button in the bottom right-hand corner of the popup. Accept any dialogs with licensing prompts, and the packages will be installed and ready to go in just a few minutes.



MonoGame.Framework.Content.Pipeline.Portable 47,153
This package provides bait-and-switch PCL build of MonoGame Framework Content Pipeline.

MonoGame.Framework.Content.Pipeline 4,267
The Monogame Content Pipeline for Windows, Mac and Linux is used to comp

MonoGame.Framework.Content.Pipeline.Portable.90... 1,548
This package provides bait-and-switch PCL build of MonoGame Framework Content Pipeline.

MonoGame.Aseprite 1,411
MonoGame.Aseprite is an extension for MonoGame Framework to import .ase/.aseprite Aseprite files using the content pipeline, with a collection of classes to support rending the animation.

Ryan.MonoGame.Framework.Content.Pipeline 388
MonoGame is an open source implementation of the Microsoft XNA 4.x Framework. The goal is to make it easy for XNA developers to create cross-platform games with extremely high code reuse. This package

MonoGame.SplineFlower.Content.Pipeline 1,422
Load your splines with the ContentManager of the MonoGame.Framework.

MonoGame.Framework.Content.Pipeline
The Monogame Content Pipeline for Windows, Mac and Linux i

ID	MonoGame.Framework.Content.Pipeline
Author	MonoGame Team
Published	8/10/2020
Downloads	4,267
License	View License
Project Page	Visit Page

Dependencies
AssimpNet (>= 4.1.0)
Microsoft.Win32.Registry (>= 4.5.0)

New Version: 3.8.0.1641 (latest stable)

this figure will be printed in b/w

There's one last thing for us to do before we can start coding. We have to add the Monogame content resource file to our project. The content file contains font data for font sizes 1–50, used by the C# version of the game engine. Remember that the C# version of the game engine uses this font data to emulate the more fluid font functionality of the Java version. Using the file system navigator of your choice, locate and open the "MmgGameApiCs" project's folder.

Copy the folder inside named “Content” and paste it into the root of the “DungeonTrap” project’s folder. Overwrite the existing “Content” folder if one exists. Open the project in Visual Studio when the folder is done copying. I want to double-check that the new content compiles correctly. Once the DungeonTrap project is done loading, expand the “Content” folder listed in the Solution view. Right-click the file named “Content.mgcb” and select the “Open With …” option.

Click the “mgcb-editor” context menu option. This will open the content database in a special editing utility. Find the “Rebuild” button, third from the right in the editor’s button bar. Click it and verify that the content compiles correctly. The project is configured and ready. We’ve got some coding to do!

Setting Up Core Code: C# and Java

Now that we’ve got our game project set up and configured, it’s time to start coding. In this section, we’re going to set up the game project’s core code. Previously I outlined some of the major differences between the C# and Java implementations, but I’m going to move away from that level of detail during this game build as much as possible.

Both the Visual Studio and NetBeans IDEs will help you get your `using/import` statements corrected when working with the game’s classes. Also, you can always reference the completed chapter project included in the game engine project for both C# and Java. Aside from these nuances, the only differences in code are due to fundamental differences in the two programming languages.

That being said, I want to discuss the format of the game build process for the DungeonTrap game. I’ll employ the same loose review style that was used during the PongClone game jam. However, once we get into the game’s key classes, I’ll switch to the more structured code review style used in Parts 1 and 2 of this text. As I mentioned earlier, in this section, we’ll be setting up the DungeonTrap game’s core code. I’ve listed the files we’ll be working with, for both C# and Java versions of the game, in the following:

Java version

- DungeonTrap.java
- MainFrame.java
- GamePanel.java

C# version

- DungeonTrap.cs
- GamePanel.cs

Before we start adding classes to our project, let's create a place to keep our files. In Java, we'll create a new package to hold our work. For the C# version of the project, we'll emulate the Java version's package with a series of project folders and by using a namespace that matches the Java package. In the NetBeans IDE, right-click the DungeonTrap project and select "New" from the context menu and then select the "Java Package …" option. Set the name of the package to "game.jam.DungeonTrap." In Visual Studio, we're going to create the following folder structure in the DungeonTrap project:

```
/src/game/jam/DungeonTrap
```

You can accomplish this by right-clicking the project and selecting the "Add" context menu option and then clicking the "New Folder" option. Repeat this process until you've created the structure listed in the preceding. Make sure to set the namespace for any new C# class files to "game.jam.DungeonTrap." Let's redirect our attention to the Java and C# versions listed in the preceding. Note the slight difference in the files listed. Do you know why this is? From the review of the MmgCore API in Part 2 of this text, the C# implementation doesn't need a `MainFrame` class due to the underlying framework's implementation.

The first class that we'll add to the project is the static main class. It just so happens that we have a static main class all set up and ready to use. Copy the `MmgApiGame` class from the game engine project's MmgCore API. Paste the file into the DungeonTrap project and rename the file and class so they are both named "DungeonTrap." Next you'll need to change the value of the ENGINE_CONFIG_FILE field to "../cfg/engine_config_mmg_dungeon_trap.xml". Open up the "engine_config_mmg_dungeon_trap.xml" file and set the NAME entry's value to "DungeonTrap". Now the application configuration and resources are aligned. There is also one reference to the `MmgGameApi` class in the file. Search for it and replace it with the class type `DungeonTrap`.

You can perform the refactoring operation automatically in the NetBeans IDE if you have both projects open. If you do, when you copy and paste the class into the new project, you can choose the "Refactor Paste" option, which will take care of renaming the class for you. In the C# version of the DungeonTrap project, you'll need to delete the default files, `Game1.cs` and `Program.cs`, from the project before pasting a copy of the `MmgApiGame.cs` file into the project.

Make sure you paste the file into the `/src/game/jam/DungeonTrap` folder. You'll have to refactor the class as noted in the preceding. Right-click the newly pasted file and select the "Rename …" option. Rename the file and the class to "DungeonTrap." Next you'll need to change the value of the ENGINE_CONFIG_FILE field to "../cfg/engine_config_mmg_dungeon_trap.xml". Open up the "engine_config_mmg_dungeon_trap.xml" file and set the NAME entries value to "DungeonTrap." Now the application configuration and resources are aligned. Again, make sure the project is setup to use "./dist/" as the working directory. Next up, we'll make an adjustment to the Java and C# projects. If you're following along in Java, open up the NetBeans IDE and the DungeonTrap project if they're not open already.

Right-click the project and open the properties popup. Select the "Run" option from the list of categories on the left-hand side of the popup. Find the "Main Class" text field and click the "Browse …" button. Select the `DungeonTrap` class as the default main class. The adjustment that is necessary for the C# version of the DungeonTrap project is to open the `DungeonTrap.cs` class and change the name of the `AltMain` method to `Main`.

Now that we have a solid foundation to work from, we can add a few more classes to move the project along. These classes are actually defined in the MmgCore API, but we want to extend those classes and define them locally so that we can have full control over the game's code. The next class we'll work on is the `MainFrame` class. Again, this only applies to the Java version of the game. The C# version of the project doesn't have an equivalent `MainFrame` class.

We want to extend the MmgCore API's `MainFrame` class and preserve its functionality. Create a new empty class in the DungeonTrap project. It should reside in the same package as the static main class. The class is concise, so I'll list it here.

Listing 20-1. MainFrame Java Setting Up Core Code 1

```
public class MainFrame extends net.middlemind.MmgGameApiJava.MmgCore.MainFrame {

    public MainFrame(int WinWidth, int WinHeight, int PanWidth, int PanHeight, int GameWidth, int GameHeight) {
        super(WinWidth, WinHeight, PanWidth, PanHeight, GameWidth, GameHeight);
    }

    public MainFrame(int WinWidth, int WinHeight) {
        super(WinWidth, WinHeight);
    }
}
```

Notice that I refer to the MmgCore API's `MainFrame` class explicitly. If I used an `import` statement, there would be two classes named "MainFrame" in the current scope, and that can be confusing. We want to avoid this. By not importing the library, it forces the class reference to be explicit. You can clearly see where I'm using the MmgCore API's `MainFrame` class and where I'm not. Notice that we define the same class constructors that exist in the API's `MainFrame` class and we use the `super` keyword to call the super class' constructor.

What we've done here is bring the functionality of the MmgCore API's `MainFrame` class into the current project's package through class extension as shown in the preceding. That takes care of the second main step we needed to complete. The next class for us to work on is the `GamePanel` class. We're going to be using a similar technique of extending base class functionality, or super class functionality if you're following along in Java. By extending and customizing the MmgCore API's `GamePanel` class, we can leverage all the existing code and add new, project-specific functionality at the same time.

Taking a look at the game's general specifications, listed in Chapter 15, I can see that I need to support the same four game screens that the PongClone game used. Of course, some of them will be customized for this game. We know that we'll need a splash screen, loading screen, and main menu screen. We also know that the splash screen and loading screen use generic events, MmgCore API, to indicate when the game state needs to change.

At this point in the project, after reviewing this game's general specifications, we know that we'll need to customize the game panel class' constructor along with the `SwitchGameState` and `HandleGenericEvent` methods. Let's start a class outline. I've listed the code in the following.

Listing 20-2. GamePanel Java Setting Up Core Code 2

```
public class GamePanel extends net.middlemind.MmgGameApiJava.MmgCore.
GamePanel {

public GamePanel(MainFrame Mf, int WinWidth, int WinHeight, int X, int Y,
int GameWidth, int GameHeight) { ... }

@Override
         public void SwitchGameState(GameStates g) { ... }
@Override
          public void HandleGenericEvent(GenericEventMessage obj) { ... }
}
```

We'll use the same process for the C# version of the game with some slight, language-based differences.

Listing 20-3. GamePanel C# Setting Up Core Code 3

```
public class GamePanel : net.middlemind.MmgGameApiJava.MmgCore.GamePanel {

public GamePanel(int WinWidth, int WinHeight, int X, int Y, int GameWidth,
int GameHeight) { ... }

public override void SwitchGameState(GameStates g) { ... }

public override void HandleGenericEvent(GenericEventMessage obj) { ... }
}
```

Notice how similar the two class outlines are. The only differences we've seen so far are based on differences between the two programming languages, Java and C#. Indeed, the code used to complete the outlined constructor and class methods will also be very similar. Let's fill in the class constructor next. As usual, I'll work with the Java version first.

Listing 20-4. GamePanel Java Setting Up Core Code 4

```
01 public GamePanel(MainFrame Mf, int WinWidth, int WinHeight, int X,
   int Y, int GameWidth, int GameHeight) {
02     super(Mf, WinWidth, WinHeight, X, Y, GameWidth, GameHeight);
03     screenSplash.SetGenericEventHandler(this);
04     screenLoading.SetGenericEventHandler(this);
05 }
```

Because we're using the `GamePanel` class' built-in functionality, we have access to the splash screen and loading screen class fields. I want to make sure the classes' event handlers are wired up correctly. To do so, I explicitly set them on the preceding lines 3 and 4. Let's take a look at the C# version of this method.

Listing 20-5. GamePanel C# Setting Up Core Code 5

```
public GamePanel(int WinWidth, int WinHeight, int X, int Y, int GameWidth, int GameHeight) :
    base(WinWidth, WinHeight, X, Y, GameWidth, GameHeight) {
    screenSplash.SetGenericEventHandler(this);
    screenLoading.SetGenericEventHandler(this);
}
```

As you can see, the constructor implementations are nearly identical. Keep in mind that this is where we'll initialize our game screens. We didn't need to create any new class fields because the `screenSplash` and `screenLoading` fields are available by default. This will change as we add new game screens that are outside of the default functionality. Next up, we'll tackle the `SwitchGameState` method. I'm going to follow the default implementation exactly as you'll see in the following. The code will be almost identical for the Java and C# versions of the game, so we'll cover both right now.

Listing 20-6. GamePanel Setting Up Core Code 6

```
//Java Method Signature
@Override
public void SwitchGameState(GameStates g) { ... }

//C# Method Signature
public override void SwitchGameState(GameStates g) { ... }

    if (gameState != prevGameState) {
        prevGameState = gameState;
    }

    if (g != gameState) {
        gameState = g;
    } else {
        return;
    }

    if (prevGameState == GameStates.BLANK) {
        MmgHelper.wr("Hiding BLANK screen.");

    } else if (prevGameState == GameStates.SPLASH) {
        MmgHelper.wr("Hiding SPLASH screen.");
        screenSplash.Pause();
        screenSplash.SetIsVisible(false);
        screenSplash.UnloadResources();

    } else if (prevGameState == GameStates.LOADING) {
        MmgHelper.wr("Hiding LOADING screen.");
        screenLoading.Pause();
        screenLoading.SetIsVisible(false);
        screenLoading.UnloadResources();

    } else if (prevGameState == GameStates.MAIN_MENU) {
        MmgHelper.wr("Hiding MAIN_MENU screen.");
        screenMainMenu.Pause();
        screenMainMenu.SetIsVisible(false);
        screenMainMenu.UnloadResources();

    }

    if (gameState == GameStates.BLANK) {
        MmgHelper.wr("Showing BLANK screen.");

    } else if (gameState == GameStates.SPLASH) {
        MmgHelper.wr("Showing SPLASH screen.");
        screenSplash.LoadResources();
        screenSplash.UnPause();
        screenSplash.SetIsVisible(true);
        screenSplash.StartDisplay();
        currentScreen = screenSplash;

    } else if (gameState == GameStates.LOADING) {
        MmgHelper.wr("Showing LOADING screen.");
        screenLoading.LoadResources();
        screenLoading.UnPause();
        screenLoading.SetIsVisible(true);
        screenLoading.StartDatLoad();
        currentScreen = screenLoading;

    } else if (gameState == GameStates.MAIN_MENU) {
        MmgHelper.wr("Showing MAIN_MENU screen.");
        screenMainMenu.LoadResources();
        screenMainMenu.UnPause();
        screenMainMenu.SetIsVisible(true);
        currentScreen = screenMainMenu;

    }
}
```

This class method should look familiar from our review of the MmgCore API. We have one more method to work on; then we'll compile and test our project. Let's take a look at the `HandleGenericEvent` method. This code will be the same in the Java and C# versions of the game. The `HandleGenericEvent` method is designed to process game screen events from the splash and loading screens. We're going to start with the same code that exists in the MmgCore API's `GamePanel` class. As we progress with the development of the game, we'll add any new code needed to the methods and constructor we've just outlined. Let's jump into some code!

Listing 20-7. GamePanel Setting Up Core Code 7

```
//Java method signature
@Override
public void HandleGenericEvent(GenericEventMessage obj) { ... }
```

```
//C# method signature
public override void HandleGenericEvent(GenericEventMessage obj) { ... }
```

Again, the only difference here is driven by the underlying programming language. Let's take a look at the method's contents in the following.

Listing 20-8. GamePanel Setting Up Core Code 8

```
if (obj != null) {
    MmgHelper.wr("TestSpace.HandleGenericEvent " + obj.GetGameState());
    if (obj.GetGameState() == GameStates.LOADING) {
        if (obj.GetId() == ScreenLoading.EVENT_LOAD_COMPLETE) {
            //Final loading steps
            DatExternalStrings.LOAD_EXT_STRINGS();
            SwitchGameState(GameStates.MAIN_MENU);
        }
    } else if (obj.GetGameState() == GameStates.SPLASH) {
        if (obj.GetId() == ScreenSplash.EVENT_DISPLAY_COMPLETE) {
            SwitchGameState(GameStates.LOADING);
        }
    }
}
```

Add the code shown in the preceding to your GamePanel class and then save and build your project. Make sure you address any typos or bugs. We're going to run the project and see what we've got. To run the Java version of the project, simply right-click the game project's static main class and select "Run." You should see a window popup with your game running in just a few seconds.

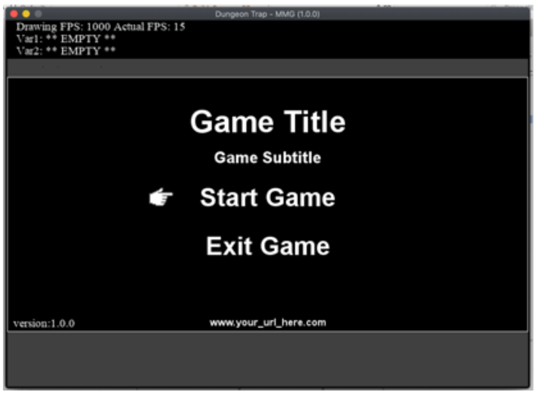
If you're following along in C#, you're going to want to run the game from a terminal window. Open up your favorite terminal and navigate to your game's root directory. Make sure you build your game in Visual Studio by clicking "Build" and selecting "Rebuild All" from the IDE's main menu. You should then see the following folder in the project directory:

```
/bin/Debug/netcoreapp3.1/DungeonTrap.dll
```

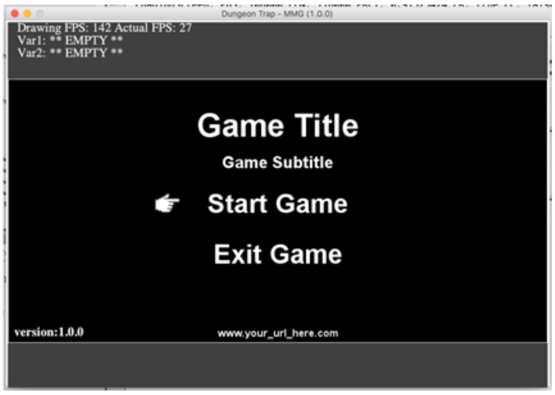
Run the following command to launch your game:

```
cd /bin/Debug/netcoreapp3.1/
dotnet DungeonTrap.dll
```

You should see images similar to those shown in the following depending on your OS and programming language choices.



```
Drawing FPS: 1000 Actual FPS: 15
Var1: ** EMPTY **
Var2: ** EMPTY **
```



```
Drawing FPS: 142 Actual FPS: 27
Var1: ** EMPTY **
Var2: ** EMPTY **
```

this figure will be printed in b/w

We're off to a great start. Three small classes, two if you're following along in C#, and our project now supports a splash screen, a loading screen, and a default main menu screen. Not too bad! Keep in mind this is a project in development; so if you try hard enough, or at all maybe, you'll probably find a way to get it to crash. No worries, we're only at the start of our game project.