Programming with the Fraction Game Engine.

A user-friendly guide to programming games with the Fraction Game Engine.

Oliver Charles

Version 1

This guide was last updated Monday, 07 March 2005

Table of Contents

SECTION 1

Getting Started

- *Setting up a project in Visual Studio .NET 2003*
- *State machines*
- *Helper classes and management systems.*

1 Getting Started

The Fraction Game Engine is a powerful tool for anyone, but it's pretty crucial to understand how to use it. In this section I will be introducing how to set up a project to use Fraction, and some basic design patterns that can be adopted (and should be) for games. I'll finally introduce the management and helper classes that Fraction includes to make development even quicker and easier.

1.1 Setting up a project in Visual Studio .NET 2003

I will explain here how to setup Fraction with Visual Studio .NET 2003. Fraction can also be used with other IDE's, but I feel that Visual Studio is best and most people will find themselves using it sooner or later.

The first thing we need to do is create a new project. To do this we just need to undertake the following steps:

1. *Load Visual Studio .NET.*

   You can either navigate to where you installed Visual Studio or just choose it from the start menu.

2. *Create a new project.*

   The quickest way to do this is choose New | Project from the file menu, or you use the keyboard shortcut, Ctrl + Shift +N. Choose the language you wish to develop in and choose 'Windows Application' as the template.

3. *Create references.*

   Now we need to setup our references. A reference is like linking in C++. You choose the DLL (or EXE) that you wish to reference, and you can access all the classes and methods without having the code. We'll add a new reference to the Fraction Game Engine.

   To create the reference, click Project | Add Reference on the menu. Click 'Browse' and then navigate to where ever you extracted the Fraction Game Engine to. You want the file called 'Fraction 1.1.dll' that is commonly in the 'Binaries' directory.

4. *Create the launcher file.*

   The launcher file will contain the class that launches the game. Launcher file probably isn't the proper name, but it makes sense. Anyway, firstly delete the 'Form1' file if it exists, we don't need this file. Create a new class for the project by going to "Project | Add Class…" Give it a suitable name, such as GameLauncher.

Now we can start coding! The first thing we need in our code is to tell it that we need to access Fraction's classes. This changes depending on the language but you need one of the following:

Visual Basic .NET

```
Imports Fraction.Core
```

C#

```
using Fraction.Core;
```

5. *Setup the game loop.*

   Now we have everything we need to start coding, so lets create a window to render with! We need an engine, and… that's it! So, add the following to your GameLauncher class:

   Visual Basic .NET

```
Dim GameEngine As New Engine(800, 600, False)
```

   C#

```
Engine GameEngine = new Engine(800, 600, false);
```

   Now, I like that we have our engine created, we need an entry point and a loop for our program. Below is some basic code.

   Visual Basic .NET

```
Private Function Go()
	While (GameEngine.Visible)
		GameEngine.Flip()
		System.Windows.Forms.Application.DoEvents()
	End While
End Function

Public Shared Sub Main()
	Dim MyApp As New GameLauncher
	MyApp.Go()
End Sub
```

   C#

```
void Go()
{
	while(GameEngine.Visible)
	{
		GameEngine.Flip();
		System.Windows.Forms.Application.DoEvents();
	}
}

public static void Main()
{
	GameLauncher MyApp = new GameLauncher();
	MyApp.Go();
}
```

That's it! You've made your first program using the Fraction Game Engine. That's really how simple creating applications are with Fraction. Now that you know the basics of setting up a project with Fraction, you can start using Fraction and get making a game!

1.2 State Machines

1.2.1 An introduction to state machines

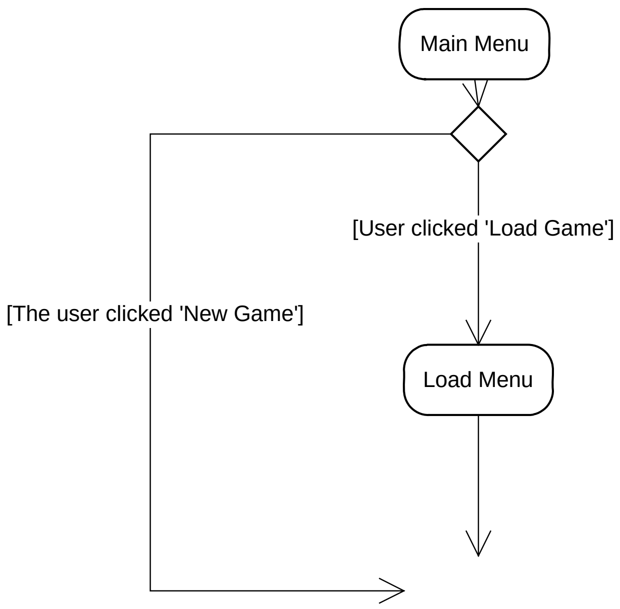
State machines are quite a complex topic, and often missed by new game programmers. Fraction includes a state machine and the sooner you understand how to use and harness it, the better! I'll start with an introduction to state machines.

State machines are a design pattern that allows the program to be split up into different parts, and the computer can handle it automatically. You split the game into different states and depending on what is happening, you change to the appropriate state when necessary. First, lets understand what a state really is in the context of a game.

Lets pretend we're making a pretty basic game – Breakout for instance. The user fires up your game and gets a nice main menu screen. This is a *state* that the game is in. They have the options: new game, load game and quit. If they click new game then the current *state* that the game is in is changed to the 'game state.' The game state is set-up to load the first level, with a default amount of lives and a default score of 0.

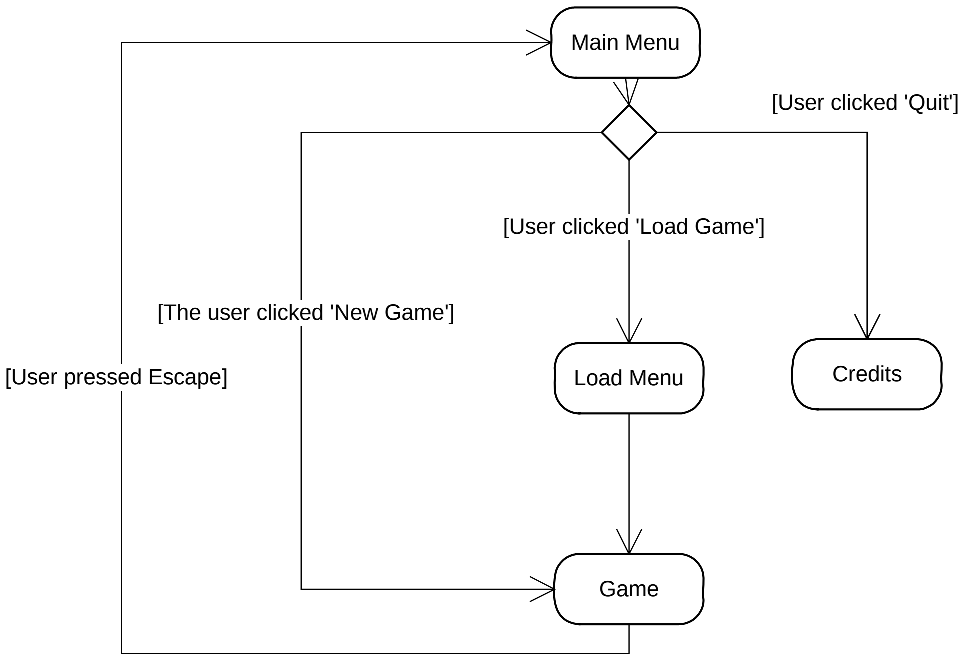
But if they clicked load game on the main menu, then the game state changes to the 'load game' state. This is a state that shows a list of saved games. When the user clicks a game to load then the current state is changed to 'game state.' This time it is set-up to load whatever level the save game is on, along with the score and the amount of lives they had.

Hopefully, that above real-life example made some sense! It's also very good to visualise state patterns. Below is a state diagram that explains the above concept.



So what does this mean to the programmer? Well, for a start you've got 3 separate states. None of them need to know about the others, which means you can easily change them, without having to update anything else. It makes your code a lot more readable and has the potential to allow other developers to easily join your development team.

State machines, as said above also allow for extensibility. It doesn't take much to add a new state in. Lets say that you wanted to allow people to return to the menu screen when you press escape, and lets also add the "credits" state that is called when the user clicks quit.



See how we have maintained existing operability but now added that in? It's really that simple, and I feel one of the prime reasons why state machines are so useful.

1.2.2 Setting up a state manager

All of Fraction's state machine functionality is included in the Fraction.StateManagement namespace. You'll need to add the StateManagement namespace to the file you are working with by adding "using Fraction.StateManagement;" or "Imports Fraction.StateManagement" to the top of your file. Then you can begin working with states.

There are 2 things you need to do use states with your game The first step is to create an instance of the StateManager class, and the other is to create classes that inherit the State class (in Fraction.StateManagement).

If you are using the StateManager to run your entire game for you, you need to add a function to the Update event in the StateManager. Below are examples of using the StateManager to run a game. This is should be put in your "GameLauncher" class.

Visual Basic .NET

```
Dim GameEngine As New Engine(800, 600, False)        ' This is just the main
                                                     engine
```

```
WithEvents GameManager As New StateManager()

Sub Update(Sender As Object, E As EventArgs) Handles GameManager.Update
       GameEngine.Flip()
       System.Windows.Forms.Application.DoEvents()
End Sub
```

C#

```
Engine GameEngine = new Engine(800, 600, false);      // Just the engine

StateManager GameManager = new StateManager();

public GameLauncher()      // A constructor on the GameLauncher class.
{
       GameManager.Update += new EventHandler(Update);
}

void Update(object sender, EventArgs e)
{
       GameEngine.Flip();
       System.Windows.Forms.Application.DoEvents();
}
```

Ok, so we our state manager created, and its ready to be put to use! The first thing is to tell your Go function to use the state manager. All you need to do is to delete everything in the Go function and replace it with GameManager.Execute(); So that's it? Well… not quite. We still need to create some states for our game! Fraction has classes to help you with this, so just inherit from the state class and away you go!

1.2.3 Creating states

As said above, to create new states for your game you just need to inherit from the State class. Below are the bases needed to create a state.

```
public class BaseState : State
{
       public BaseState(StateManager Parent) : base(Parent) { }

       public override void OnEnter() { }
       public override void OnExit() { }

       public override void Render() { }
       public override void Update() { }
}
```

That is the base needed to create any state. So, what does this all mean? The first 'function' in there is actually called a constructor. This is called the moment the state is created and you should leave this blank (you'll understand why later). Now we have the OnEnter and OnExit functions. The former is called when you actually activate the state and the second is called when the state is left. It's a good idea to search and create