# Quick and Dirty Dev Guide for Sharplike 0.4

by Ed Ropple (ed.ropple@largerussiangames.com)

## Introduction

Since the Sharplike team just pushed Sharplike 0.4.0, I figured it was high time to actually dogfood our code and make a real game with it. I’m writing this guide as I go along to make the game, and I’ll include any helpful asides or other things I noticed during development to better enable people to use our library to realize their own games.

I should note that this game is using Sharplike *trunk*, not Sharplike 0.4.0. Sharplike trunk is our current code snapshot, with all the code that Sean, Alex, or I (the current Sharplike team as of this writing) have committed to the Sharplike repository. You can get Sharplike trunk by opening up your SVN client (TortoiseSVN[[1]](#footnote-1) for Windows is my recommendation) and checking out the following directory:

[**http://opensource.edropple.com/svn/Sharplike/trunk**](http://opensource.edropple.com/svn/Sharplike/trunk)

We try to keep Sharplike trunk in a buildable state—I am far and away the record-holder for “most broken builds”—but it may occasionally not be buildable, so keep an eye out for our (translation: my) screw-ups.

This dev guide is written almost from a dev diary point of view, and as such it’s tailored toward my own personal workflow. This means Visual Studio 2010 (most of which should be transparently doable in VS2005 or VS2008). I’m also expecting users to have at least some C# experience, so I probably won’t stop to explain C# or .NET concepts. Feel free to post in our forums if you have any questions regarding what you read here.

## Game Concept

This is going to be packed with Sharplike as a demo project, so it needs to be fairly simple and illustrative of our system’s capabilities. As it’s mostly an example rather than an intended playable game, I’ve decided that I’m going to be building a really simple Hack-type game, without a whole lot of pre-planning as far as game mechanics go. I’m calling it **Sharphack**, and the goal will be to go through ten levels of mazes, recover the Big Shiny Gem, and escape with it without ending up all dead and such.

Sharplike’s pretty cool and has a lot of features, but not enough to make a full game by itself; not having to build your own systems would make for a lot of cookie-cutter roguelikes. So, just off the top of my head, a list of features we’re going to need:

* Character creation
* Dungeon generation algorithms
* Items
  + Inventory/inventory management
* Monsters
  + The player class (a subclass of a monster)
* Combat system
* Magic

So—let’s get started!

## Setting Up A New Game

We spent a lot of time trying to figure out how to make setting up a game quick and painless. I think we mostly succeeded. I’m not going to include massive code listings in this document, because I don’t think they’re overly helpful. When I need to refer specifically to code, I’ll do so with short snippets, or just tell you to look at the demonstration sample in your Sharplike solution.

My game is going to be part of Sharplike, so I’m going to be creating it within the Sharplike solution. As I’m naming it Sharphack, the logical project name is Sharplike.Demos.Sharphack, and I’ve created a project to this end. (It’s not a *lot* harder to create a project outside of the Sharplike solution, but you’ll need to be cognizant of a few things I’m getting to avoid here. In particular, check out the wiki for information on Mono.Addins and its behaviors. Not a big thing, but important.)

### Creating the Project

So I’ve created a Console Application with the name Sharplike.Demos.Sharphack, and I have the usual project shell—a Program.cs file and nothing else. First things first: I’m going to need a *glyph set* (a set of tiles for the game), and I prefer to get that set up before I do any coding, so I’m going to add **curses\_640x300.png** to my game. (Credit where credit’s due: this file is a modified version of the base DOS/IBM glyph set used by Tarn Adams’s Dwarf Fortress[[2]](#footnote-2).) It’s in trunk/Tilesets, and I add it to my project via right-click->Add->Existing Item. It’s now in my project, and copied to the root directory of my project. I’m going to need it in the root directory of my builds, though, and so I click on the file in my Solution Explorer. The Properties Window below the Solution Explorer fills with more properties than I know what to do with, and I change “Copy to Output Directory” to “Copy if newer.” So now, once I actually start writing code, my glyph set will be in the right directory from the start. (Yaaaaay.)

Last thing to do before starting to write code is adding my .NET references. I leave System and System.Xml in the list, remove System.Data because I never, ever use it, and add System.Drawing, Sharplike.Core, Sharplike.Mapping, and Sharplike.UI to the references list. I’m pretty sure that I’m now good to go.

### Writing the Bootstrap Code

So I open up Program.cs and am greeted with the usual boilerplate code. While I set this application to a Console Application, I know I’m going to need to add the STAThread attribute to the main function: our OpenTK code uses a GLControl within a Windows Form object, and WinForms requires that the main thread use the single-threaded apartment threading model in order to play nicely with COM. So my main function now looks like this:

[STAThread]

static void Main(String[] args)

{

}

Awesome. Simple as pie. Anyway, let’s get started with some basic startup code. There’s a little bit of annoying boilerplate to do to build a game, but there’s not a whole lot that can be done about that. When you’re done, it’ll look something like the Main() function of Sharplike.Demos.Sharphack; I won’t copy it here because it’s well-commented and straightforward. Consult the wiki if you have any questions as far as that boilerplate (in particular, many people might be thrown off by the .SetXSystem calls, which are wrappers for Mono.Addins and our dynamic frontend and audio system loading. Those are as close to magic as anything you’ll see in Sharplike—you don’t need to care about them.

So now that we’re asking for a **commands.ini** file, we should probably build one. They’re easy. I just took the one out of Sandbox and edited it a little bit for my (initial) purposes here:

[KeyBindings]

[KeyBindings MainMenu]

Escape = quit

Q = quit

N = new\_game

L = load\_game

C = credits

The format of a **commands.ini** file is pretty simple. Keys can map to commands, which are just arbitrary strings you specify. In the above example, “Escape” binds to the “quit” command. The “quit” command doesn’t actually do anything yet, mind you: it’s just a string. You have to wire up functionality yourself. Fortunately it’s pretty easy. The sections in square brackets specify what *control state* you want these bindings to be active in. The first KeyBindings section is global and generates commands in every game state—if you’re using the StateMachine class, it uses this by default, but you can trigger input state changes yourself even there. While it calls it KeyBindings, this name is slightly misleading: mouse button presses are defined as LButton, MButton, RButton, XButton1, and XButton2 (the latter two for five-button mice like my Logitech MX518). The scroll wheel is handled by generating PgUp and PgDn events; while they look the same as the keyboard events in **commands.ini**, they can be distinguished in code from the keyboard button presses via the **CommandData.IsMouseEvent** property.

I don’t want any application-wide key bindings yet, so that INI entry is empty. For the main menu, I’m going to want to be able to create a new game, load an old game, show some credits, or quit the game. I decided I wanted Quit to respond to either Q or Escape, so I created entries for both of those that bind to the same command.

1. <http://tortoisesvn.tigris.org/> [↑](#footnote-ref-1)
2. <http://bay12games.com/dwarves> [↑](#footnote-ref-2)