Razor! Developer Companion

Last changed 2003-05-31

1 This is the “MIT” license. See www.opensource.org.

Table of Contents

The Razor! Framework

Razor! is a highly modular, easy to use, embeddable presentation engine for Palm Powered™ devices. It makes it easier for developers to create multimedia software, such as animations and games. The engine is not a ready-to-use executable. Instead it comes as a bundle of C++ source files which need to be customized, compiled and linked with your own code[2].

Its main features include

- Timing + Flow control - Razor! ensures the correct order of all actions, and their precise timing.
- Canvas management - Razor! manages the drawing area (aka Canvas). It provides double buffering with optimized copying.
- Sprite engine - Razor! can draw sprites (small bitmaps with transparent parts). It manages their position, shown/hidden state, etc.
- Sound engine - Razor! can play music (faking three voices), and sound FX.
- Input management - Razor! can detect the status of control devices (rocker switch, 5-Way Navigator, buttons), and direct other events to your app.

Razor! is written in C++ and is delivered as a so-called *hybrid framework*.

The nature of a Framework

It is important to understand the nature of a framework, especially in comparison to a class library.

The Hollywood Principle

Frameworks operate by the „Hollywood principle“ (don't call us, we'll call you). That means, the framework controls the entire flow of your application's execution, calling your code in a few well-defined places. When you are using a class library, it is just the other way around: You control the flow, and make calls into the library.

A framework is harder to design than a class library, but provided that your application fits the domain of the framework, it can save you a lot of work, and provide results of superior quality, because you are not only using proven code, but also proven patterns for application design.

In reality, you will rarely find a pure framework. Most frameworks are also accompanied by a class library. These are called *hybrid frameworks*.

2 Some users have successfully ripped the code apart to incorporate parts of it into their C programs. The license permits this kind of use.

Hotspots

The places in your code, which are invoked by the framework, are called *hotspots*. These are the places where you are given the chance to customize the behavior of the resulting application. The control flow between the invocations to your code is defined entirely by the framework, and cannot be modified by you.

Application development with the Razor! framework

The Hollywood principle applies

As has been explained in “The nature of a Framework“, the behavior of an application which is based on a framework is defined by the code which you provide to the framework for invocation during program execution.

This fully applies to the Razor! framework. The entire main application, including the startup code, the event loop, all the decisions about control flow, are provided by the Razor! framework. The framework will make calls into code which you will have to provide. This is described in detail in the chapter on “Framework Control Flow”.

Example Project

Look at the SimpleDemo project in the `Projects/SimpleDemo` folder for a working project. Having this project handy while reading this companion is probably helpful.

If you want to try out things as you read along, you can either manipulate the SimpleDemo project, or create your own new project, as described in Appendix A. Install and build instructions are also in this Appendix.

API Documentation

The API documentation is contained in a separate document. The API documentation and this Developer Companion document complement each other. This document provides a high-level conceptual overview, whereas the API documentation provides the details of the API, but hardly explains any concepts.

Framework Control Flow

The framework executes as described in this chapter.

Application Startup

The following sequence is executed every time an application based on Razor! is started.

1. Low-level setup

   The framework starts up just like any Palm OS application. It determines the type of device, sets up an event handling loop, ...

   See “Framework anatomy: Primordial Soup & Class Presentation”

2. Display setup

   The bitmap metadata is read and the physical display is setup

   See “Appendix B” and “Class Display”

3. `ActionEngine` setup

Your `MyActionEngine` class is instantiated through its constructor, and then initialized through its `init()` operation.

See "Framework anatomy: Class ActionEngine & Screens"

4. Begin Application Excecution

Application execution

The application is executed until a return to the launcher is requested by the user. The framework supports this automatically through its event handling.

1. Screen requested from ActionEngine

   `ActionEngine::getCurrentScreen()` is invoked.

   See "Framework anatomy: Class ActionEngine & Screens"

2. Screen executed

   See "Screen Execution"

3. Next screen requested from ActionEngine

   `ActionEngine::getNextScreen()` is invoked.

   See "Framework anatomy: Class ActionEngine & Screens"

4. Go to step 2, or exit the application

Screen Excecution

The screen is executed until it indicates that it is finished. The execution loop runs at a controlled speed, making sure the application is making a fluid impression on the user[3].

1. Screen queried for display properties and viewport properties

   The screen is first queried for its expectations about the physical display. The framework then sets up the display canvas. After that it queries the screen for its expectations about the viewport. The calculations for the viewport can use information from the display canvas. The framework then sets up the viewport canvas.

   See "Display Management"

2. Screen initialized

   `Screen::init` is invoked. This is a good place to set up the internal state of the screen and to prepare resources for use during the paint phase.

3. Screen set to next period

   `Screen::nextPeriod` is invoked to ask the screen to advance its state to the next period. This could include moving sprites, calculating computer AI, etc.

   Do not draw anything during this phase, just advance the state and return.

4. Screen contents painted

---

3 Temporal precision is the most important factor in creating the illusion of smooth execution of a game. Much more so than spatial precision (i.e. how many pixels the objects move per timestep).

`Screen::drawBackground` and `Screen::drawAction` is invoked to draw to the viewport canvas.

`Screen::drawFixedOverlays` and `Screen:drawDynamicOverlays` is invoked to draw to the overlay canvas.

The background and the fixed overlays are only drawn when their respective canvasses are considered in need of a redraw.

Never change the internal state of the screen in this phase. Only draw the visual representation of the current state of the screen, then return.

See “Display Management”

5. Go to “Screen set to next period”

Framework anatomy

Primordial Soup & Class Presentation

Deep in its bowels, Razor! is an ordinary Palm OS application, of course. It has a `PilotMain()` and an event loop just like every other application. These are contained in a file called `Starter.cpp`, and do not belong to any class. Starter delegates the actual control flow to a class called `Presentation`.

Class ActionEngine & Screens

The Presentation expects your code to be presented in child-classes of two framework classes called `ActionEngine` and `Screen`. These two classes contain all the hotspots that will be invoked during execution.

The child-class of ActionEngine needs to be called `MyActionEngine`. It is responsible for instantiating the Screens and presenting them in a proper order. It is also responsible for saving and restoring the application's state between invocations.

The Screens are the distinct sections within the game. There could be a title screen, a highscore screen, a configuration screen, a game screen, a bonus level screen, etc.

Example

In the SimpleDemo example you can see a class definition for MyActionEngine, and 3 screen classes, called FlyingChickenScreen, RectangleScreen, and FlatshadeScreen.

MyActionEngine presents the 3 screens in alternating order until you exit the application.

Display Management

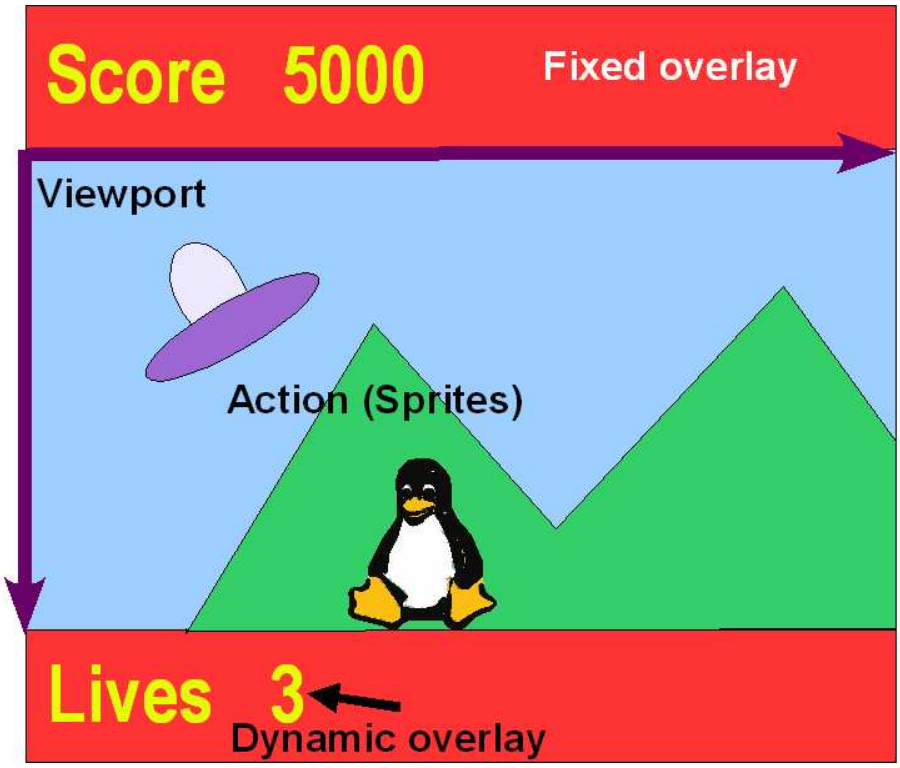

Figure 1The components of a display

The display components

The display is divided into two distinct sections. The viewport and the overlay areas.

The *viewport* is a double-buffered section in the middle of the display. Everything outside the viewport is part of the overlay area. It is perfectly valid to specify the viewport to cover the entire screen. Specifying a smaller viewport will mostly give you more performance, and save some memory.

The viewport contains a background and action elements (Sprites), which are drawn over the background. Since the sprites can store and restore the section of the background which they cover, the background only needs to be painted once when it changes, whereas the sprites are typically painted during every frame of animation.

The *overlay area* is made up of static and dynamic parts. Static (fixed) parts of the overlay are those that never change, such as the colored backdrops, and the words "Score" and "Lives". Dynamic parts are those parts that change their appearance, such as the actual score, or the actual number of lives. Even though the overlay area is not double-buffered, it is still possible to manipulate it in a limited way, without producing visible flicker.

Coordinate systems

The display uses two different coordinate systems. One for the viewport, and one for the overlay area. The (0,0) coordinate of the viewport is in the topleft edge of the viewport, which is not necessarily identical to the topleft edge of the physical display. The (0,0) coordinate of the overlay area is at the topleft edge of the physical display.

You have to use viewport coordinates when you are drawing the background and the action (`Screen::drawBackground`, `Screen::drawAction`). You have to use overlay coordinates when you are drawing overlays (`Screen::drawFixedOverlays`, `Screen::drawDynamicOverlays`).

Logical vs. Physical Coordinate systems

There is a wide variety of devices on the market. Older ones are lores, newer ones are hires. In order to make it easier to develop applications that support both families of devices, the framework is making a distinction between the physical resolution of the display, and the logical coordinate-system that you use to draw on it. The physical resolution is set during application startup and cannot be changed subsequently[4]. The logical coordinate system however can be specified on a per-Screen basis. There are two modes:

1. STANDARD (the default): The logical coordinate system emulates a lo-res device, independent from the actual physical coordinate system. On hires devices you can still use hires bitmaps to make your application look extra neat, but you can only place them at lo-res coordinates.
2. NATIVE: The logical coordinate system is identical to the physical coordinate system. It is up to your application to deal with any display size that it may encounter.

This is very similar to the mode of operation of Palm OS 5, but is made available on all devices, including the OS 4 based Sony Clies.

See `Canvas::CoordSystem` and `Screen::getDisplayProperties` to find out how to override the default.

Display depth

The entire display has a depth in bits per pixel (bpp). Depths 1, 2, and 4 bpp have a grayscale palette. Depths 8bpp and higher have a color palette. The display depth is determined once during the initialization of the display (see Class Display) and cannot be changed during runtime.

Class Display

The *Display* class is responsible for setting up the physical display. This includes setting the resolution, and the display depth. It also establishes the double buffer for the viewport. It can provide access to all display properties, most importantly the display depth in bpp.

Physical display setup is done based on metadata that you have to provide along with your bitmap resources. See Appendix B for details.

Class Canvas

The *Canvas* is an area on which you can draw. It can be a representation of the physical display, or be offscreen[5]. The Canvas has both a logical size (width/height), as well as a physical size (cols/rows). This distinction is important if you wish to access the bitmap memory of the Canvas directly.

---

4 Most devices would not support this, and those that do tend to flicker annoyingly.

5 In the current implementation, the “viewport” is an offscreen Canvas, whereas the “overlay” area is a Canvas that is directly linked to the physical screen.

The Canvas supports a variety of primitive graphics operations which are commonly needed for game development. You can either invoke those operations directly from a Screen's drawing operations, or you can use classes which wrapper invocations. The most prominent one being the Sprite class.

- Drawing transparent bitmaps
- Drawing non-transparent ("solid") bitmaps
- Drawing filled rectangles
- Drawing text
- Drawing filled polygons

While these might be enough for most games, you will probably miss some operations. You can use the OS to draw to the Canvas, and you can also access the bitmap of the ***viewport*** canvas directly[6]. See "Direct Access to Canvas Memory".

Class AnimFrames, Sprites & SpriteGroups

Sprites are certainly the most important element of a game. They are well-supported in this framework. Sprites have a position, a visibility state, and a currently displayed frame of animation.

AnimFrames manage the frames of an animation. Several sprites can have a reference to the same instance of AnimFrames, which can save a considerable amount of resources, if you have lots of identical sprites on the screen.

Example

If the aliens attack with 5 motherships, and 95 UFOs, you would have two AnimFrames, one containing the frames of the mothership, the other containing the frames of a UFO. You would then instantiate 100 Sprites, and let 5 of these reference the AnimFrames of the mothership, and let the other 95 point to the AnimFrames of the UFO. Each of the 100 sprites would have a completely independent position, and each one could show a different frame of its associated AnimFrame.

SpriteGroups are used to more efficiently handle large numbers of identical sprites, i.e. Sprites which use the same AnimFrame. Creating a SpriteGroup of 100 sprites is a lot more memory efficient than creating 100 separate instances of the Sprite class. Entire SpriteGroups can be drawn with a single command.

Example continued...

Instead of creating 100 sprites for the alien attack fleet, you should create 2 SpriteGroups, one containing the 5 motherships, the other containing the 95 UFOs.

Direct Access to Canvas Memory

Many interesting graphics effects are not provided directly by the drawing operations of the Canvas class. But you can achieve anything you want by directly accessing the memory that represents the viewport[7] canvas. Such drawing code is therefore invoked during the Screen::drawAction operation.

The canvas memory is organized like most generic framebuffers. For a display depth of n bits per pixel, the first n bits of the screen are the topleft pixel. The following n bits are the pixel to the right of that pixel, and so on.

---

6 If you have written an interesting graphics routine you are invited to contribute it (hint, hint).

7 Directly accessing the overlay display canvas is a big no-no, and will lead to crashes on many devices.

The Display class contains information about the depth in bpp (`Display::depth`). This depth is the same for all canvases. The Canvas class contains information about size (both physical and logical), and memory location (`Canvas::drawBits`).

You can access the memory using any combination of C++ and assembly code that you want. Make sure you are setting the dirty rectangle properly in order to see the results.

Optimizing for Maximum Performance

Make sure you've got a problem

Do not optimize performance unless you actually have a performance problem.

Play your game. If it's not sluggish why optimize it? Always remember: Performance optimizations are a great way to introduce bugs.

Establish the Facts

Use the profiling version of the POSE emulator to obtain reliable profiling data about your application. Do not take guesses! They will be wrong. Keep a snapshot of this data. This will be your "baseline". If you want you can visualize it using the "ProView" tool from www.tilo-christ.de/proview. Establish a performance goal. Pinpoint those 20% of code that eat up 80% of the execution time. Focus on those. Optimize your application until you have reached the performance goal. After every step of optimization you should obtain new profiling data to make sure that performance has actually improved.

Then stop optimizing, and continue to develop more fun features for the game!

Designing For Speed

While Razor! places no restrictions on you while you design your application, sticking to certain restrictions and conventions will allow it to operate with greatly improved speed.

Sprites

- Make the width of your sprites a multiple of 8
- Make the x-position of your sprites a multiple of 2
- Use 8bpp for color, and 4bpp for greyscale (*Caution:* 4bpp greyscale is only available on devices with OS3.5+ and an EZ CPU)
- Create SpriteGroups instead of single Sprites. This will reduce memory fragmentation.

Graphics primitives

- Make the width of rectangles a multiple of 8
- Make the x-position of the left edge of rectangles a multiple of 2

Reduce drawing power

- Only use the Canvas draw operations for drawing. This will ensure you are not using slow OS routines, but the most optimized routines.
- Don't redraw everything during each frame. Try to place objects to the background, where they will remain unharmed by sprites.
- Make the viewport as small as possible. Make use of overlays.
- Try to avoid transparency.

Extra Performance Hints

Don't play music during gameplay.

Appendix A: Installation and Compilation

Prerequisites

Developer

Using Razor! requires C++ skills and a basic understanding of Palm OS programming. It will in no way enable you to just "design" a game, and then input it through some kind of high level tools, like some game development environments on the PC allow you to do. But, as one user recently noted:

> ***** [Jan 11, 2003] by **jbrown**
> This is a really good toolkit. If you are a beginner with some so-so programming skills you can learn it really fast. Much easier than learning the details of the Palm SDK.

Build tool

- ***Codewarrior 6***, or higher

  or

- ***PRC-Tools 2.2*** (http://sourceforge.net/projects/prc-tools/)
- ***PilRC 2.9p10*** or higher is highly recommeded, and is required to compile the example application.

SDKs

For build tool specific installation instructions for the various SDKs consult the chapter on Build Tool Specifics.

Palm OS SDK

SDK 5.0 is recommeded[8]. SDK 4.0 Update 1 is still supported, but *NOT* recommended.

The SDK version needs to be at least the OS version of the OS on the target devices but may well be higher. You can use SDK 5.0 to target anything from OS 3.0 to the latest OS 5 devices.

Third Party SDKs

An additional Sony SDK is required if your compiled software shall work perfectly on pre-OS 5 Sony Clie devices. The SDK can be downloaded from http://www.cliedeveloper.com/develop_tool/sdk_50.html.

The Five Way SDK is required if your software shall support the 5-Way Navigator on the Tungsten T. The SDK can be downloaded from http://www.palm.com or through a membership in Palm's PluggedIn program.

Razor! Distribution

You can unpack the Razor! Archive to an arbitrary directory. Make sure path names and case of file names are preserved. Use WinZip 8.x on Windows, or "unzip -U" on Linux.

---

8 Not using SDK 5.0 will bereave you of Hires support, which would be a pity.

Razor! Directory Tree

The following path specifications are relative to the directory where you unpacked the Razor! distribution file. Several projects can be developed within the same directory tree if you choose a different project name for each one. The project name of the sample project is “SimpleDemo”.

Razor framework code

`Razor/Src`

Should not be touched unless you have very specific needs which are not filled by the unmodified framework.

Razor framework resources

`Razor/Rsc`

Should not be touched.

Project specific code

`Razor/Projects/<Project name>/Src`

Application code that you will need to write.

Project specific resources

`Razor/Projects/<Project name>/Rsc`

Application specific resources that you will need to provide. Consists of application icon, menus, “About” dialog, music, sound fx, and most importantly the bitmap resources.

Project specific framework config file

`Razor/Projects/<Project name>/Src/Customization.h`

Determines some very basic aspects of the framework's configuration for the project, such as the SDKs that are available for compilation, and whether some official Palm programming guidelines may be ignored.

Build tool specific project file

`Razor/Projects/<Project name>`

Depending on your build tool this can either be a GNU Makefile for prc-tools, or a Codewarrior project file.

Build Tool Specifics

CW 6 (Lite)

You should install the following extra software:

- All CW 6 updates from Metrowerks (essential)
- SDK 5.0 for Codewarrior (essential)
- Sony SDK (recommeded but not essential)
- PilRC plugin 2.9p10b for Codewarrior from www.calliopeinc.com/pilrcplugin.html (essential for the SimpleDemo project, and strongly recommeded for your own work)

Palm SDK 5.0

SDK 5.0 is not part of the CW6 package. In order to get SDK 5.0 to work with CW6, you will have to download the SDK 5.0 for Codewarrior from Palm Source (www.palmsource.com), then follow the instructions for a manual installation that come with the SDK. Make sure you install the PalmRez post-linker from the SDK. Otherwise the PilRC plugin might crash the IDE.

Third Party SDKs

In order to guarantee faster success without having to install several third-party SDKs first, the default for the framework is to compile with the Palm SDK 5.0 only, and not to support the third-party features (Clie, 5 Way Navigator). Activate support by doing the following:

- Clie support:
    - Obtain the Sony SDK
    - Install Sony SDK to the "`Sony SDK Support`" subdirectory in your CW installation.
    - In `Src/CWInstalledSDKs.h`. Specify `#define SUPPORT_CLIE`
- Five Way support:
    - Obtain the Five Way SDK
    - Copy the file `PalmChars.h` into your SDK 5.0 installation, e.g. to `Incs/Core/System/PalmChars.h`
    - In `Src/CWInstalledSDKs.h`, specify `#define SUPPORT_FIVEWAY`.

Building and Running

Load the project file `CW6SDK50.mcp` select the 'Debug' target and build. If this works you can also try the 'Release' target.

Caution:

The optimization options for the Release target have been set to Level 2. Higher levels optimize the framework to death.

Due to bugs in the current version of the PilRC plugin, it is currently not possible to build working hires applications with this environment.

CW 7

“Just thought you might want to know that it works just fine with CW7, although I did have a problem when I unpacked the new source over an old installation. Maybe a README1ST file in the distribution warning that you should install clean ? Also, does turning off the postlinker make any difference? All I did to get going was open the project (`CW6SDK50.mcp`), unselect the sony sdk (because it isn't installed) and hit the make button and everything was fine. “

CW 8

1. Open the project (`CW9Project.mcp`)
2. Say OK to the dialog that warns that the project is for a newer version of CW
3. Set the linker to "Macintosh 68K" and set the Post-linker to "PalmRez Post Linker" for the Debug and Release targets
4. Build the project

CW 9

Your environment already comes with all required tools and SDKs.

You might want to install the following optional software:

- Latest update patches for CW 9 (recommeded but not essential)
- Palm Five Way SDK (recommeded but not essential)

Third Party SDKs

In order to guarantee faster success without having to install several third-party SDKs first, the default for the framework is to compile with the Palm SDK 5.0 only, and not to support some of the third-party features (Five Way Navigator). Activate support by doing the following:

- Clie support

This SDK is already included with CW9, but support for it is not enabled by default, because most of the other build tools do not include this SDK.

    - In `Src/CWInstalledSDKs.h`, specify `#define SUPPORT_CLIE`

- Five Way support:

    - Obtain the Five Way SDK.
    - Copy the file `PalmChars.h` into your SDK 5.0 installation at `Palm OS Support/Incs/Core/System/PalmChars.h`
    - In `Src/CWInstalledSDKs.h`, specify `#define SUPPORT_FIVEWAY`.

Building and Running

Load the `CW9Project.mcp` project file. The IDE might complain about a missing directory. Ignore this message. Choose one of the targets “Debug”, “Debug-Hires”, “Release”, or “Release-Hires”. Build the target. Use the CW IDE debugger or a real device to run the application.

PRC-Tools 2.2

You should install the following extra software:

- SDK 5.0 for PRC-Tools (essential).
- Sony SDK (recommeded but not essential).
- Palm Five Way SDK (recommeded but not essential)
- PilRC 2.9p10 (essential). A suitable version for use with Cygwin can be downloaded from http://sourceforge.net/projects/razor-engine/ . Read the release notes for installation instructions.

Palm SDK 5.0

Make sure you obtain the latest version for ***PRC-Tools(!)*** which includes the proper Glue libraries from www.palmsource.com. If you have freshly installed SDK 5.0 make sure you run "palmdev-prep" afterwards to let PRC-Tools know about the existence of the new SDK. Palmdev-prep should output a message stating that it made SDK 5 your default SDK.

Third Party SDKs

In order to guarantee faster success without having to install several third-party SDKs first, the default for the framework is to compile with the Palm SDK 5.0 only, and not to support the third-party features (Clie, 5 Way Navigator). Activate support by doing the following:

1. Clie support
    - Get the SDK. Make sure you convert it to work with PRC-Tools using the instructions from www.falch.net.
    - In `Makefile.incl`: Set "`SONYSDK`" to "yes"
2. Five way support:
    - Get the SDK. Copy `PalmChars.h` to `/PalmDev/FiveWaySDK/PalmChars.h`.
    - In Makefile.incl: Set "`FIVEWAYSDK`" to "yes"

Building and Running

Your `/PalmDev` directory tree with the SDKs installed should look like this:

Change to the `Projects/SimpleDemo` directory and run '`make`'. Run the resulting PRC files.

Falch.net

Make sure your version of Falch.net is based on PRC-Tools 2.2. Also, since this is essentially a PRC-Tools environment the instructions regarding PRC-Tools might help in case of problems.

1. Make a new Framework project with DeveloperStudio. Name it what you want, but all of the other settings don't really matter.
2. Go into the directory that devstudio created for the new project. Delete all files except for the .FNP file.
3. Go back to devstudio and remove all of the files from the project (right click on the file, press R on the keyboard)
4. Go into the SimpleDemo folder that comes with Razor! and copy all of the files into your new project directory. You can delete the codewarrior directories, etc
5. Go into the project properties, and click Advanced... uncheck the first three check boxes (for building a makefile, def file, etc). MAKE SURE THESE ARE UNCHECKED. Click ok
6. Right click on the project name in the Project Explorer and click add existing file... Find the makefile that you moved over from simpledemo
7. Do the same for the .cpp and .h files moved over from simpledemo under Source and Headers, and the resource files under Resources
8. Click compile.... it should work fine.
9. If you want to add files to the project, you have to edit the makefile by hand and add a file to the compile list (I couldn't get falch.net working with its automatically generated makefiles, so I just have it use a custom makefile)

Setting up a new project

In order to develop a new application with Razor!, you have to

- Set up a new project in the `Projects` subdirectory. The easiest way to do that is to copy the SimpleDemo project into a new folder.
- Create your resources (bitmaps, music, menus, icons)
- Create your Screens
- Create your MyActionEngine
- Build and run

TBD: Write sth. Really helpful :-(

Appendix B: Preparing artwork (bitmap resources)

Bitmaps

Bitmaps are grayscale or colorized images, that will be blitted to the screen. They are always rectangular in shape, but some parts of them may be declared as transparent.

Bitmaps have two relevant properties: Color-depth and resolution. If you wish to support a specific device you need to make sure that your application is equipped with the necessary bitmap resources for that device.

Example

Supporting a Palm Vx, which is a 4bpp lo-res device requires 4bpp lo-res bitmaps (2bpp and 1bpp could also be used).

Supporting a Sony Clie, which is an 8bpp hires device requires 8bpp hires bitmaps.

You can place several variations of the same bitmap into your application, in order to support multiple devices with it.

Bitmap Families

Variations of the same bitmap are grouped together into a "bitmap family". Family members may vary in resolution and bit depth, but not in size.

Bitmap Family Identifiers

Each bitmap family is identified by a numerical identifier. The identifier identifies the entire family. There is no way to address a single member of a family.

Family layouts

The following table lists all currently supported bitmap family layouts and their consequences.

It is recommeded you build at least two completely separate builds of your application. One containing only lo-res families for all lo-res pre-OS5 devices, and one "mixed family" build for all hires devices, and maybe the lo-res color devices.

Example

For a build that supports all lo-res devices you can create bitmap families that contain 2bpp, 4bpp, 8bpp lo-res representations of your bitmaps.

For a build that supports all hires devices you can create bitmap families that contain 8bpp hires representations of your bitmaps. For technical reasons, these bitmap families will also have to contain a lo-res representation of your bitmaps. You could either choose 1bpp lo-res images to keep application size small, or you could use color lo-res images, in order to make this a build for all color devices (both lo-res and hires).

Layout	*Remarks*
Lo-res family only (bitmaps of density 1)	Will work on all devices, but will be lo-res only
Mixed family (bitmaps of density 1, and density 2)	Will work on lo-res devices, Sony Clie, and all OS5 devices. *Might be problematic because of bitmap family size!*

Transparency

Transparent bitmaps and Sprites require the definition of a mask for each bitmap. The mask defines which parts of the bitmap shall be transparent. A white pixel in the mask means transparent, a black pixel means opaque.

It is highly important that masks are only declared with a pixel depth of 1bpp. Otherwise strange color effects may occur (and it would be a waste of resources as well).

Considerations

Bitmap Family size

It is important that none of the bitmap families you define are larger than 64k. This is due to a limitation in Palm OS, that cannot be overcome by this framework.

Compression

Some versions of Palm OS do not support compressed bitmaps properly. Do not use compressed 1bpp and 2bpp lo-res bitmaps for opaque drawing. Transparent bitmaps are not affected.

Background color

Transparency is currently based on a mask. Unfortunately, the masking can fail on older devices, if the masked out area of the bitmap is not white. You should therefore make sure that your 1bpp and 2bpp transparent bitmaps use white as their background color.

In upcoming versions, Razor! might honor the transparent color of a bitmap - as specified in the bitmap definition - as an alternative to masks. For your color bitmaps you might consider using a color in the transparent area that you are not using elsewhere in the image.

Bitmap Metadata

It would be hard for the framework to figure out reliably which display depths and resolutions you wish to support with your bitmap resources. You will therefore have to provide metadata along with your bitmaps. Metadata is contained within a resource record. It contains several density/depth pairs. The framework will iterate over these pairs during application startup and setup the display for the first pair that can be used on this device.

Example

You have specified (hires/8bpp, lores/8bpp, lores/2bpp) in your metadata.. The device is a Palm Vx (lores/4bpp). During application startup, the framework will dismiss the first two pairs, because the device cannot support them, and will then switch the display to a depth of 2bpp.

Using PilRC

PilRC is the recommended tool for the preparation of bitmap resources. You can use Constructor from the SDK as well, but the process of turning existing artwork into resources is much more streamlined with PilRC.

Artwork can be created using any paint program that can output the BMP image format. These BMP files are then turned into resources for Razor! using the PilRC-tool[9].

9 PilRC comes bundled with Codewarrior 8, or higher, and can be obtained separately from http://www.ardiri.com

PilRC is provided with an input file[10] that controls how resources are created from the BMP images.

Example for PilRC 2.9 before 2.9p5

Versions of PilRC 2.9 before 2.9p5 can only create bitmap families with density 1. That means lo-res families, and hires families with density 1. This is limiting in that it will not allow you to create applications that make use of high resolution graphics for OS5 devices. It is recommended that you upgrade to the latest patched PilRC 2.9[11].

Look at the `bitmaps.rcp` file that comes with the SimpleDemo example.

Bitmap metadata

Metadata comes in pairs of density and depth indicators. The end of the data is indicated by two 0x00 bytes. The resource type is 'Tbmt' (Bitmap Meta), the ID is 1000. The best modes should come first, since Razor! will pick the first mode that is applicable on the given device.

1. The density indicator for the resources

   Value 0x01 means lo-res (density 1) resources

   Value 0x02 means hires (density 2) resources

2. The depth indicator for the resources

   Value is the depth in bpp, eg. 0x01 = 1bpp, 0x10 = 16bpp

```
HEX "Tbmt" ID 1000
    0x01 0x08
    0x01 0x04
    0x01 0x02
    0x00 0x00
```

Bitmap families

A bitmap family is declared through a BITMAPFAMILY statement. Each of these will prepare a single bitmap family with a resource ID and associated images for each supported display depth.

The syntax is:

`BITMAPFAMILY ID` *<resource id>* `"`*<1bpp BMP file>*`" "`*<2bpp BMP file>*`" "`*<4bpp BMP file>*`" "`*<8bpp BMP file>*`" COMPRESS`

The filenames may either be specified, or left empty, if that display depth shall not be supported by your game. You may specify the same file for several of the display depths. This will cause PilRC to run it through a crude color mapping for each depth, which may or may not be satisfactory.

You can also create masks with BITMAPFAMILY. Make sure you specify only a single filename for the 1bpp depth, and leave the other filenames empty.

Syntax:

`BITMAPFAMILY ID` *<ID>* `"`*<B/W mask BMP file>*`" "" "" "" COMPRESS`

10 The usual file extension is `.rcp`

11 Version 2.9p10 at the time of this writing

Example for PilRC 2.9p5 and later

These versions of PilRC allow you to create bitmap resources for all Palm devices, including the very latest OS5 devices.

Look at the `hiresbitmaps.rcp` file that comes with the SimpleDemo example.

TBD: Give more detailed instructions