Navigating the Playground SDK™

A User and Reference Guide
For Playground SDK 4.0

by Tim Mensch

Navigating the Playground SDK™

Published by PlayFirst, Inc.
160 Spear St, Suite 1300
San Francisco, CA 94105

Copyright © 2007-2009 PlayFirst, Inc. All rights reserved.

PlayFirst and Playground SDK are trademarks of PlayFirst, Inc.

Microsoft®, ActiveX®, Internet Explorer®, Windows®, Windows Vista®, Developer Studio®, Visual C++®, Visual Studio®, MSN®, and DirectX® are registered trademarks of Microsoft Corporation in the United States and other countries.

Mac OS® is a registered trademark of Apple Computer, Inc.

America Online®, AOL.com®, and AOL® are registered trademarks of AOL LLC.

Macromedia®, Flash®, and Director® are registered trademarks of Macromedia, Inc.

This book is an independent publication and is not affiliated with, nor has it been authorized, sponsored, or otherwise approved by Microsoft Corporation.

Book was generated using Doxygen and LaTeX.

Cover photo by Gabriela Rojas.

Printed by http://LuLu.com.

Print Version 2.0

Contents

0.1 Playground SDK™ Documentation vii
0.2 Playground SDK™ Design vii

I User's Guide **1**

1 Introduction **3**
1.1 Welcome to the Playground SDK™! 3
1.2 What's on the Playground? 3

2 Getting Started **7**
2.1 How to Play on the Playground 7
2.2 A First Example Program 9
2.3 Game Assets 12
2.4 The Game Window 12
2.5 Dealing With User Input 16
2.6 Why Modal Windows Are Your Friend 16
2.7 How to use Playground SDK™ features. 17
2.8 What is Lua? 20
2.9 Lua Makes it Easy 20
2.10 What About Speed? 21
2.11 What are the pitfalls? 21

3 Playground Fundamentals **23**
3.1 Dynamic Casting 23
3.2 RTTI in TWindow 23
3.3 How to Share and Play Well With Others 24
3.4 Using shared_ptr (TClassRef) Classes 24
3.5 Managing Assets 24
3.6 Common Mistakes 25
3.7 Your Very Own shared_ptr 25
3.8 Implementation Details 26
3.9 About the Playground SDK™ Hiscore System 27
3.10 Initialize The System 27

3.11 Connecting to a debug server . . . 27
3.12 Set Properties . . . 28
3.13 Logging Scores . . . 28
3.14 Logging Medals . . . 28
3.15 Viewing local scores . . . 28
3.16 Figuring out what score the user can submit . . . 29
3.17 Submitting a score and medals . . . 29
3.18 Switching to the global score view . . . 30
3.19 Hiscore FAQ . . . 31
3.20 Writing Debug Messages . . . 33
3.21 Dynamic Debugging . . . 33
3.22 Game Version Numbers . . . 34
3.23 Flat File Basics . . . 35
3.24 Creating Your Own Flat File . . . 35
3.25 Appending Files . . . 35

4 Beta Versions **37**
4.1 How to Create Limited Builds for Beta Testers . . . 37
4.2 Tracking Game Metrics . . . 38
4.3 Writing Status on Elapsed Time . . . 40
4.4 FirstPeek Module Reference . . . 41

5 Lua **43**
5.1 About Lua . . . 43
5.2 Using Lua to Script Your Game . . . 43
5.3 How do I get Lua data in my C++ code? . . . 50

6 Particle System **51**
6.1 Particles in a Scripting Language? . . . 51

7 Localization and Web Versions **57**
7.1 The String Table . . . 57
7.2 Quick and Easy Web Games . . . 58

8 Game Footprint **61**
8.1 Smaller is Better . . . 61
8.2 Shrinking Your Game . . . 61

9 Utilities **63**
9.1 FirstStage: The Playground Resource Editor . . . 63
9.2 Building A Dialog Tutorial . . . 80
9.3 FluidFX 2.0: Interactive Particle System Editor . . . 86
9.4 sidewalk: Animation Creation and Asset Compression . . . 92

9.5 Filmstrip: Creating and Editing Animations . . . 96
9.6 3dsconvert: Creating 3d Models For Playground . . . 104
9.7 Creating a Playground Font . . . 104
9.8 axtool: Testing Your Game in a Browser . . . 105
9.9 xml2anm: Convert XML to ANM . . . 105

10 Advanced Features **107**
10.1 Playing With the Big Kids . . . 107
10.2 Deriving a Custom Sprite Type . . . 107
10.3 Sending Custom Application Messages . . . 108
10.4 Calling a Lua Function from C++ . . . 110
10.5 Substituting a Custom TModalWindow . . . 110

II Reference **113**

11 Windowing Reference **115**
11.1 Windowing and Widget Functionality . . . 115

12 Lua Reference **117**
12.1 Lua-Related Documentation . . . 117
12.2 Query Values for Current Configuration in Lua. . . . 122
12.3 GUI-Related Constants in Lua. . . . 124
12.4 Text and Window Alignment. . . . 125
12.5 Defined Message Types in Lua. . . . 126
12.6 Animation Script Functions . . . 128
12.7 Lua Particle System Reference . . . 132
12.8 Lua GUI Command Reference . . . 140

13 Vertex Rendering Reference **157**
13.1 Vertex Support for Triangle Rendering . . . 157

14 Font Reference **159**
14.1 Introduction . . . 159
14.2 Quadratic Bezier Curves . . . 159
14.3 Data Types . . . 159
14.4 Vector Commands . . . 160
14.5 File Format Specification . . . 160
14.6 Font Rasterization . . . 160
14.7 Notes on Font Sizes and Line Height . . . 161
14.8 Call for Help . . . 161

15 Class and File Reference **163**

15.1 str Class Reference . . . 163
15.2 T2dParticle Class Reference . . . 174
15.3 T2dParticleRenderer Class Reference . . . 175
15.4 TAnimatedSprite Class Reference . . . 178
15.5 TAnimatedTexture Class Reference . . . 185
15.6 TAnimTask Class Reference . . . 194
15.7 TAsset Class Reference . . . 197
15.8 TAssetMap Class Reference . . . 198
15.9 TBegin2d Class Reference . . . 202
15.10 TBegin3d Class Reference . . . 203
15.11 TButton Class Reference . . . 204
15.12 TButton::Action Class Reference . . . 213
15.13 TButton::LuaAction Class Reference . . . 214
15.14 TClock Class Reference . . . 216
15.15 TColor Class Reference . . . 218
15.16 TColor32 Struct Reference . . . 221
15.17 TDialog Class Reference . . . 223
15.18 TDrawSpec Class Reference . . . 225
15.19 TEncrypt Class Reference . . . 229
15.20 TEvent Class Reference . . . 231
15.21 TFile Struct Reference . . . 234
15.22 TFlashHost Class Reference . . . 241
15.23 TGameState Class Reference . . . 244
15.24 TGameStateHandler Class Reference . . . 250
15.25 TImage Class Reference . . . 252
15.26 TLayeredWindow Class Reference . . . 256
15.27 TLight Struct Reference . . . 259
15.28 TLitVert Struct Reference . . . 261
15.29 TLuaFunction Class Reference . . . 262
15.30 TLuaObjectWrapper Class Reference . . . 264
15.31 TLuaParticleSystem Class Reference . . . 267
15.32 TLuaTable Class Reference . . . 273
15.33 TMat3 Class Reference . . . 282
15.34 TMat4 Class Reference . . . 290
15.35 TMaterial Struct Reference . . . 299
15.36 TMessage Class Reference . . . 300
15.37 TMessageListener Class Reference . . . 303
15.38 TModalWindow Class Reference . . . 304
15.39 TModel Class Reference . . . 310
15.40 TParamSet Class Reference . . . 314

15.41 TParticleFunction Class Reference . . . 318
15.42 TParticleMachineState Class Reference . . . 321
15.43 ParticleMember Struct Reference . . . 325
15.44 TParticleRenderer Class Reference . . . 326
15.45 TParticleState Class Reference . . . 328
15.46 TPfHiscores Class Reference . . . 331
15.47 TPlatform Class Reference . . . 342
15.48 TPoint Class Reference . . . 357
15.49 TPrefs Class Reference . . . 358
15.50 TPrefsDB Class Reference . . . 363
15.51 TRandom Class Reference . . . 367
15.52 TRect Class Reference . . . 370
15.53 TRenderer Class Reference . . . 376
15.54 TScreen Class Reference . . . 395
15.55 TScript Class Reference . . . 397
15.56 TScriptCode Class Reference . . . 406
15.57 TSimpleHttp Class Reference . . . 408
15.58 TSlider Class Reference . . . 412
15.59 TSound Class Reference . . . 417
15.60 TSoundCallBack Class Reference . . . 420
15.61 TSoundInstance Class Reference . . . 421
15.62 TSoundManager Class Reference . . . 424
15.63 TSpeex Class Reference . . . 426
15.64 TSprite Class Reference . . . 427
15.65 TStringTable Class Reference . . . 435
15.66 TTask Class Reference . . . 437
15.67 TTaskList Class Reference . . . 439
15.68 TText Class Reference . . . 441
15.69 TTextEdit Class Reference . . . 448
15.70 TTextGraphic Class Reference . . . 455
15.71 TTextSprite Class Reference . . . 464
15.72 TTexture Class Reference . . . 473
15.73 TTransformedLitVert Struct Reference . . . 482
15.74 TURect Class Reference . . . 483
15.75 TVec2 Class Reference . . . 485
15.76 TVec3 Class Reference . . . 491
15.77 TVec4 Class Reference . . . 498
15.78 TVert Struct Reference . . . 505
15.79 TVertexSet Class Reference . . . 506
15.80 TWindow Class Reference . . . 509

15.81 TWindowHoverHandler Class Reference . . . 531
15.82 TWindowManager Class Reference . . . 532
15.83 TWindowSpider Class Reference . . . 541
15.84 TWindowStyle Class Reference . . . 542
15.85 TXmlNode Class Reference . . . 546
15.86 debug.h File Reference . . . 553
15.87 pftypeinfo.h File Reference . . . 556
15.88 pflibcore.h File Reference . . . 560

III Appendix **561**

A Forward Declarations **563**
A.1 forward.h File Reference . . . 563

B Change History **565**
B.1 Playground SDK Change Log . . . 565
B.2 Changes to Playground 3.2.1 from Playground 3.2.0 . . . 599
B.3 Changes to Playground 3.2.0 from Playground 3.1.5 . . . 600
B.4 Changes to Playground 3.1.5 from Playground 3.1.4 . . . 601
B.5 Changes to Playground 3.1.4 from Playground 3.1.3 . . . 601
B.6 Changes to Playground 3.1.3 from Playground 3.1.2 . . . 601
B.7 Changes to Playground 3.1.2 from Playground 3.1.1 . . . 602
B.8 Changes to Playground 3.1.1 from Playground 3.1.0 . . . 602
B.9 Changes to Playground 3.1 from Playground 3.0.x . . . 602
B.10 Changes to Playground 3.0 from Playground 2.3.x . . . 604

C Annotated Class Listing **605**
C.1 Class List . . . 605

0.1 Playground SDK™ Documentation

Many people made this documentation possible. I would like to thank Jim Brooks who wrote most of the high score documentation, the web game documentation, some of the utility documentation, and probably other sections I'm forgetting. In this edition, Brad Elelman contributed the chapter describing how the Playground font engine works. Also new in this edition, Robert Crane wrote most of the utilities chapter, as he has owned the tool chain since he started working with PlayFirst. Cover design is by Juan Botero. Maria Waters helped out with valuable typesetting advice.

Thanks also go to the members of the PlayFirst team who spent time proofreading and making suggestions as to how to improve the documentation: Jim Brooks, Dan Chao, Peri Cumali, Joshua Dudley, Brad Edelman, Teale Fristoe, Oliver Marsh, Drew McKinney, Solveig Pederson, Ryan Pfenninger, Shannon Prickett, Reggie Seagraves, and Eric Snider.

And I would also like to thank my wife Deborah, who dusted off her technical editor hat and practically rewrote parts of the user's guide to help make it easier for people to understand.

0.2 Playground SDK™ Design

Tim Mensch is the Playground SDK™ lead at PlayFirst, and he provides much of the design direction, but he doesn't work in a vacuum.

Brad Edelman, the CTO of PlayFirst, contributed the initial groundwork to Playground, including its fast, flexible, and portable font renderer. Brad is the one who makes sure Tim is doing his best work on the library, keeping him focused on the end user experience—both the experience of the user of the game and that of the user of the library.

Jim Brooks has been an ever–willing first consumer of early library features, and has contributed no small amount of code to the library as well. From the OGG file reader to the high score system to many useful tweaks and hours of advice and consultation, the library wouldn't be the same without him.

Eric Snider is our resident early adopter of the Mac version of Playground SDK, always willing to test out new Mac builds to make sure they function as expected. He brings years of game-writing wisdom to the design and direction of Playground.

Reggie Seagraves spent a year and a half developing the Mac version of Playground, and now David Parks is maintaining and updating the Mac code.

Oliver Marsh was on the front lines of early adoption of new Playground features, and he was invaluable in helping debug the library, as well as in consultation on the Playground SDK™ particle system.

Thanks also go to the QA department at PlayFirst, led by Christopher Dunn, for helping us iron out the problems in the Playground SDK™: Amy Belden, Peri Cumali, Valerie Gorchinski, Adam Gourdin, Devin Grayson, Bryan Kiechle, Cesar Lemus, Drew McKinney. Earlier QA efforts on Playground were led by Rebekah Cunningham.

Finally, thanks to the many external developers who have asked questions and provided feedback on Playground, either in person or on the developer site at https://developer.playfirst.com.

Part I

User's Guide

Chapter 1

Introduction

1.1 Welcome to the Playground SDK™!

The Playground SDK™ is designed to provide all of the core features you'll need to create a polished, successful downloadable game while handling many of the distractions that would otherwise slow you down. A game written in Playground will run on multiple platforms, including Windows, Windows-ActiveX, and Mac OS X 10.4. Playground is an object-oriented C++ library that relies on Lua for scripting support. Familiarity with Lua is helpful, but not a requirement, since most of the game is written in C++. Like C++ itself, Playground exposes both low-level and high-level functionality, giving you the ability to directly modify textures and map them on polygons at the lowest level, and a game-centric GUI/windowing system at the highest level.

While the Playground team has its own ideas as to how Playground should be used, we've tried not to overly restrict the number of development paradigms that make sense. For example, button messages can be set up to run entirely in Lua, or you can ignore Lua and simply process button messages in C++. Dialogs can easily be specified in a human-friendly format based on Lua, or they can be completely constructed by hand. Some developers are using their own custom libraries to do just about everything, handing only polygons to Playground to render, though in those cases it's often harder to guarantee cross-platform portability.

We're also constantly working on the Playground SDK™ to improve it; if you have a suggestion, idea, or complaint about the SDK, please let us know so that we can address it! Internally the code has been written for easy modification, so we're not afraid to add new features if they seem useful.

You can learn more about Playground and share your ideas with others at the developer web site and forum at https://developer.playfirst.com. Between active discussions, important announcements, and the latest released version of the SDK, it's a place any active Playground developer should visit frequently.

1.2 What's on the Playground?

1.2.1 Basic Features

Playground currently supports Windows XP and Windows Vista, both in stand-alone and ActiveX modes; and Mac OS X 10.4 or newer. Most Playground games will continue to run on Windows 98, Windows 98SE, Windows ME, Windows 2000, and Mac OS 10.39, though now that official support has been dropped, PlayFirst is no longer fixing corner-case bugs with semi-broken video card drivers.

Development is currently supported on Windows platforms using Visual Studio 2005 (8.0) or Visual C++ 2005, and on Mac OS X using Xcode.

The Playground SDK™ provides basic GUI features via its TWindow system, which supports message passing, Lua dialog/screen layout files, buttons, scroll bars, text entry, and image layering. The Lua script subsystem is integrated so as to be an optional component, though most developers choose to take advantage of it.

Playground supports the reading and display of JPG and PNG files, the latter with optional transparency. Animations are handled through an animated texture and sprite system. For finer grained control, the TRenderer::DrawVertices() API exposes full 3d triangle rendering with color, pre-lit, or pre-transformed-and-lit vertices, for those developers who have their own graphics library and who want to take advantage of the portability that Playground offers. Triangle rendering is also the most supported path to 3d game development at present. For sound playback, Playground supports the OGG/Vorbis sound format, a free format (no royalties required) that produces better quality sound than MP3 at similar bit rates.

Tools on the Playground

There are three GUI tools included in the download package on Windows. These tools are not yet available on Mac OS X.

The FirstStage resource editor can edit Lua files that describe dialog and screen layouts. The editor supports color-coding of Lua files, as well as auto-detecting custom window definitions. The editing is minimally destructive to your existing Lua files, so even if you want to modify a very complex script it shouldn't damage any of the fancy bits.

The FluidFX particle editor can display custom particle systems, again with a color-coded Lua editor. The FilmStrip animation editor provides basic animation functionality, but primarily allows you to add additional metadata to each frame of an animation. It's used extensively to add anchors or hot-spots to character frames.

1.2.2 Design Goals

The Playground SDK™ has several important design goals:

1. Portable to multiple platforms (OpenGL/DirectX, Mac/Windows, potentially others)
2. **Small**, to keep download size small.
3. Robust.
4. Complete.
5. **Separable**, so that features not needed by a client are not linked in.
6. **Readable**. The coding practices include readable design, judicious use of macros, comments on any code whose purpose isn't obvious, and class and member names that clearly indicate functionality.

1.2.3 Portability Concerns

There is a list of Coding Standards on the developer web site that, if you follow it, will help make your game completely and immediately portable to other platforms. The Coding Standards page is a living document—developers are a creative bunch, and from time to time they come up with new and interesting ways to write code that's not portable. As that happens, we add new standards. In addition, in order to ensure that the Playground SDK™ is portable, all platform-specific functionality you should need resides in the library. If there's anything missing, please ask for it!

All APIs exposed by the SDK hide platform-specific complexity behind an abstraction that specifies the intent of the request rather than the specific platform feature you need. For example, application configuration data

should be managed by the library with no reference in the API to application-specific information such as a path. The TFile file abstraction allows you to read and write files with no knowledge of the local file-system topology.

1.2.4 On Making a Library Robust

There's more to making a library robust than just expunging the bugs; the design needs to take into account how the library is likely to be used, making it as easy as possible for the user to write correct code. Any time we come across a point in SDK design where we feel the need to warn the user about a potential pitfall, we try to step back and reevaluate the design to see if there is a way we can eliminate the need for a warning by making an architecture change. Warnings remain in the documentation only in instances where we decide that the additional flexibility is worth the risk.

Designing a robust SDK also involves evaluating the entire process of writing a game, keeping track of the places that bugs tend to develop, and then handling as many of those problem domains as possible *in the library*. While we can't offload your game logic, we can make sure your game will run on any target architecture, that you have access to container classes that are well documented and thoroughly tested, and that the problems you do face concern the game you're writing and not the environment.

1.2.5 When is a Library Complete?

The Playground SDK™ will probably always be growing and evolving; how can we have completeness as a design goal? Conceptually, we intend Playground to be able to create any typical casual downloadable game. We hope to make available in Playground any feature that you would need to create any current game.

As we extend Playground in the future, improvements will fall into one of three categories. We'll be refining the core library, extending the API and making Playground more robust. We will augment the core library judiciously, when new features would have broad utility or require hardware support. And we will be creating more specialized features that exist in layers on top of the current library. Since these specialized components will be helpful to some, but not all, developers, the game developer will have the option whether to link them into their game. This keeps Playground's download footprint smaller for developers who don't need the optional functionality.

Chapter 2

Getting Started

2.1 How to Play on the Playground

The easiest way to get started with the Playground SDK™ is to start with the skeleton application, modifying the window name and splash screen sequence as appropriate for your product. Most of the substance of your application will live in, or be spawned from, a class derived from TWindow—the skeleton application creates several TWindow-derived classes.

The TWindow base class provides the functionality you would expect of a hierarchical window class: Add children, set a position, draw, send and respond to messages or events, and other standard supporting members and interfaces. Generic messages can be handled using TWindow::OnMessage(); other events trigger specific On∗() calls: TWindow::OnChar(), TWindow::OnMouseDown(), etc. Note that your window will only get keyboard events if the window currently has focus; see TWindowManager::SetFocus().

Custom windows are created from Lua resource files using dynamic creation; since C++ doesn't support named dynamic creation natively, we've added some support macros to enable that functionality. Here's an example of what that looks like:

C++
```
// The TGame window definition header
class TGame : public TWindow
{
   PFTYPEDEF_DC(TGame,TWindow) // Add the dynamic creation functions
...
}

// In the .cpp file:
PFTYPEIMPL_DC(TGame) // Define the TGame dynamic creation functions
```

Among other things, these macros will declare a ClassId() function that will return the unique class of the window type. To add a new window type, you register it by passing the ClassId() of the window to TWindowManager::AddWindowType. From main.cpp in the skeleton:

C++
```
   TWindowManager * wm = TWindowManager::GetInstance();
   wm->AddWindowType("GameWindow",TGame::ClassId());
```

This sequence allows you to specify the position and size of the game window in a Lua resource file, which (minimally) looks something like this:

Lua
```
   GameWindow
   {
      x=20, y=20, w=600, h=600
```

```
}
```

In this file you would also define any buttons that are children of that window, or alternately any background or status windows that are separate from the game window. The hierarchy can work however you like—a bunch of sibling windows that sit exactly next to each other, a strict hierarchy, or a hybrid. See Using TWindows for more information on deriving a class from TWindow.

A number of window types are defined for you by Playground, including buttons (TButton), text (TText), editable text (TTextEdit), and bitmaps (TImage). Look at the derived classes in the TWindow documentation to see a complete list.

Assets are loaded and managed by Playground in reference-counted containers. The T∗Ref classes are the containers, e.g., a TTextureRef holds a TTexture that you acquire from TTexture::Get(). See Game Assets for more information.

See TTexture::Draw and TTexture::DrawSprite for ways to draw your texture on the screen. These draw calls should only be called in your derived TWindow::Draw() function, in a TBegin2d block.

3d models and sounds are handled similarly to textures: A TSound is acquired with TSound::Get() and stored in a TSoundRef, and a TModel is acquired using TModel::Get() and is stored in a TModelRef. You can play a TSound with TSound::Play(). Drawing the TModel is done with TModel::Draw() after setting up the model's texture, matrices, and lighting. See the section in TRenderer on 3d-related functions for more information.

2.2 A First Example Program

2.2.1 The Main Program: Part One

In this section I will walk you through a skeleton application that demonstrates some basic functionality. First, a bit of sample code:

C++
```
void Main(TPlatform* pPlatform, const char* /*cmdLine*/ )
{
   InitializeGameState();

   // Set the application name
   pPlatform->SetWindowTitle(pPlatform->GetStringTable()->GetString("windowtitle"));

   TSettings::CreateSettings();
   TSettings::GetInstance()->InitGameToSettings();
```

This code is from main.cpp in the skeleton project. The functions are straightforward: The window title is the text that appears in the bar at the top of the window. The InitGameToSettings() call loads the application's saved settings and initializes the window to full screen or windowed mode, depending on the user's saved preferences.

C++
```
   pPlatform->SetCursor( TTexture::GetSimple("cursor/cursor"), TPoint(1,1) );

   pPlatform->SetCursor( TTexture::Get("cursor/thumb.png"), TPoint(12,2), true );

   TWindowManager * wm = TWindowManager::GetInstance();
   wm->AddWindowType("GameWindow",TGame::ClassId());
   wm->AddWindowType("MainMenu",TMainMenu::ClassId());
   wm->AddWindowType("OptionsWindow",TOptions::ClassId());
   wm->AddWindowType("HiscoreWindow",THiscore::ClassId());
   wm->AddWindowType("ChoosePlayerWindow",TChoosePlayer::ClassId());
   wm->AddWindowType("CreditsWindow",TCredits::ClassId());
   wm->AddWindowType("Swarm", TSwarm::ClassId());
   wm->AddWindowType("ChessPiece", TChessPiece::ClassId());
```

Here we set up a custom cursor for the application. Then, for convenience, we grab a pointer to the window manager. Then we register several window creation commands, which will allow us to easily specify our custom windows later.

C++
```
   // Start the Lua GUI script; this script will never exit
   // in a typical Playground application.
   wm->GetScript()->RunScript("scripts/mainloop.lua");
```

This last call to TScript::RunScript() causes our Lua main loop to begin.

2.2.2 What's a Window Creation Command?

The window creation commands mentioned above are simple classes that are overridden to create the custom game window. By "custom game window", we mean the window in which you plan to do all of the interesting stuff that makes your game fun—drawing sprites and/or 3d objects that dance around in response to user interaction. You can have more than one of these window classes in your application, and you can even specify that

several coexist on the same screen—but in order for the screen building code to know how to create your custom window, your window needs to have dynamic creation enabled.

First, in the class definition:

```C++
class TGame : public TWindow
{
   PFTYPEDEF_DC(TGame,TWindow)
```

Then, in the implementation file:

```C++
PFTYPEIMPL_DC(TGame);
```

Pretty simple. If something needs to happen after the window has been created, you can override either the TWindow::Init function, or the TWindow::PostChildrenInit function, depending on when it needs to happen.

To enable the custom window in the window scripts, you just need to call TWindowManager::AddWindowType() with the window name and class id:

```C++
   wm->AddWindowType("GameWindow",TGame::ClassId());
```

And then the window will be created in the script with a simple:

```Lua
      ...
      GameWindow
      {
            x=300,y=100,w=400,h=400
      },
      ...
```

2.2.3 Lua Main Loop? Custom Window Creation? What's this about?

The Playground SDK™ uses Lua as a way to achieve light cooperative multithreading, as well as for dialog/window layout. In the Lua main loop script, you can specify the order of windows you want to display, or a simple animation sequence, or pop up a modal dialog. When it's time for the script to pause (to wait for an animation or user input), you call a command that returns control to the C++ code. Some commands, like DisplaySplash(), implicitly return control and wait for a specified amount of time before continuing. Others, like DoModal(), pause to wait for a particular event, such as the closing of a window. Here is the main Lua GUI loop:

```Lua
-- Main game loop
function Main()

   DisplaySplash(
      "splash/playfirst_animated_logo.swf",
      "splash/playfirst_logo",4000
      );
   DisplaySplash("", "splash/distributor_logo",4000);

   -- Push the game selection screen
   while true do
      DoMainWindow("scripts/mainmenu.lua");
      -- DoMainWindow will exit only if there are NO windows pushed on the stack, so
      -- a PopModal()/PushModal() combination will not cause this to loop.
   end
end

-- Return a function to be executed in a thread
return Main
```

That last return statement is important: When you call TScript::RunScript, it just runs through the script once and returns. What we want to happen is for it to be able to run in a threaded manner. So TScript::RunScript watches for a return value from the script that it just ran, and if it finds one that's a function, it runs the function as a thread.

So what's happening here? First, a call to DisplaySplash() displays a splash screen for 4000ms (or until the user hits a key). A second call to DisplaySplash() brings up the second screen for 2000ms. Then an endless loop starts that consists entirely of bringing up the game selection screen. Why is it an endless loop? Because the game selection screen destroys itself when someone selects a game, and so when the user is done playing and the DoModal() subroutine finally exits, the game will need to create a new game selection screen.

2.2.4 The Main Program: Part Two

Back to main.cpp to show you the rest of the main loop:

C++
```
// The main C++ loop
TEvent event;
while(true)
{
   pPlatform->GetEvent(&event);
   if(event.mType == TEvent::kQuit)
   {
      break;
   }

   if (event.mType == TEvent::kClose)
   {
      TWindowManager::GetInstance()->GetScript()->RunScript("scripts/quitverify.lua");
   }

   if (event.mType == TEvent::kFullScreenToggle)
   {
      TSettings::GetInstance()->UpdateFullScreen();
   }

   // Pass the event to the Window manager for further processing
   TPlatform::GetInstance()->GetWindowManager()->HandleEvent(&event);
}
```

Here we have our "message pump," the place where top level application messages get processed. This code is pretty straightforward: Get an event, do something if we know how to react to it (i.e., if it's a kClose event), and pass the event on to TWindowManager::HandleEvent() for further processing. TWindowManagerHandleEvent() then propagates events appropriately; for example, it triggers mouse and keyboard events on appropriate windows.

2.3 Game Assets

Game assets are loaded and managed by the library. Internally, they are reference counted, but the reference count is updated entirely by C++ container classes. To use a bitmap texture loaded from disk, for instance, you would acquire it from the library using TTextureGet() and keep it in a TTextureRef instance. The TTextureRef handles the reference counting.

Here's some code to illustrate:

C++
```
class MyGame : public TWindow
{

...

   void LoadAssets()
   {
      // Load myimage.png or myimage.jpg
      mMyImage = TTexture::Get("images/myimage");
   }

   void Draw()
   {
      // Draw the image to the middle of an 800x600 screen
      mMyImage->DrawSprite(400,300);
   }

   TTextureRef mMyImage ;
}
```

The basic idea is that you keep around a persistent TTextureRef for each image you need. If you call TTextureGet more than once for the image you've already loaded, it will hand you a second reference to the same image. When the last TTextureRef to a particular image is destroyed, the image will be deallocated. Note that when you destroy a window and create a new window, as when you are switching between game modes, it doesn't actually delete the old window until you have created the new window—so any assets in common are simply referenced and won't need to be reloaded.

A TTexture can currently be loaded from a JPG or a PNG file, and will be auto-converted to a bit depth compatible with the current screen resolution. A TTextureRef acts like a pointer:

C++
```
   TTextureRef t = TTexture::Get("my.jpg");
   t->Draw( ... )
```

The example with TTexture above works similarly for TModel/TModelRef, if you plan to use 3d models.

2.4 The Game Window

The skeleton is built of a number of custom window classes derived from TWindow. In a real game, each of these windows could be used to display a different part of the game or user interface.

For example, in the Playground skeleton application the TSwarm class draws a swarm of butterflies. The butterflies are managed as sprites; here we create the sprites and assign them to a container:

C++
```
TSwarm::TSwarm() :
   mLastUpdate(0),
   mLevel(1)
{
   mSpriteHolder = TSprite::Create();
```

```
    // We're going to lazy-load all of our textures
    mAssetMap.SetAutoLoad(true);

    TAnimatedTextureRef at= mAssetMap.GetAnimatedTexture("anim/cardinal");

    if (at)
    {
        for (int i = 0 ;i<kNumSprites; ++i)
        {
            mSprites[i] = TAnimatedSprite::Create(i);
            mSprites[i]->SetTexture(at);
            mSprites[i]->Play();
            TDrawSpec drawSpec(
                TVec2( (float)((float)kBoundary+i*(float)(kWidth-kBoundary*2)/10.0F),
                       (float)((float)kBoundary+i*(float)(kHeight-kBoundary*2)/10.0F) ),
                1,0.8F
                );
            drawSpec.mMatrix.Scale( 0.5F + TPlatform::GetInstance()->Rand()%1000/1000.0F );

            mSprites[i]->GetDrawSpec()= drawSpec;
            mVelocity[i]= TVec2(0,0);

            mSpriteHolder->AddChild(mSprites[i]);
        }
    }

    mHitCount = 0;
}
```

The member mSpriteHolder is a sprite itself, though it doesn't have a texture assigned—rather it's only being used as a container. The same TAnimatedTexture is being assigned to each sprite, but since each TAnimatedSprite keeps track of its frames independently, and the script throws in some randomness, each butterfly flaps its wings independently.

Now that we have a bunch of butterfly sprites, let's draw them. Here's the start of the skeleton application's TSwarm::Draw:

C++

```
void TSwarm::Draw()
{
    TRenderer * r = TRenderer::GetInstance();
    TBegin2d begin2d ;

    mSpriteHolder->Draw();
```

That's it—now the sprites are drawn on the screen. The particles also get drawn here. We'll go into more detail about them later.

2.4.1 Let's Move!

Timed event processing is easy in Playground: You can either derive a class from TAnimTask and hand it to the current top modal window, or simply activate the internal TWindow animation timer. Moving the sprites around in the skeleton is handled in TSwarm::OnTaskAnimate—here is a simplified version for illustrative purposes:

C++

```
bool TSwarm::OnTaskAnimate()
{
    // Mark the screen as dirty so it will update
    TWindowManager::GetInstance()->InvalidateScreen();

    uint32_t now = GetWindowAnim()->GetClock()->GetTime();
    TReal elapsedTime = (float)(now - mLastUpdate)/1000.0f;
    mLastUpdate = now;

    for (int i=0; i<kNumSprites; ++i)
    {
```

```
        // Get the distance from this butterfly to the mouse cursor.
        TVec3 delta = TVec3(mLastMouse)-mSprites[i]->GetDrawSpec().mMatrix[2] ;

        // Add the mouse "pull" to the velocity.
        mVelocity[i] += delta * elapsedTime * mPull;

        // Move the sprite position by the velocity
        mSprites[i]->GetDrawSpec().mMatrix[2] += TVec3(mVelocity[i] * elapsedTime,0);
    }
    // Update the particle systems (pretend it's always 16ms)
    mParticles.Update(16);
    mParticles2.Update(16);
    return true;
}
```

Here TSwarm::OnTaskAnimate() iterates through the sprites and performs some simple math to move the sprites around. Then it calls TLuaParticleSystemUpdate() on each particle system to process their animation. Note that it's calling those systems with a constant value; this tends to keep the particle system looking more consistent.

To enable OnTaskAnimate(), you have to do one more thing:

C++
```
void TSwarm::Init(TWindowStyle &style)
{
    // Start the window animation to be called
    // (at most) every 15 milliseconds
    StartWindowAnimation( 15 );
}
```

The excerpt above is a simplified version of the code in the skeleton application (which also bounces the butterflies off the edge, damps the velocity, and tries to prevent the butterflies from clustering).

This is where the ongoing game logic typically takes place: In a routine that's called at a particular rate, so your game can always run at the same speed.

Be careful not to put more processing in a call like this than can comfortably be crunched in its given time slot. If this call were to take longer than 15ms, for instance, then it would be executed again on the very next update pass, and the game would slow down. You can prevent this and keep your game at a constant speed by increasing the delay step so that it's always longer than the call takes.

2.4.2 Drawing 3d Objects

Here's the start of a function in the skeleton responsible for rendering the chess piece window, TChessPiece::Draw.

C++
```
void TChessPiece::Draw()
{
    // Getting a copy of TPlatform for convenience
    TRenderer * r = TRenderer::GetInstance();
    TRect screenRect ;
    GetWindowRect(&screenRect);

    {
        TBegin3d begin3d;

        // Set up our perspective projection matrix
        r->SetPerspectiveProjection(0.1F,100.0F,PI/5.0f);

        // Set up our rendering texture
        r->SetTexture(mTexture);

        // Set up our light
        r->SetLight(0,&mLight);

        // Set up our material
        r->SetMaterial(&mMat);
```

So far it sets up a perspective projection matrix, a default texture, and a default material. Next it needs to set up the world matrix and a light:

C++
```
    TMat4 localMatrix ;
    localMatrix.Identity();

    // Build transform for yaw about board-y axis
    localMatrix.RotateY(mYaw);

    // Build transform for pitch about view-x axis

    TMat4 pitch, view;
    TRenderer::GetInstance()->GetViewMatrix(&view);
    pitch.RotateAxis((TVec3)view[0], PI/6);

    localMatrix = pitch*localMatrix ;

    localMatrix[3][0] = 0;
    localMatrix[3][1] = -0.75;
    localMatrix[3][2] = 3;

    mSpinLight.mDir.x = cos(mYaw*2) ;
    mSpinLight.mDir.y = sin(mYaw*3) ;
    mSpinLight.mDir.z = cos(mYaw*2) ;
    mSpinLight.mDir.Normalize();

    // Set up our light
    r->SetLight(1,&mSpinLight);
    r->SetWorldMatrix(&localMatrix);
```

We're spinning our chess piece around to mYaw radians and setting up a light at some other orbit for interesting reflections here. Next we do the actual drawing of the model:

C++
```
    if (mModel)
    {
        mModel->Draw();
    }
    begin3d.Done();
}
```

Note the TBegin3d: It's necessary to tell Playground whether you want it to be in 2d or 3d mode before you actually do any drawing. This allows us to optimize certain aspects of set-up, and makes this necessary overhead more explicit so that a game programmer knows that switching between these modes is expensive.

Finally, we're simply drawing a box around the window. Not brain surgery. Note TBegin2d, which is analogous to the TBegin3d above. TBegin2d and TBegin3d are helper classes that automatically release the state on close of scope.

That's all there is to it. So where did mModel come from? It was initialized in the TChessPiece constructor along with mTexture:

C++
```
mModel = mAssetMap.GetModel("mesh/king");

// Also note that, for consistency (and to guarantee that we don't accidentally
// forget to preload a file) we load all assets via the asset map.
mTexture = mAssetMap.GetTexture("mesh/white");
```

And there you have it!

2.5 Dealing With User Input

A TWindow-derived class will receive user input via the On∗() class of functions. If a user clicks in a window, it will receive an TWindow::OnMouseDown() - TWindow::OnMouseUp() pair. When a mouse moves over a window, it will receive TWindow::OnMouseMove(). Messages are passed from child to parent if the child doesn't handle them.

Similarly, the window with the keyboard focus gets TWindow::OnChar when a key is hit. For finer-grained control, TWindow::OnKeyDown and TWindow::OnKeyUp are fired when a key is pressed and released. Messages sent from child windows, like the "button pressed" message sent by TButton, can be fielded by TWindow::OnMessage().

2.6 Why Modal Windows Are Your Friend

In the Playground SDK™ framework, when a message or event arrives at the window hierarchy, it typically starts at a window and works its way up through parents looking for a handler. It always stops looking if it encounters a TModalWindow, however; modal windows act as boundaries to the game context, and input never travels past them on the stack. In fact, the TWindowManager maintains a stack of just TModalWindows, and you can query the top modal window using TWindowManager::GetTopModalWindow().

When Playground starts up, it pushes a special modal window called TScreen on the top of the window stack. This window must never be popped from the stack, or the game will exit.

The standard paradigm when switching between game modes is to pop the current modal window off the stack and push your new window; alternately, you can use the Lua function SwapToModal() if you are not creating custom-derived TModalWindows.

If you need to bring up a game-pausing event ("Are you sure you want to quit?"), you can push another modal window onto the stack, and pop it when you're finished. If you have a game with sub-games, you can push a sub-game window onto the stack. When a new modal window is on the stack, the previous window gets *no* messages, its TTask events stop firing, and its clock stops. Messages and TTask events resume when the child modal window closes. Note that in order to take advantage of its clock you need to explicitly assign the clock to the class that needs it, e.g., TAnimTask or TAnimatedSprite.

Modal window have no technical limit as to their depth; however it's probably not wise to push more than two to three levels, just from a user interface perspective.

2.7 How to use Playground SDK™ features.

2.7.1 Using TSprites

The TSprite class can be used to help manage the display of game objects. TSprite objects are stored in reference-counted variables. You can use them individually, or you can create a hierarchy of them.

A TSprite is an encapsulation of the following items:

- A texture to render.
- A layer value.
- A TDrawSpec for position/orientation/tint/scaling/alpha.
- A list of children.

When you create a TSprite you give it a layer, which is used only for sorting relative to its siblings and parent: Higher numbered layers appear in front of lower numbered layers among siblings, and negative layers appear behind the parent TSprite.

Each TSprite has an associated TTexture, and an inherent TDrawSpec. The latter is used to position and orient the sprite relative to its parent, and in general to determine how to draw the texture. See the documentation on TDrawSpec for more details.

A related object, the TAnimatedSprite, is identical to the TSprite with two exceptions: It's designed to be able to handle TAnimatedTexture particularly well, and it contains a TScript for animating the TAnimatedTexture. A normal TSprite can have a TAnimatedTexture assigned to it, since TAnimatedTexture inherits the TTexture interface, and therefore in object-oriented terms is-a TTexture. However, since TAnimatedTexture objects can be reused, any animation state has to be kept with each instance—in this case with the TAnimatedSprite object.

Why a TAnimatedSprite and a TAnimatedTexture?

A TAnimatedTexture is simply a list of frames. A TAnimatedSprite has a script that plays back the frames. So there's no concept of "Play" on TAnimatedTexture, but there is on TAnimatedSprite.

2.7.2 Using TDrawSpec

A TDrawSpec includes the following information:

- A position.
- An orientation/scaling matrix
- A logical center to render the texture relative to.
- A tint value (that's local to this sprite).
- An alpha value (that's inherited by children).

There are two overloaded versions of TTexture::DrawSprite. The first takes a set of simplified parameters, and the second takes a TDrawSpec for increased control over how the image is drawn. TDrawSpec comes with a convenience constructor that will allow you to set most common values in-place. After it's constructed, you can modify it with much more sophisticated requests if you need to.

There were three reasons for the decision to migrate to a TDrawSpec-style interface: One, TTexture::DrawSprite was really getting over-overloaded with confusingly similar parameter lists. Two, the parameter lists were getting so long that it became quite annoying to set the ones later in the list if you just needed to set one or two. And third, the DrawSprite parameters simply weren't flexible enough to do full inheritance of rotation/scale matrices, which is what we wanted for the sprite system.

2.7.3 Using TWindows

TWindow and its descendants are for GUI elements. The typical Playground SDK™ model is one where you create your game in a TWindow, possibly layered with TText windows for score and other messages, and TImage windows for interface elements. The TWindowDraw() call is where you do the heavy lifting, actually drawing elements of your game. Depending on your preferences, you can manage that part yourself entirely, drawing textures exactly where you want them, or you can use a support class like TSprite to help manage the display of your game objects for you.

TWindows clip what they and their children draw to their rectangle. For a GUI element, that's a feature: You can draw a block of text that's clipped to the window, or you can draw an image that's zoomed so that its boundaries clip to the window. TWindows will receive mouse messages when the mouse is within their bounds (note that mouse messages are also clipped to parent windows), and TWindows can also receive the keyboard focus and thereby receive key events (see TWindowManager::SetFocus()).

Many functions return a TWindow*, and you will often need a derived class. The function TWindow::GetCast<>() will get you that pointer with type safety (see Type Information and Casting for details). Windows can optionally draw their contents (TWindow::Draw()), contain other child windows, and respond to events or messages. Window updates and messages are handled by the TWindowManager class, which is available from the singleton accessor TWindowManagerGetInstance().

To assist in debugging window hierarchies, in debug builds on Windows pressing "F2" will dump the current hierarchy to the debug log.

2.7.4 Modifying TTexture Objects

When you've modified a texture in the game, you need to set up a texture-refresh-listener to monitor whether the texture contents have been lost.

When DirectX loses a texture, it needs to be rerendered. This is handled by the library for textures loaded from files that haven't been modified, but any modified by the game via TRendererCopyPixels, TTextGraphicDraw, or TRendererBeginRenderTarget need to be redrawn if they are lost.

The process is simple enough. Create a TTask-derived class and override DoTask:

C++
```
class MyTextureListener: public TTask
{
public:
   virtual bool DoTask()
   {
      RenderMyPrivateTextures();
   }
};
```

Then call:

C++
```
TPlatform::GetInstance()->AdoptTextureRefreshListener( new MyTextureListener() );
```

Now any time that your texture needs to be redrawn (because of lost surfaces), your function RenderMyPrivate-Textures() will be called.

2.7.5 Using TTextGraphic Objects

You can use TTextGraphic to render to any texture, not only a render target. In this example, assume the following class definition:

```
TTextureRef mTextTexture ;
```
C++

And then the code:

```
mTextTexture = TTexture::Create(512, 512, true);

TColor color(1,1,1,0); // White background that's transparent (alpha 0)
TPlatform::GetInstance()->FillRect(0,0,512,512,&amp;color,mTextTexture);

TTextGraphic * tg = TTextGraphic::Create(
  "<outline␣color=\\"000000\\"␣size=\\"3\\">SWEET␣PAUSE</outline>",
  512,512,TTextGraphic::kHAlignCenter,"fonts/arial.mvec", 60, TColor(1, 1, 1, 1) );
tg->SetNoBlend();
tg->Draw( TRect(0,0,512,512), 1, 0, 1, mTextTexture );
tg->Destroy();
```
C++

When this is complete, mTextTexture ends up with SWEET PAUSE rendered in Arial font and a black outline in a texture that's transparent except for the text itself which is opaque. The TRect in TTextGraphic::Draw can be used to position the text: Just bump down the top Y coordinate for each successive render. The text size is determined in the TTextGraphic::Create call—I arbitrarily chose 60 pixels tall, but feel free to customize to the appropriate height.

2.8 What is Lua?

Lua is a compact, fast, flexible, and extendable scripting language that is included as part of the Playground SDK™. Its syntax has a few quirks that we put up with because we love it so much. You can read more about Lua at http://www.lua.org. We're currently using version 5.0x of Lua in Playground, though we're investigating an upgrade to the latest, 5.1.

2.9 Lua Makes it Easy

Using a scripting language to write the basic control flow of a game is a very powerful technique. Take, for example the following code:

Lua

```
-- Main game loop
function Main()

   DisplaySplash(
      "splash/playfirst_animated_logo.swf",
      "splash/playfirst_logo",4000
      );
   DisplaySplash("", "splash/distributor_logo",4000);

   -- Push the game selection screen
   while true do
      DoMainWindow("scripts/mainmenu.lua");
      -- DoMainWindow will exit only if there are NO windows pushed on the stack, so
      -- a PopModal()/PushModal() combination will not cause this to loop.
   end
end

-- Return a function to be executed in a thread
return Main
```

Here the sequence is clear: Display one splash screen, then another, then enter main options loop. There's no jumping around from one place to another to see what happens next; it's right there in front of you. If you want to change the sequence, it's a simple matter of adding or moving a line of Lua code.

Or this code excerpt from a window definition:

Lua

```
   Button
   {
      x=40, y=40,       -- Position of the button
      label="Start␣Game", -- Button text
      command=function()
                SwapToModal("gamescreen.lua"); -- switch to the game screen
             end;
   }
```

Curly-braces in Lua define a table—they're not used for scope, though they may appear to in the previous example. The above syntax is a shortcut for calling the function "Button" with a single table as the only parameter: Button({ x=40 ... }).

In this table we're setting a few button characteristics, including its position, label, and a command to execute when it's pressed. The Lua command "function" actually defines a function right there in the Button definition. It's an anonymous function, in that it doesn't have a name of its own, but that's OK, because in Lua functions are first-class data: You can assign them to variables, return them from functions, or, as in the case above, add them to a table entry.

If the above example used DoModal() instead of SwapToModal(), it would have simply brought a window up on top of the current window—a standard "Modal" dialog, possibly asking for user input. And that window could have its own actions embedded in its buttons.

2.10 What About Speed?

There are those who worry about the speed of a scripting language to write a game. And they're right: It's not as fast as raw C++ code. But there are plenty of places in your code where you don't need the most speed possible: Responding to a button click and deciding what splash screen to display next are two examples from above. The general 80/20 rule usually holds: more than 80% of the execution time is spent in 20% of the code. Some go further and say that 90% of the execution time is spent in less than 10% of the code. So don't write that part in Lua.

Lua interfaces to C and C++ very easily: It was written to be an embedded scripting language (originally a configuration language). We've included template wrapper functions that allow you to painlessly drop a member function of a class right into Lua. And it's actually not that slow: It compiles to an interpreted byte code. It benchmarks favorably against Perl, Ruby, PHP, JavaScript, and Python. And it adds less than 40k of object code to the executable.

2.11 What are the pitfalls?

So what's the catch? Well, we're using the Lua "coroutine" feature to run the main window thread—most of the time the main loop thread is waiting for a message to tell it what to do next. We can use the Lua interpreter to run simple functions from our C code at that point, but that code can't itself yield, because that particular interpreter has already yielded. You can call a function using TLuaFunction::Call() even if there's a currently yielded function on the Lua stack, but that function can't itself yield. Nor can the function that you call directly or indirectly attempt to resume the original yielded function, which by that point is buried under both the Lua and system stacks.

The Playground SDK™ handles the first problem internally for the GUI script (the one you get from TWindowManager::GetScript), so you don't need to worry about it, since it will take any code you run and inject it into the currently paused script, if it's in a state where it can do this.

The second problem, trying to yield across a C call, comes up most frequently when C/C++ code that the Lua script calls attempts to inject its own function or pass a message into the running Lua stack. Avoiding this is easy, though: Any calls that you expose to Lua should be restricted to manipulating C++ data entirely, which could include, for instance, adding a message to a queue that will later be injected in a Lua script or processed by your game.

In fact, the TMessage system works great for this; just create a message type for your game with whatever payload you want, and trigger complex events in your game using those messages rather than just calling the necessary code directly. By "complex", I mean events that themselves may rely on the Lua interpreter to do something that might require a Yield or Resume.

Chapter 3

Playground Fundamentals

3.1 Dynamic Casting

C++ has support for runtime type information (RTTI), which includes the ability to safely dynamically cast from one type to another, but there are several shortcomings to the standard RTTI support. For one, there is no dynamic creation support. For another, when classes are contained in smart pointers, a straight dynamic_cast won't work, because as far as C++ is concerned the objects have no relationship to each other. Finally, there is no reflection in C++ classes–you can't ask a class what type it is and get a predictable response.

The Playground SDK™ RTTI system supports dynamic creation of classes, safe casting between smart pointers to related classes, and identifying classes by name.

3.2 RTTI in TWindow

The TWindow hierarchy doesn't require the use of the smart-pointer features, but does use dynamic creation. Every TWindow-derived type includes a static member function ClassId(). TButtonClassId() will return the class identifier for the TButton class, for instance.

To find out if a TWindow* is a TButton, use the following pattern:

C++
```
TWindow * window = ....

if (window->IsKindOf( TButton::ClassId() ))
{
   // Our window is a button
}
```

Most of the time you'll probably need to get a TButton directly. That's where GetCast() comes in–it works like dynamic_cast to cast our TWindow* to an appropriate type, returning a NULL pointer if the cast is inappropriate:

C++
```
TWindow * window = ....

TButton * button = window->GetCast<TButton>();

if (button)
{
   // Our window is a button, and now we have a button to play with
}
```

3.3 How to Share and Play Well With Others

When writing a game, some objects have clear owners: In the case of a button, it's fine to allow the enclosing window to own it and clean it up when it's destroyed. But there are some situations where an object or resource has no clear owner. The texture to draw on the button is one simple example: You wouldn't want the button to destroy the texture when it's deleted for fear another button on the screen is still using it.

One solution to this problem is to use reference-counted pointers: Smart containers that keep track of how many references are currently held to them. The above example would keep the button texture around until the last button window holding the texture container was deleted. Additional tricks can allow you to keep a list of currently loaded textures, so that when each button requests its texture they all get shared instances of the same texture.

You can use reference-counted pointers to manage your own game objects as well, but it's important to thoroughly understand the implications of using smart pointers before using them. It's not really that difficult, but there are a few common traps that you need to be aware of in order to avoid stumbling into them.

3.4 Using shared_ptr (TClassRef) Classes

When you include pf/ref.h, you gain access to the shared_ptr template class. Several Playground classes use shared_ptr internally, and provide convenience typedefs of the form TFooRef, indicating that it's a reference-counted container for a Foo. A few common examples include:

- TTextureRef
- TModelRef
- TSoundRef
- TSpriteRef

The full set of Playground shared_ptr typedefs is defined in pf/forward.h. In each case where Playground intends you to use shared_ptrs, the associated class has a static member factory named Get() (or prefixed by Get) when the class manages sharing of instances of the asset, or Create() for when you need a new, unique item. See TTexture::Get() or TTexture::Create() for two examples.

Part of the reason to have factories like this instead of allowing you to new objects and assign them to shared_ptrs is to help avoid common reference-counted-pointer errors by never encouraging you to operate on the raw pointers. If you always hold one of these objects in an appropriate Ref (e.g., TTextureRef) container for as long as you intend to use it, passing it from place to place as a Ref, and never converting it to a pointer at all, then it's really hard to make a mistake that would cause the reference counting mechanism to fail.

3.5 Managing Assets

Another common strategy to handle assets is to have an asset pool that a particular portion of your game can draw assets from. Sometimes that portion is the entire game, where you can load up assets at the beginning during a loading screen. Sometimes you want to load some assets at the start of a level, and then release them when the next level starts.

Playground also supports this pattern: Create a TAssetMap for your level, or for the whole game, or both, and use it to hold references to all of your game assets. When you're done with that portion of the game, destroy the

TAssetMap and all of those assets will be released. Here's the best part: If you "load" the next section of your game before you delete the old TAssetMap, any assets in common will simply be referenced and not reloaded!

You can use a TAssetMap in either a strict mode where it requires that any asset you request through it must already be referenced, or in a more forgiving mode where you can request an asset it doesn't have yet and it will both load and hold a reference to the asset. See TAssetMap::SetAutoLoad() for details.

3.6 Common Mistakes

So what are the common mistakes you need to be aware of? Well, here's one:

C++
```
// This is BAD! Don't do it!
TTexture* someTexture = TTexture::Get("foo.png").get();
```

Assuming the texture isn't referenced elsewhere, the code above will assign a dangling pointer to someTexture. By the time the assignment occurs, the texture object will have been destroyed.

C++
```
// Correct
TTextureRef someTexture = TTexture::Get("foo.png");
```

Keeping it in a TTextureRef ensures that you keep a reference to it when the temporary on the right hand side of the assignment operator is destroyed.

C++
```
// This is also BAD! Don't do it either!
void Foo( TTexture* texture )
{
    TTextureRef textureRef(texture); // ERROR!
}

TTextureRef someTexture = TTexture::Get("foo.png");
Foo( someTexture.get() ); // BAD BAD BAD
```

In the example above, when someTexture leaves scope, it will delete the texture. Funny thing is, when textureRef leaves scope, it will *also* delete the texture. This is not considered a good thing. If you have a good memory debug tool in place, you'll find this right away because it will alert you right when it happens. Without malloc debugging active, some time later, in an unrelated part of your game, it will probably crash when allocating or freeing memory. And where it crashes may change between debug and release builds, because the memory allocators work differently in debug and release. In other words, this is a bug you never want to lay the groundwork for by using raw pointers when you should be using shared_ptrs.

Again, simply passing the reference around as a reference solves the problem:

C++
```
// Correct
void Foo( TTextureRef texture )
{
    ...
}

TTextureRef someTexture = TTexture::Get("foo.png");
Foo( someTexture ); // No pointer necessary
```

3.7 Your Very Own shared_ptr

The shared_ptr template class can be used to manage game assets as well. If you're using shared_ptrs in your own classes, though, you can end up with a cyclic reference that never gets freed: A holds a reference to B and vice versa. When all references to A and B are freed, you still have these two referring to each other. All is not

lost, however. One or both of these references can be a "weak" reference. If A owns B, but B needs a reference back to A, then you can make the pointer in A a shared_ptr and the one in B a weak_ptr. That way when the last reference to A is released, it will destroy A. If another object still holds a reference to B, it won't be destroyed when A releases its reference–but B's weak reference to A will magically NULL itself, so that B knows A has been destroyed.

3.8 Implementation Details

Technical details for the current implementation can be found at http://boost.org/libs/smart_ptr/smart_ptr.htm

Feel free to look at the documentation there to understand more about the philosophy and design decisions behind smart pointers. However, we are using a subset of the full Boost libraries, so don't expect all of the features they describe to be available. We do use weak_ptr internally to handle our Get() functions (so we know if an asset has already been loaded, and to correctly return the existing shared_ptr), but we reserve the right to reimplement the shared_ptr if we decide there's a compelling reason.

3.9 About the Playground SDK™ Hiscore System

Playground includes a hiscore management system that can be used to keep track of your players' highest scores in multiple categories in your game. For games that are published by PlayFirst, the system also includes the ability to connect with PlayFirst servers and share high scores globally.

3.10 Initialize The System

For PlayFirst published builds, you'll need to set the PlayFirst-provided encryption key using TPlatform::SetConfig().

C++
```
// This line typically appears in TSettings/settings.cpp
TPlatform::GetInstance()->SetConfig(TPlatform::kEncryptionKey,ENCRYPTION_KEY);

TPfHiscores *pHiscores = new TPfHiscores();
```

The game name on Windows is extracted from the program's resource: look in the project's .RC file for the ProductName to set the name.

3.11 Connecting to a debug server

Until your game is officially launched, the PlayFirst hiscore server will not know how to accept submissions from your game. Therefore, to test your implementation of the hiscore system, you need to use a local server.

To connect to the fake local server:

1. Place the pfservlet_stub.dll in the same folder you are loading the pfhiscore.dll from (or in the same folder as your .exe if you are using a static lib). The system will detect the existence of this dll, and if it can find it, it will use a local server instead of connecting to one over the Internet.

2. Make sure that you are correctly setting up the TPlatform::SetConfig settings for kPFGameHandle, kPFGameModeName, and kEncryptionKey. This is done in PlaygroundInit() in the Playground Skeleton sample application.

3. You are now set up to use the hiscore system without connecting to a server. Note that this will create a file called "serverdata.txt" file in the same folder as the dll that will store information between executions so you can test building up long lists of scores. Also, note that this local server does not contain all the functionality as the real online server (i.e. it does not contain the profanity name filter and it will also not correctly filter the scores by "daily", "weekly", etc.) but it does have enough functionality for you to fully test your game.

4. The pfservlet_stub.dll comes with 2 user accounts created for testing PlayFirst user submissions. They are "testname" with password "testpass" and "testname2" with "testpass2".

5. Refer to the values recorded in "serverdata.txt" to verify that your score and medals submissions were successful.

3.12 Set Properties

As the user plays the game, you need to set various properties with the hiscore system. If the user changes player names/profiles, call:

```
pHiscores->SetProperty(TPfHiscores::ePlayerName, PLAYERNAME);
```
C++

If the user changes game modes, call:

```
pHiscores->SetProperty(TPfHiscores::eGameMode, GAMEMODE);
```
C++

If the user changes languages, call:

```
pHiscores->SetProperty(TPfHiscores::eLanguage, LANGUAGE);
```
C++

3.13 Logging Scores

At the end of a game (or when quitting a story-mode game where you want to log a score), you need to log the players score so it will be eligible for a hiscore. In addition to logging a score, you should also log game specific data (i.e. "5" for someone who has reached level 5, etc.). Please do not put text inside the game specific data, as it makes it difficult to localize. The game has the chance to parse out this game specific data later on and put it in visible form. The server currently supports up to 60 characters of data.

```
// Log a score of 500 points, replace existing score by this player,
// and pass in the GAMEDATA string defined elsewhere...
pHiscores->KeepScore(500, true, GAMEDATA);
```
C++

3.14 Logging Medals

When a user earns a medal, you can log the medal for submission to the server.

```
pHiscores->KeepMedal(PFGAMEMEDALNAMES[0], TPfHiscores::eMedalType_Game);
```
C++

3.15 Viewing local scores

On the local score screen, you can view all the logged local scores. To view the local scores for the current game mode:

```
pHiscores->GetScoreCount(true);
for (int i = 0; i < numScores; i++)
{
   int rank;
   char name[16];
   bool anon;
   int score;
   char gameData[64];

   if (pHiscores->GetScore(true, i, &rank, name, 16, &anon, &score, gameData, 64))
   {
```
C++

```
        char outputTest[512];
        sprintf(outputTest, "%d) %s (%d) %d\n", rank, name, anon, score);
        DEBUG_WRITE((outputTest));
    }
}
```

In the local scores screen, you can also change the game mode by setting the game mode property, and then recalling the code above for the new game mode.

3.16 Figuring out what score the user can submit

Once in the local score screen, you can figure out if the current player has a score eligible to submit:

C++
```
if (pHiscores->GetUserBestScore(TPfHiscores::eLocalEligible,&score, &rank, gameData, 32))
{
    // display what the users current eligible score is to submit
}
else
{
    // user has no eligible scores to submit
}
```

3.17 Submitting a score and medals

To submit a score, you either submit an anonymous score or a user/password official score. They both use the same call. If you submit data with a password, it will also submit any medals that have been logged. You can control whether you are submitting medals and scores, or just one individually. It is recommended that you submit everything at once if the UI allows it - but if the UI is specific to just medals you may need to only submit medals. Once you submit the score and medals, you need to poll for the status of the submission to check for any errors:

C++
```
pHiscores->SubmitData(USERNAME, PASSWORD, REMEMBERSETTINGS, TPfHiscores::kSubmitAll);

TPfHiscores::EStatus status;
char errorMsg[256];
bool qualified
status = mpHiscores->GetServerRequestStatus(errorMsg, 256, &qualified);
while (status == TPfHiscores::ePending)
{
    status = mpHiscores->GetServerRequestStatus(errorMsg, 256, NULL);
}
if (status == TPfHiscores::eError)
{
    DEBUG_WRITE((errorMsg));
}
else if (status == TPfHiscores::eSuccess)
{
if (qualified)
    {
        // inform user of qualified score
    }
    else
    {
        // inform user that score did not qualify
    }

}
```

3.18 Switching to the global score view

The first thing you need to do when switching to the global score view is get the available categories from the server. The categories are things like "last 24 hours" or "all time records".

C++
```
pHiscores->RequestCategoryInformation();

TPfHiscores::EStatus status;
char errorMsg[256];
status = mpHiscores->GetServerRequestStatus(errorMsg, 256, NULL);
while (status == TPfHiscores::ePending)
{
   status = mpHiscores->GetServerRequestStatus(errorMsg, 256, NULL);
}

if (status == TPfHiscores::eError)
{
   DEBUG_WRITE((errorMsg));
}
else if (status == TPfHiscores::eSuccess)
{
int numTables = pHiscores->GetCategoryCount();

      for (int i = 0; i < numTables; i++)
      {
         char name[16];
         if (pHiscores->GetCategoryName(i, name, 16))
         {
            char outputTest[512];
            sprintf(outputTest, "TABLE:␣%s\n", name);
            DEBUG_WRITE((outputTest));
         }
      }
}
```

Now that you've obtained the categories, you can request scores for a specific category. Note that before you call RequestScores() you must have set a proper game mode with SetProperty() and you must have received the category information with RequestCategoryInformation().

C++
```
pHiscores->RequestScores(CATEGORYNUM);

TPfHiscores::EStatus status;
char errorMsg[256];
status = mpHiscores->GetServerRequestStatus(errorMsg, 256, NULL);
while (status == TPfHiscores::ePending)
{
   status = mpHiscores->GetServerRequestStatus(errorMsg, 256, NULL);
}
if (status == TPfHiscores::eError)
{
   DEBUG_WRITE((errorMsg));
}
else if (status == TPfHiscores::eSuccess)
{
int numScores = pHiscores->GetScoreCount(false);

   for (int i = 0; i < numScores; i++)
   {
      int rank;
      char name[16];
      bool anon;
      int score;
      char gamedata[64];
```

```
        if (pHiscores->GetScore(false, i, &rank, name, 16, &anon, &score, gameData, 64))
        {
            char outputTest[512];
            sprintf(outputTest, "%d␣%s␣(%d)␣%d␣%s\n", rank, name, anon, score, gameData);
            DEBUG_WRITE((outputTest));
        }
    }
}
```

Finally, you can determine the user's current ranking in the currently fetched table from:

C++

```
int score;
int rank;
char gameData[64];

if (mpHiscores->GetUserBestScore(TPfHiscores::eGlobalBest, &score, &rank, gameData, 64))
{
    // user is ranked
}
else
{
    // user is not ranked
}
```

3.19 Hiscore FAQ

Question: How can I connect to the PlayFirst hiscore server to test my implementation?

Answer: You should be able to fully test your hiscore implementation against the pfservlet_stub. If it works with this stub (and you have configured TPlatform::SetConfig just like the Playground Skeleton does in PlaygroundInit()), it will work when your game goes "live".

Question: Hiscores is not working. Can I get any information about why it is failing?

Answer: You should check your logfile. Depending on the error, some information may be displayed in the logfile which may make it obvious why your hiscore submission is failing.

Question: QA is telling me that hiscores fail on their server, but they work fine on my stub. What is the difference?

Answer: Make sure that you are using the PlayFirst provided key.h file, which correctly specifies the exact names for:

- The game handle (in Windows, this is set in the resource file)
- The encryption key
- The game modes
- The medal names

Question: I can view global hiscores, but all the submissions fail. What is wrong?

Answer: If you can successfully view hiscores but not submit them, then your encryption key is incorrect.

Question: When I submit a hiscore with medals in it, I get an error in my logfile about invalid XML. What does that mean?

Answer: (This question involves an older version of the API that is being deprecated - use KeepScore/-KeepMedals/SubmitData to avoid this problem). XML must be "well-formed". A common mistake is to forget to close the XML tag with a forward slash. Note that using XML is no longer necessary for submitting medals - use the KeepMedals and SubmitData calls and this will be handled automatically.

Question: I am seeing weird results when I submit a score and medals. Sometimes I see the "Did Not Qualify" response, but then the score gets recorded anyway. Other times the system lets me submit the same score over and over again. What is going on?

Answer: (This question involves an older version of the API that is being deprecated - use KeepScore/-KeepMedals/SubmitData to avoid this problem). This is most likely because you are issuing several server requests but only looking for one response. Each call to the server must then poll GetServerRequestStatus() before the next call should take place. One specific situation that can cause this behavior is if you submit a score, then submit medals, and then poll the server. The response you get back from the server will be the response to the medals submission only. Therefore, the response to the score submission is lost completely. The correct thing to do in this situation would be to submit a score, then poll the server. Once a response is received, a medal submission could continue, polling for a response to that as well.

Question: Why are there 2 different ways of submitting medals?

Answer: (This question involves an older version of the API that is being deprecated - use KeepScore/KeepMedals/SubmitData to avoid this problem). Medals can be associated with a score (by using the serverData parameter in TPfHiscores::LogScore()), or they can be submitted independently by using TPfHiscores::SubmitMedals(). In some games, you can only earn a new medal at the same time you earn a new hiscore (i.e. you can only earn a medal by beating a new level, which also means your total score goes up). In this case, by associating a medal with a score in TPfHiscores::LogScore(), you don't have to worry about how to submit medals - they will be submitted when the user submits their score. In other games, you can earn a medal without earning a new score (i.e. by replaying a previous level and completing a series of events that unlocks a new medal). In this case, the user needs to submit their medals separately from their hiscore, because they may not have any hiscores to submit. In this case, more UI needs to be added to the game to inform the user when they have a medal to submit, since you cannot depend on the existence of a submittable hiscore in order to submit medals. Note that TPfHiscores::submitScore() and TPfHiscores::SubmitMedals() should never be called immediately one after the other - you always need to poll TPfHiscores::GetServerRequestStatus() after each hiscore server call.

Question: How do I know if I should be running in full hiscore mode, anonymous hiscore mode, or local hiscore mode?

Answer: You can use the following to determine which mode to run in:

- TPlatform::GetInstance()->IsEnabled(TPlatform::kHiscoreAnonymous) - use this to detect anonymous mode
- TPlatform::GetInstance()->IsEnabled(TPlatform::kHiscoreLocalOnly) - use this to detect local mode

Question: What is the difference between the full, anonymous, and local hiscore modes?

Answer: This is described in the Production Guidelines document, and your PlayFirst producer can provide you with the full details about this. Additionally, the Playground Skeleton application demonstrates each of these 3 modes, which you can test by changing the settings.xml file. The basic differences between the modes are:

- In "full" mode the user can use a PlayFirst account and password to submit their hiscore. Scores associated with a PlayFirst account have a special icon displayed next to them. There is a text description about the PlayFirst global hiscores system, and a description of how to register for an account.
- In "anonymous" mode users can submit hiscores only with an "anonymous" account. There is no mention of PlayFirst in the hiscore system, though some text about a Privacy Policy may be displayed. Please contact your producer for the exact text.
- In "local" mode there is no way to submit a hiscore to the global hiscore system. The user can only see scores that they have earned on their local computer. There is no mention of the PlayFirst hiscore system in this mode.

3.20 Writing Debug Messages

The Playground SDK™ defines a number of useful macros for debugging your code:

- DEBUG_WRITE() Writes messages to the debug log in debug builds.
- ERROR_WRITE() Writes messages to the debug log in debug and release builds.
- TRACE_WRITE() Writes the current line, function, and file to the debug log (in debug builds only).
- ASSERT() Breaks in debug builds and prints ASSERT condition to the debug log; removed in release build.
- VERIFY() Breaks in debug builds and prints VERIFY condition to the debug log; just prints the condition to the log in release build. Guaranteed NOT to be removed in release build. Useful for evaluating a function return value where the function has side effects (it does something important).

On Windows, all printing to the log is mirrored in the debugger output window in debug build. In release builds, the debug log is capped at 100k, but in debug build it is not limited in size.

3.21 Dynamic Debugging

Playground defines several hot keys to help you debug your application. Most are available only during debug builds.

- **Alt-F1** Bring up a display that includes current game vital statistics, including Playground build version, frames-per-second, allocated memory, and allocated texture memory. This works even in release builds.
- **F2** Dump a window hierarchy to the debug log.
- **Shift-F2** Draw a yellow rectangle on the border of the window the mouse is currently rolling over.
- **F3** Dump all currently allocated assets to the debug log.

3.22 Game Version Numbers

The version number for your game is read from the file version.h into your resource file (e.g., skeleton.rc, via version.rch).

If you're developing a game for PlayFirst, then when it is submitted to PlayFirst to be built, **your copy of version.h is replaced by a machine generated version for each new build.**

Considering this is true, if you're working with PlayFirst to publish your Plaground SDK game, you should be sure to:

- Only edit right hand side values in version.h
- Not add additional definitions to version.h
- Not change your game's resource file (e.g., skeleton.rc or version.rch) to have a hard-coded version number, or otherwise prevent it from reading and using the version number in version.h

You may change the version information defined in version.h for your own purposes, as long as you keep to the same format. However, please remember that for games released through PlayFirst, this file is overwritten by PlayFirst's release engineering process. This means that the build numbers that your game has for your own builds will differ from the official builds that PlayFirst generates. This is by design. If you violate the guidelines here, you will be asked to change your code back, so that your game source code will not break the PlayFirst release engineering process.

3.23 Flat File Basics

Playground supports grouping most of your assets into a single flat file. The support is completely transparent to your game—when you open up a file in the assets folder, it will behave identically regardless of whether it exists as a file on the disk or embedded in a flat file. One difference your users might notice, though, is that your game will install more quickly with a flat file—and that it will load more quickly when you run it. On Windows we use memory-mapped file IO to access the file, meaning it's particularly fast to open and read small files.

A flat file also provides a small amount of obfuscation to your assets—it's no longer possible to just open up the .png files in your game by double-clicking them. In fact, PlayFirst does not provide a way to open up a flat file, though of course a serious developer could come up with a way to produce such a tool.

Flat files in Playground are given the .pfp extension—and files with that extension will be auto-loaded by Playground from the assets root, user: root, or common: root folders, so be sure not to name your files with a .pfp extension unless they really are Playground pfpack files.

3.24 Creating Your Own Flat File

The way you compress your folder is to run the included pfpack.exe tool on the folder:

```
pfpack assets assets.pfp
```

This will create a file, assets.pfp, that you can drop into your assets folder instead of all the files you currently include there. But there's a catch: Two specific files, strings.xml and settings.xml, can't be included in assets.pfp. In addition, Flash can't read any .swf file from assets.pfp. So you'll need to create a folder with those files (strings.xml, settings.xml, ∗.swf) excluded, pack that folder, then reintroduce those files to the combined folder. Additionally, strings.xml and settings.xml need to be read in before the flat files are initialized, so those files need to be excluded.

As an example, a functional flat-file assets folder might look like this:

```
assets/assets.pfp
assets/strings.xml
assets/settings.xml
assets/splash/playfirst.swf
```

3.25 Appending Files

You can also use .pfp files to add new downloaded modules to your game. When Playground starts up, it scans the assets folder, the user: folder, and the common: folder for any files that end in .pfp. It then loads them in alphabetical order, with later files appending to overriding earlier files.

This means that if you have a file gamesettings/levels.xml that appears in three .pfp files, the instance of gamesettings/levels.xml that appears in the latest .pfp file alphabetically will be the one that the game sees. The folders are scanned in the order:

1. assets
2. user:
3. common:

So your game assets.pfp can be overridden by any .pfp files you download into user: or common:. Finally, any files that physically exist in the assets folder will override any files with the same name in any of the pfp files that are found in the assets folder; .pfp files found in the user: or common: folder will override filenames present in the assets folder.

Chapter 4

Beta Versions

4.1 How to Create Limited Builds for Beta Testers

When it's time to send out your game to hordes of beta testers, the last thing you want to do is send them your entire high-profile moneymaker: Even with copy-protection wrapped around it, you'll only slow down the crackers, who will have an unprotected version of your game up as soon as a few days or a week after you've distributed it.

This may also be true of the final build once you post it on your site, but in the case of particularly popular games, if a user really wants to play it sooner than later, they'll be willing to download a cracked version if the real build isn't available.

Playground supports the development of limited builds using the same source and assets as your final build. It does this by parsing one of two text files that indicate files and folders that the internal file system should ignore; in the build you send out, if those files aren't included in the package, then that's content that they can't pirate. Using the text files you can simulate that content being removed without creating a new entire development environment, or a second copy of your assets tree.

To enable this feature, you need a settings.xml file if you don't have one already. It should look like:

```
<?xml version="1.0" ?>
<settings>
     <firstpeek>1</firstpeek>
</settings>
```

...or if you already have one, then just add the <firstpeek> line above.

If it isn't obvious, the number should be '1' for FirstPeek and '0' for the normal release build. This file goes in the assets folder next to strings.xml.

In the folder that CONTAINS assets you need two more files:

final.txt firstpeek.txt

These files list asset filenames or folders that should be EXCLUDED from each view. This way you can exclude files or folders from FirstPeek using firstpeek.txt, and you can exclude FirstPeek-only assets from the final build.

The files are each read in and fed through strtok with " \t\n\r" as the tokens: Any whitespace will separate filenames. The filename needs to match EXACTLY the filename on disk. Be careful with case sensitivity here– Playground doesn't normalize the case, so your case will need to match exactly.

There are no wildcards. You must name a complete path to a file:

path/to/file.jpg

...or the path to a folder:

path/to/folder

If you want to exclude a list of files, you'll need to collect those filenames in a text file formatted as you see above. The paths are all relative to the assets folder.

At runtime, to tell the difference between FirstPeek and normal builds, you can use: TPlatform::GetInstance()->IsEnabled(TPlatform::kFirstPeek). That way you can make runtime game-play decisions–though that's not completely safe as a protection method on its own, since a cracker would only need to change one value to disable FirstPeek mode.

On the Mac, in order to make this work, you'll need to add a build step that copies final.txt and firstpeek.txt to the application bundle so the game can read them. On the build machine, these files will need to be deleted, since we don't want them in the package.

4.2 Tracking Game Metrics

The TGameState singleton allows you to easily accumulate game metrics for your FirstPeek build. Using the SetState() and GetState() calls as outlined below, you can automatically accumulate state information about how the user is playing your game, and it will be written to the file user:metrics.txt.

The metrics are only actually calculated and written to the output file when the module **firstpeek.pfmod** is placed in the game's assets folder. PlayFirst will automatically include firstpeek.pfmod with FirstPeek builds, and the PlayFirst game wrapper will automatically collect that file from the client's computer and forward it back to PlayFirst for analysis.

Below are examples of how to set up and collect the metrics for your FirstPeek builds.

4.2.1 The Interface and States

All of the examples below use the TGameState interface to talk to the FirstPeek DLL. To get that interface, you simply use the following code block.

```
TGameState * gs = TGameState::GetInstance();
```
C++

The C++ examples below assume a TGameState pointer called gs. On the other hand, the Lua examples don't need one! From within the Lua GUI script, SetState() and GetState() are available directly:

```
SetState("MyState", "1" );
```
Lua

The above line sets a state variable named "MyState" to the value "1". Later you can retrieve that value:

```
state = GetState("MyState");
```
Lua

Note that you can call TGameState::SetState() regardless of whether it's a FirstPeek version. In a normal release of your game, the call will simply do (almost) nothing.

Sometimes you just want to record that someone has done something, like the fact that they started story mode. In that case, you can just call:

```
gs->SetState("StartedStoryMode",1);
```
C++

If you have a state that keeps track of how many times a user does something, you can prefix it with the ++ operator:

```
gs->SetState("++LemonadeUsed"); // increment the LemonadeUsed state
```
C++

A state starts out with value 0, so you can use the above line to count how many times the player uses a lemonade power-up.

Later you will be able to write the state value to the FirstPeek log file, either immediately on request, at a particular time delay, or at the end of the game, as I will describe below.

4.2.2 FirstPeek Timers

A common need in a FirstPeek game is to have timers that record, for example, how long the player spends in different sections of the game–including how long they spend playing an entire game session. To achieve this, at the start of the game, you'd typically write the following line to start a session timer.

```C++
gs->SetState("SessionTime", 1) ; // start our session timer
```

Setting the state of any variable to "1" starts a timer with that name, and setting it to "0" pauses that timer. The overhead is minimal, so don't worry about setting a variable to "1" even if you don't intend to use it as a timer.

If you want to start and stop a timer that keeps track of the time spent in the main menu, you could use the following code.

```Lua
-- in the mainmenu.lua file
SetState("MainMenuScreenTime", 1) ;

-- in each button command that exits
-- the main menu, call this function
function LeaveMainMenu()
   SetState("MainMenuScreenTime", 0) ;
end

...

Button
{
   name = "startthegame";
   command = function() LeaveMainMenu(); StartGame(); end;
}

...
```

Sometimes you'll want to restart a timer so you can use it again. In that case you can set it to the special value "restart".

```Lua
gs->SetState("LevelPlayTime","restart");
```

It's OK to set a timer to "restart" even if it's never been used before. You can stop the timer normally by setting it to "0".

4.2.3 Writing an Immediate Status Line

To write out stats during gameplay, such as at the end of a level, you start by naming the status line block, and then adding status arguments to it one at a time until you're done, at which point you end the status line.

A simple example:

```Lua
gs->SetState("begin-status","LevelCompleteStory"); // Name the status line
gs->SetState("add-state","LemonadeUsed");     // Add the current state of LemonadeUsed
gs->SetState("end-status","LevelCompleteStory"); // End the status line
```

A few things here. First, the block is defined between a "begin-status" and an "end-status" command. Every value you add between those commands will be added to the status line.

The "add-state" command takes the name of a state that you've defined earlier, and it writes out the current value of that state. You need to have previously set that state to some value, or it will just write out an empty string.

There are two other commands you can use to add entries to the log line:

Lua
```
gs->SetState("begin-status","LevelCompleteStory"); // Name the status line
gs->SetState("add-state","LemonadeUsed");     // Add the current state of LemonadeUsed
gs->SetState("add-timer","LevelPlayTime");    // Add the timer value LevelPlayTime
gs->SetState("add-value",score);              // Add a constant value
gs->SetState("end-status","LevelCompleteStory"); // End the status line
```

The "add-timer" command writes out the value of a state variable that's being used as a timer. It needs a different command than "add-state" because its state value is simply "1" or "0"–and that's not what you want. To reiterate, when you SetState("LevelPlayTime",1), it effectively starts an internal timer associated with the key "LevelPlay-Time", and later when you call SetState("LevelPlayTime",0), that timer is paused.

Finally, "add-value" adds the constant you give it in the second parameter directly. There's no lookup performed–in the example above, it simply writes the score (presumably an integer) to the log file line directly.

4.2.4 Write Status Line on Exit

A frequent need is to write a status line when the game exits. You could use the above "begin-status"/"end-status" calls to write the line as your game quits, but we've made it even easier. Instead you can set up a block at the start of the game that it holds onto until the game is being quit, and on exit it will write out this status line with the values of the states at that time.

A simple example:

Lua
```
gs->SetState("begin-session","SessionData");
gs->SetState("add-timer","SessionTime");      // Add the session timer to the output
gs->SetState("add-timer","StoryModeSessionTime"); // Add a story mode timer
gs->SetState("add-timer","MainMenuScreenTime"); // How much time spent on the main menu?
gs->SetState("add-timer","ComicScreenTime");  // And how much on the comics?
gs->SetState("add-timer","HelpScreenTime");   // And how much on the help screen?
gs->SetState("add-state","LemonadeUsed");     // Add the current state of LemonadeUsed
gs->SetState("end-session","SessionData");    // Finish recording SessionData output
```

When the game exits, it will automatically write a SessionData line to the FirstPeek data file with the final results of the states and timers listed. You can also use "add-value", but keep in mind that you're sending it a constant, and that the constant will therefore be defined at the start of the game rather than later on. If you want to add a variable, then you should set it using SetState() instead–that's what it's for!

Keep in mind that you should only call "begin-session"/"end-session" once during your game. If you call it more than once, the results are undefined.

4.3 Writing Status on Elapsed Time

You can create status lines that write their output at specific elapsed times. This is similar to "begin-session", but instead of at the end of a game session, it writes its data after a certain elapsed time.

The elapsed time calculation is cumulative across sessions: Once they've played the game for 15 minutes TOTAL, the 15-minute timer will fire and write the data. It doesn't matter if they play it in one sitting or play 5 3-minute games–it keeps track of the cumulative elapsed time.

For example:

Lua
```
gs->SetState("begin-elapsed(15)","FifteenMinutesUsed"); // Start a 15-minute elapsed time record
gs->SetState("add-state","LemonadeUsed");         // Have they used that lemonade yet?
gs->SetState("end-elapsed(15)","FifteenMinutesUsed"); // End the record.
```

Obviously the "15" above is the number of minutes of cumulative gameplay to compare with. If the game has been played for more than 15 cumulative minutes on launch, this log file line will be silently ignored.

4.4 FirstPeek Module Reference

These command are used as the key in TGameState::SetState() as described above, with the meaning of the value varying based on the command, as described below.

4.4.1 begin-status/end-status

Parameters:

value The name of the output line.

Begin and end a status line. The status line is written to metrics.txt immediately on receiving the end-status command.

The line is written in CSV format, with the name, date, time, and cumulative elapsed time as the first four parameters. Additional parameters can be added with the various add-∗ keys, below.

4.4.2 begin-session/end-session

Parameters:

value The name of the output line.

Begin and end a session line. The session block is written to metrics.txt when the game exits, using the status variables as they are defined at that point.

See begin-status/end-status above for details status line format.

4.4.3 begin-elapsed(time)/end-elapsed(time)

Parameters:

time The number of elapsed minutes to use.
value The name of the output line.

Begin and end an elapsed-time data block. The elapsed time block is written to metrics.txt when the game has been played (cumulatively across all sessions) the number of minutes given as time.

See begin-status/end-status above for details status line format.

4.4.4 add-timer

Parameters:

value Name of the key to time.

Add the elapsed time that a key has been set to a non-zero value as a parameter to the current status line block. The value is queried when the status line block is actually written.

Must be within a status line block (begin-status/end-status, begin-session/end-session, begin-elapsed(time)/end-elapsed(time)) to have any effect; an error otherwise.

4.4.5 add-timer

Parameters:

value Name of the key to query.

Add the value of the given key as a parameter to the current status line block. The value is queried when the status line block is actually written.

Must be within a status line block (begin-status/end-status, begin-session/end-session, begin-elapsed(time)/end-elapsed(time)) to have any effect; an error otherwise.

4.4.6 add-timer

Parameters:

value Constant value to add.

Add the given value (literal) as a parameter to the current status line block. This value is set immediately, and not queried when the block is actually written.

Must be within a status line block (begin-status/end-status, begin-session/end-session, begin-elapsed(time)/end-elapsed(time)) to have any effect; an error otherwise.

Chapter 5

Lua

5.1 About Lua

Lua is an embedded scripting language that's very fast and has a very small footprint. It's also very extensible, and very easy to link to C or C++ functions.

A few Lua quirks you should know to make reading Lua code eaiser:

- Line comments in Lua start with double-dash $(--)$
- Block comments are between $--$[[and $--$]].
- Lua's syntax is free-form–whitespace is irrelevant in most circumstances.
- Use of semi-colon to terminate a statement is optional, though it can enhance readability.
- Single or double quotes can be used to frame a string. [[and]] will frame a multi-lined string.

You can read more about Lua syntax at http://www.lua.org . We're linking with the base library, the string library, and the table manipulation library at present. In debug builds of the library, the debug library is also linked in. TLuaParticleSystem additionally links with the math library.

5.2 Using Lua to Script Your Game

The examples that are covered in Why Use Lua? are entirely concerned with scripting the overall game state flow and in-game events. But what if you want to use Lua to script more interesting behaviors in your game? The Playground SDK™ fully supports arbitrary Lua threads running at whatever rates you specify, which can control anything from game AI to animations. The set-up requirements have been minimized wherever we can, but you should still expect to put some effort into connecting your C++ code to Lua. The benefits are great, and the effort will be worth it.

At the most basic level, you need to create a TScript-derived class with your added functionality. Here is an example:

Lua

```
class MyClass
{
    public:
        int MyMemberFunction( str param1, int param2 );
}
```

```
...

TScript * script = new TScript ;
MyClass * myClass = new MyClass ;

ScriptRegisterMemberDirect(
    script, "LuaNameForMyMemberFunction", myClass, MyClass::MyMemberFunction );

script->DoLuaString("␣a␣=␣LuaNameForMyMemberFunction(␣'test␣param',␣4␣);␣");
```

See how the constructor binds the C++ member functions directly to Lua functions. See RegisterMemberDirect() for a list of the valid parameter types you can use. For an explanation of TMessage, see the advanced topic Sending Custom Application Messages.

5.2.1 When to Yield

The Lua interpreter supports cooperative multithreading. In practice, this means that a script that is going to persist needs to explicitly or implicitly yield periodically. The most basic yield command is Yield, which returns control to the C++ code immeidately.

But that doesn't take into account when your script may want to be run next; TScript derives from TAnimTask so that it can resume running on a schedule. You can register your TScript as a task, either at the TModalWindow level or at a global level in TPlatform. Conceptually the difference is that TModalWindow tasks are only executed when the TModalWindow is the *top* modal window; another modal window will pause the tasks of any windows beneath it. Note that the "Adopt" symantics (TPlatform::AdoptTask) means that you are relinquishing control of the script. It won't be deleted when it's done executing, because it never flags itself as "done" animating, but if the context that owns it is destroyed (i.e., the TPlatform or TModalWindow), it will be deleted at that point.

5.2.2 Running Your Script

Discussion of adding the TScript to either the current top modal or the system timer.

5.2.3 Subclassing Existing Window Types

To subclass from an existing window type (for example, if you want to have a button that has some custom behavior in it), all you need to do is be sure that your derived class calls the base class for any virtual functions it overrides. For example, if you derive from TButton and create a PostChildrenInit() call in your derived class, you'll need to call TButtonPostChildrenInit() to ensure that the base class gets properly initialized.

5.2.4 Lua Dialog Creation Internals

Here is a brief description of the syntax of the internals of dialog-description Lua files. The first thing to note is that instead of calling functions with ordered parameter lists, which is the norm in Lua, we're calling with the table (Lua's map/hash/dictionary) syntax: Function{}. This allows us to have both positional and named parameters.

At the most basic, in the Lua script you need to use the function MakeDialog:

```
MakeDialog                                                    Lua
{
  ...
}
```

Inside the body of MakeDialog is a list of (potentially nested) window creation commands. A typical window will start with an image background:

Lua
```
MakeDialog
{
    Bitmap
    {
        image="background.png"
    }
}
```

This is pretty simple so far. The Bitmap command will create a TImage window scaled to the size of background.png. There are ways to override this, but we'll get to that later.

Now say that you also want a button:

Lua
```
PropFont = {
    "fonts/prop.mvec",
    15,
    Color(0,0,91,255)
};

BasicButtonGraphics =
{
    "controls/lozengeup.png",
    "controls/lozengedown.png",
    "controls/lozengeover.png"
};

MakeDialog
{
    Bitmap
    {
        image="background.png", -- Note that this is a list, and needs commas

        -- The optional mask tag says to use "backgroundmask.png" as the
        -- alpha channel for the "background.png" image.
        mask="backgroundmask.png",

        Button
        {
            x=kCenter, y=-40, -- See below for coordinate options
            name="button3",   -- The name used to look up the button
            font=PropFont,    -- The button text font
            graphics=BasicButtonGraphics, -- The button graphics
            label="foo",         -- The default button label
            sound="click.ogg",          -- Sound to play when button is clicked
            rolloversound="over.ogg", -- Sound to play when button is rolled over
            scale=0.7     -- scales the size of the button
        }
    }
}
```

There are several new things here worth mentioning. We've added a font for the button text, and graphic images for the various button states. Note that the graphics entry is a table with three items: The first is the "up" image, the second is the "down" image, and the third is the roll-over image. You can skip the third image if the down and roll-over image should be the same.

The font itself is a table with three entries: The font filename (relative to the assets directory), the font height in pixels, and the font color.

The x and y coordinates can be simple coordinates from the upper left corner of their window, but there are also other convenient options:

- Positive x and y are normal positions.
- A constant, kCenter, added to x or y will position the window relative to the center (based on width or height) in x or y directions. For example, kCenter+0 centers the window, and kCenter+3 centers the window 3 pixels to the right of the center of the parent.

- You can subtract a value from kMax to specify the window origin from the right or bottom of the parent window. x=kMax-20 means to put the window twenty pixels from the right edge, for instance.
- You can set alignment values using the align tag and one (or two, added together) of the window alignment constants. See Text and Window Alignment.. The alignment values change what edge or corner you're specifying: If you say align=kHAlignRight+kVAlignBottom, then x and y will specify the bottom right corner of the window. Mixing alignment values with the kMax constant is supported, but kCenter overrides any alignment setting for that dimension.
- Negative x and y are relative to the opposite edges. *This feature is deprectated and will be removed from Playground 4.1. Instead use offsets from kMax, or alignment settings. When using "align" tag, this feature is disabled.*

Now let's add a text field:

Lua
```
...
      Text
      {
         x=0,y=kMax-80,w=kMax,h=-kMax, -- The entire dialog, minus the bottom 80 pixels
         font=PropFont,              -- The text font
         label="This␣is␣text"
      },
```

The x and y coordinates work the same for the text field as above. We've chosen not to name this field, which is okay since it's just static text (it will get a window id of -1). But we have two new fields: w and h for width and height. Similar to x and y, a positive number is a number of pixels. In addition:

- A constant, kMax, specifies that the rectangle should grow to fill available space.
- A positive width or height grows from the right of or down from the corresponding position, while a negative width or height grows the opposite direction.
- The kMax constant can be negated as well, filling to the left or up from the x,y coordinate.

5.2.5 Styles

Repeating information for each and every widget is tiresome and mistake prone. To address this, there is a concept of a current style. It's recommended that most features in your project are defined using styles.

Lua
```
PropFont = {
   "fonts/prop.mvec",
   15,
   Color(0,0,91,255)
};

BasicButtonGraphics = {
   "controls/lozengeup.png",
   "controls/lozengedown.png",
   "controls/lozengeover.png"
};

DefaultStyle = {
   font=PropFont,
   graphics=BasicButtonGraphics,
   x=0,y=0,w=kMax,h=kMax
}

BottomEdgeButton = {
   parent = DefaultStyle,
   y=-40
}
```

```
BodyText= {
   parent = DefaultStyle,
   y=-80,h=-kMax
}

MakeDialog
{
   SetStyle(DefaultStyle),
   Bitmap
   {
      image="background.png",
      SetStyle(BodyText),
      Text
      {
         label="This␣is␣text␣that␣goes␣in␣the␣body"
      },
      SetStyle(BottomEdgeButton),
      Button
      {
         x=kCenter,       -- The y coordinate is set already.
         name="button3",  -- The name used to look up the button
         label="foo"      -- The default button label
         sound="click.ogg" -- Sound to play when button is clicked
         rolloversound="over.ogg", -- Sound to play when button is rolled over
      },
      TextEdit            -- creates a user editable text edit field
      {
         password=true,   -- "*" instead of letters to hide passwords
         length=26        -- max characters user can type into edit field
         ignore="#!@"     -- ignore '#','!', and '@' if typed by the user
      };
   }
}
```

There are several new things in this example: The styles are also Lua tables, with one notable extra field: "parent" refers to the parent of a style. When MakeDialog is looking for a property, it looks first in the table passed to the creator function (e.g., Bitmap{}), and then in the current style, and then in the parent style(s).

The current style only exists in the "scope" of the list it's defined in.

So we start by setting the DefaultStyle, because that's just a good habit to have. Next we set a style for the BodyText, and the Text{} creator function suddenly gets much simpler. Finally we set the style for the Button, and it also ends up simpler.

There are two notable advantages to styles: One is that you can apply one style to many widgets, and then when you change the style it affects all of the widgets. Another is that you can then move these styles to another file entirely and apply them to all of the dialogs in your entire project. See the Lua command "require", which includes another Lua file in the current file.

If you want buttons that are radio or toggle style, you can specify those as follows:

```
...                                                                    Lua
   Button
   {
      type=kToggle,
      graphics= {
         "offImage.png",
         "onImage.png",
         "offRolloverImage.png",
         "onRolloverImage.png"
      }
   },

   -- Mark the start of a new radio group. Every radio button from this
   -- tag to the next BeginGroup() will be in the same group.
   BeginGroup(),
```

```
Button
{
    type=kRadio,
    graphics= {
        "offImage.png",
        "onImage.png",
        "offRolloverImage.png",
        "onRolloverImage.png"
    }
}
```

Lua is a language that's designed to be easy to edit and to have flexible data representations.

As such, it makes a great place to put data that you need for your game. It's easy enough to understand that your game designers/level editors can easily tweak parameters for various levels. If you have an in-game level editor, it might make more sense to have it save out XML, since then it can read and write exactly the same file, as well as retain human readability.

On the other hand, if your game needs scripted actions to occur, scripting those actions in Lua is very attractive. But how much Lua is appropriate? More the point, how much Lua can you use and still expect to be supported by PlayFirst? The answer, approximately, is a lot—as long as the usage falls into a few patterns:

1. Retrieving data. You can have as many Lua files containing game data as you want.
2. Game-play subroutines that execute and quit. This is the classic scripted action. When some event in the game occurs, then you call a Lua function that determines what happens.
3. Simple animation behaviors. Animations in the Playground SDK™ are handled via Lua scripts, and the animations can call very simple C++ functions or set Lua variables at specific points.

The main game thread shouldn't live in Lua—yet. There isn't enough support for our engineers to track what's going on if there's a problem that we need to diagnose. When would we be fixing problems in your game? PlayFirst frequently assigns engineers to fix compatibility problems that crop up on a system in our lab—if you can't reproduce the problem on your system, how can you fix it?

In addition, we're missing one of the most important game development tools in Lua at the moment: A profiler. Lua is fast for a scripting language, but still much slower than C++. If a lot of your game is written in Lua, you'll likely eventually need to optimize it. The first step in optimization should always be to profile, and we have no way to profile your Lua code yet.

One more problematic issue occurs when you end up with a stack that looks like: C->Lua->C->Lua and the Lua code tries to yield control back to C. The problem appears when your Lua code calls a C function that then calls more Lua code that tries to yield. Since pushing a modal dialog within Lua has an implicit yield, this happens more often than you'd expect. This won't happen in Lua code that doesn't use Lua threads, but the Lua UI thread in the Playground SDK™ depends on Lua threads to function, and the logical place for your game code to exist would be in the same engine as the UI thread—and that would require you to yield execution from time to time for the UI thread to function. We're investigating Lua Coco to see if it will work with Playground on the PC and Mac, which, if it works as advertised, will eliminate this issue. Again, stay tuned.

So, while we encourage you to use Lua for the three patterns listed above, we ask that the majority of the game logic exist in C++ until our Lua tools improve.

5.3 How do I get Lua data in my C++ code?

Say you'd like to use Lua scripts to do some of your configuration, or you have some Lua code that generates a table that you'd like to access in C++. How do you get to the data? Well, one way is to use the Lua API described at http://www.lua.org, and there are some things that you will have to do that way. But there's an eaiser way: the Playground SDK™ has a number of simple wrappers that allow you direct access to Lua data. The wrappers are very light in code impact, but not heavily optimized.

The Lua object wrapper base class is TLuaObjectWrapper, and it pulls the top object off of the Lua stack and wraps it in a C++ structure. During the life of the TLuaObjectWrapper object, the wrapped object is guaranteed not to be deleted by the Lua garbage collection or reference counter. The TLuaObjectWrapper class can only perform basic operations on a Lua object: Push it on the Lua stack, query whether it is a string or number, or attempt to convert it to a string or number. The derived class TLuaTable is intended to manage and read a Lua table.

Chapter 6

Particle System

6.1 Particles in a Scripting Language?

6.1.1 A Particle "Shader Language"

The Playground SDK™ particle system uses Lua as its definition language. On learning this, the first question that comes to any developer's mind who has been around the block more than once is often, "Particles need to be driven by very fast code—how can one be driven by a scripting language?" Lua is possibly the best data-definition language in common use today, and it's designed to be embedded. It's quite fast, but it's still not as fast as some tightly written C++ code. So yes, if the particles were *simulated* in Lua, it would be nowhere near as fast as a particle system written in C++. And with particles, you really want them to be as fast as possible, so you can support as many particles as possible.

So what if you just use Lua to define the structures you need for the particle system? That way you get to lean on the user-friendliness and interactivity of a scripting language, while still taking advantage of the speed of C++ by having the actual particle processing done in the faster language.

That's what we've done: The Lua code defines a number of operations that are then performed in a tight C++ loop on each particle. In some sense this is similar to writing a vertex or pixel shader, only for particles. Basing the language in Lua allows us to take advantage of the existing scripting engine in Playground.

6.1.2 Playing With Particles

The easiest way to get a feel for a new system is to see it in action. Let's start with a very simple particle script:

Lua

```
----------------------------
-- Initialization phase

-- Set the particle texture
SetTexture("star");

-- Initialize the raw particles--do this last in the Initialization Phase
SetNumParticles(1);

----------------------------
-- Action phase
```

```
-- Create an initial particle
CreateParticles( 1 );
```

So far our particle system isn't very interesting. It creates one particle in the middle of the screen that just sits there. Let's make it fall:

Lua

```
----------------------------
-- Initialization phase

-- Allocate a velocity particle property as a Vec2
pVelocity = Allocate(2) ;

-- Set the particle texture
SetTexture("star");

-- Initialize the raw particles--do this last in the Initialization Phase
SetNumParticles(1);

----------------------------
-- Action Phase

-- Animate the velocity
pVelocity:Anim( pVelocity + fTimeScale(Vec2(0,800)) );

-- Animate the position
pPosition:Anim( pPosition + fTimeScale(pVelocity) );

-- Create an initial particle
CreateParticles( 1 );
```

There are two new things here. First, we're allocating a velocity. While we can move the particle without giving it a velocity (by adding a constant value to the pPosition property), it's more interesting if each particle can keep track of a velocity value.

Second, we add two animation lines. These add actions to the particles that occur in declaration order to every particle on every TLuaParticleSystem::Update() call. Let's break down one of these lines more carefully:

Lua

```
-- Animate the velocity
pVelocity:Anim( pVelocity + fTimeScale(Vec2(0,800)) );
```

Translated into English, this says: On each step of the animation, assign the property pVelocity the value (pVelocity + fTimeScale(Vec2(0,800)). The part with fTimeScale(Vec2(0,800)) says to scale the value Vec2(0,800) by the current elapsed time, so you can give it a number in units/second and it will scale the value appropriately.

Similarly, the next line adds a time-scaled pVelocity to pPosition. What is a pPosition? I'm glad you asked...

6.1.3 Particle Properties

Each particle in a system has a number of programmable properties. Each property is a set of one to four floating point values. The default particle type is a T2dParticle, which has a number of innate properties, listed below along with their Lua names and sizes:

- A 2d Position (pPosition, TReal[2])
- A 2d Up-vector where 0,-1 make a particle "upright" (pUp, TReal[2])
- A scale (pScale, TReal[1])
- An RGBA color (pColor, TReal[4])
- A frame (pFrame, TReal[1]) for use with animated particles.

The actual texture isn't a particle property in the same sense, in that all of the particles in a system share the same texture. The texture can be a TAnimatedTexture to allow for animated particles, but it has to be one texture shared by all the particles due to the nature of the rendering.

The basic idea is that the particle script sets down how to initialize these properties for each particle as it's created, and then how to animate these properties on each frame.

6.1.4 The Particle Lua File

The order of operations in a typical particle configuration file is:

1. Create any custom particle properties.
2. Set the particle texture and blend mode.
3. Set the number of particles in the system.
4. Set up the particle initializer functions.
5. Set up the particle animation functions.
6. Define an Update function that creates particles.

The order of the first three operations above is strict; the rest are just by convention. In addition to the innate properties of a 2d particle, you can create custom properties and use them however you like. A common property to create would be a particle velocity, for instance, to give each particle its own motion. Another common property is an age: Some animated effects rely on a particle's age to calculate, so that a particle can, for example, fade from one color to another.

But not all particle systems need velocity (a set of twinkling stars wouldn't move, for example), and simple particles don't need an age, so you can add just the extra values you need. When you do need a custom particle property, you can name it however you like—this is Lua, after all.

6.1.5 Operations in the Particle Script

When you create a particle property like pVelocity, you get an object instance that you've seen do a few things. First, it can add particle initialization functions using the :Init() member function. Second, it can add particle animation functions using the :Anim() member function. The third thing that it does is a bit more subtle: When you put it in an equation, it returns an object that references the particle property that it represents.

Let's bring back the :Anim example:

```lua
pVelocity:Anim( pVelocity + fTimeScale(Vec2(0,800)) );
```
Lua

The equation $pVelocity = pVelocity + fTimeScale(Vec2(0,800))$ is then executed every frame. This is a standard equation that applies gravity to every particle.

6.1.6 A More Interesting Example

Here's an example from the sample application that includes a data source:

```lua
-- ------------------------------------------------------------
-- Initialization phase

-- Define our custom particle properties
pVelocity = Allocate(2) ; -- Allocate a Vec2 velocity member
pAge = Allocate(1) ; -- Allocate a TReal age member
pSpin = Allocate(1); -- Allocate a TReal spin member
pSpinSpeed = Allocate(1); -- Allocate a TReal spin member
```
Lua

Here we create more particle members; first the familiar pVelocity, and then three others. pAge keeps track of the age of a particle, pSpin keeps track of an initial orientation, and pSpinSpeed keeps track of a particle's rotational velocity. These elements are each unique for each particle in this particle system.

Lua
```
-- dLocus is a data source. The new FluidFX 2.0 defines dLocus for us,
-- but in case you want to load this particle system in a context
-- where you have no data source defined, this code will give dLocus
-- a default value.
if not dLocus then
   dLocus= Vec2(0,0) ;
end
```

In the sample application we add a data source to the particle system and call it dLocus. We still want the sample to work in FluidFX, however, since we don't want to give up the ability to edit the particle system in real time, so we add these lines of code to conditionally define dLocus to just be Vec2(0,0) when it hasn't been defined already.

Lua
```
-- Set the particle texture
SetTexture("star");

-- Set the blend mode
SetBlendMode(kBlendNormal);

-- Set the size of the particle pool.
SetNumParticles(400);
```

Here we're setting the blend mode, though it's not necessary since kBlendNormal is the default. See the section on TRenderer::SetBlendMode() for more information on blend modes. We're also setting the number of available particles in the pool to a higher number so that we don't run out too quickly.

Lua
```
-- ------------------------------------------------------------
-- Action Phase

pPosition:Init( fPick( Vec2(-10,0), Vec2(10,0) ) + dLocus );

-- Pick a velocity from a range
pVelocity:Init( fRange( Vec2(-200,-300), Vec2(200,120) ) );
```

Here we set the initialize function for pPosition to select a point 10 pixels to the left or 10 pixels to the right of the mouse, which is retrieved from the data source dLocus. Then we select a velocity in the given range: the fRange() function will take its arguments and return values randomly within the range in each dimension. So for a Vec2() with the parameters above, it will return -200...200 in the X direction and -300...120 in the Y direction.

The function fRange() will work with scalars, Vec3() and Color() values similarly.

Lua
```
-- Start color (tint) off as white (natural color of image)
pColor:Init( Color(1,1,1,1) );

-- Start scale out as 0.5
pScale:Init( 0.5 );

-- Start age as 0 milliseconds
pAge:Init(0);
```

These are simple enough: Set the initial pColor to a constant white color value, the initial pScale to a constant 0.5, and the initial pAge to 0 seconds.

Lua
```
-- Start initial rotation as a random angle
pSpin:Init( fRange( 0, 2*3.1415927 ) );

-- Start spin velocity random from -10 to 10
pSpinSpeed:Init( fRange( -10, 10 ) );
```

Here we're again using fRange(), this time with scalars, to initialize the particle parameters with values within a range.

Lua
```
-- ------------------------------------------------------------
-- Particle Parameter Animation Functions

pVelocity:Anim( pVelocity + fTimeScale(Vec2(0,400)) );
pPosition:Anim( pPosition + fTimeScale(pVelocity) );
pScale:Anim( pScale + fTimeScale(1) );
```

First we see the familiar velocity and position equations. Then we see something new: Animating the scale value. This equation will add one to scale per second: fTimeScale() again scales its parameter to the fraction of time that's passed, so after a second pScale will go from 0.5 to 1.5.

Lua
```
pAge:Anim( pAge+fAge() );
pColor:Anim( fFade( pAge,Color(1,1,1,1), 500, Color(1,0,0,1), 1000, Color(1,1,1,0) ) );
pSpin:Anim( pSpin + fTimeScale( pSpinSpeed ) );
pUp:Anim( f2dRotation( pSpin ) );
```

Three new functions here. First is fAge(), which takes no parameters. Instead it just returns the amount of time in the current update step in milliseconds, so with the equation above the pAge parameter will always equal the particle's age in milliseconds.

The second new function is fFade(), which performs a linear interpolation between multiple values. The first parameter to fFade() is the current age in milliseconds, the next is the initial value, and following that are pairs of deltas and values to interpolate between. In this case, it starts with Color(1,1,1,1), and then over the next 500ms interpolates to Color(1,0,0,1), then over the next 1000ms interpolates to Color(1,1,1,0). There isn't a limit to how main pairs you can add to this sequence, as long as you end with a value and not a delta.

The next line calculates a new pSpin value by using fTimeScale() to increment pSpin by the value of pSpinSpeed once per second. The last new function is f2dRotation(), which produces a 2d vector based on a value in radians, which is what we've been calculating in pSpin.

Lua
```
Anim( fExpire( fGreater(pAge,2500) ) );
```

Here's a different concept: A function that doesn't return a value. Each of the functions so far has been implicitly assigned to a particle parameter. This one just executes its function and ignores any return value.

Which is fine, because it is a function with a side effect. From the inside of the parenthesis: fGreater() returns true (a non-zero value) if its first parameter is greater than its second. Then the function fExpire() tests its parameter, and if true, flags the particle for destruction. In other words: If the particle's age parameter exceeds 2500ms, kill it.

Lua
```
-- A global variable that we can set from outside
gActive = 1

-- A function to run as we're executing
-- seconds - how many seconds have elapsed
function Update(seconds)
  if gActive>0 then
    -- Create 10 particles per second
    CreateParticles( seconds * 10 );
  end
end
```

What's all this then? OK, more fun stuff. First, we're setting a global Lua variable. There's nothing magical about this—nothing you can't read about in a Lua manual, anyway. We're just setting the variable gActive to a value of 1. Thing is, from C++ you can use TScript::SetGlobalNumber("gActive",0) to turn of new particle generation without stopping the animation.

It's important *not* to add any more animation or initialization rules in the Update() function. Every time you call Anim() or Init() it adds another animation or initialization function to the particle processing chain, so if you add

additional functions in Update() the processing chain will get longer and longer, and your particle processing will get slower and slower.

Playground ships with a particle editor to help test and refine your particle systems. For information about the particle editor see FluidFX 2.0: Interactive Particle System Editor.

6.1.7 Advanced Tip: Arithmetic

In addition to using the addition and multiplication operators in Lua to add together particle properties and function results, you can also use numerical constants with addition, multiplication, subtraction, and division. It's important to note that if you use a constant, though, it needs to come after the particle property or function result that you're modifying.

```lua
pSpin:Anim( pSpin + fTimeScale( pSpinSpeed*0.5 ) );
```
Lua

This code will take the value in pSpinSpeen and multiply it by the constant 0.5 before running it through the fTimeScale function. In this case, you could just as easily have written:

```lua
pSpin:Anim( pSpin + fTimeScale( pSpinSpeed )*0.5 );
```
Lua

But this is an error:

```lua
-- ERROR: Lua can't handle multiplying a number by a USERDATA in this case
pSpin:Anim( pSpin + fTimeScale( 0.5*pSpinSpeed ) );
```
Lua

You can also multiply vectors by a scalar:

```lua
pVelocity:Anim( pVelocity + fTimeScale(Vec2(0,250)*10) );
```
Lua

Obviously, in this case you could simply change the constant 250 to 2500, but if you've assembled a vector from several values and want to scale the entire result, you can do that.

The order of operands is important here: It always needs to be Vector∗Scalar, even if the scalar is already a Lua userdata that represents a value.

Chapter 7

Localization and Web Versions

7.1 The String Table

Playground SDK™ applications rely on a single XML file that contains all of the strings used in the application. This XML file is easily modified by translators for localization. The XML file is stored in the Microsoft Excel XML save format, which can also be edited using OpenOffice Calc.

When an app launches, it automatically attempts to load in a string table from the assets directory. Note that the file will contain other XML that supports the Excel save format, and the Cell and Data tags can contain attributes.

A single string mapping looks like:

```xml
<Row>
<Cell><Data>title</Data></Cell>
<Cell><Data>My Game Title</Data></Cell>
</Row>
```

XML

The file is expected in the root of the assets folder, and should be called strings.xml.

See TStringTable for more information. The global string table is available from TPlatform::GetStringTable().

7.2 Quick and Easy Web Games

Running your game as a web application with Playground is simple - you don't even need to recompile your game. For more information about how to run your game as a web applicaion, please read the axtool: Testing Your Game in a Browser section under the Playground Utilities section.

However, there are a few things to keep in mind when developing the web version of your game:

- It is important that your web version not have a completely functional game executable. It is not enough to remove data from your game and use the same executable as the download version. This would allow users to simply copy over their web version executable and unlock a full version of the download game. Please make sure to limit the functionality of your web game with code changes as well as data changes.
- Playground will automatically resize your game to fit in whatever window the website puts your game in. An average window size is 480x360, though some sites will use smaller windows and others will use larger windows. You do not have to do any extra work to get your game to show up properly in that size window. However, because your game will be running in a smaller window, you may want to shrink your assets and then scale them up dynamically in-game (i.e. using the scale parameter in DrawSprite). For example, if you shrink your assets by 25% and then scale them up in code by 25%, your game will run in a 600x450 window with no visual loss. If you do the same thing by 40%, your game will run in a 480x360 window with no visual loss. Shrinking your assets in this manner can help you reduce the size of your web game download.
- To implement the "Download" button functionality, you need to do two things:
  - The download URL will be available in TPlatform::GetConfig(TPlatform::kDownloadURL).
  - To launch the download page, use TPlatform::OpenBrowser().
- If your game uses the hiscore system, it is important that your game be allowed to run in "anonymous" hiscore mode. The command line to main() will contain a -anon # parameter, which you need to parse. If # is 0, you can run the full hiscore system. If it is 1, you need to run anonymous mode. If it is 2, you need to run in local high score mode.
- If your game displays the PlayFirst logo anywhere aside from the splash screen, you need to look for the "-hidelogo 1" parameter being passed into the command line. If this value is present, you need to hide these logos on every screen except the main menu screen, where this logo is allowed to remain.
- Avoid saving data between sessions. The TPrefs and TPfHiscores constructors have optional arguments that will disable session data saving.
- You will likely need to revisit some of your UI and increase the size of text to make sure it is readable in the small 480x360 windows.
- If your game uses the PlayFirst Hiscore System, you will want to add separate hiscore modes for your web game, but make sure the web game can still view the downloadable game modes hiscores when connected to the global hiscore system.
- In order for your game to be accepted for use on MSN, you need to:
  - Implement all the functionality appropriate to your game described in gamestate.h. See some examples in the sample application.
  - Submit two builds of your game, one with cheats on, one with cheats off, so it is easy to test the web functionality.
  - Detect if the game is an MSN build by checking for the string "msn" in TGameState::GetState(TGameState::kSystemMode).

  - When in MSN mode:
    1. Instead of calling TPlatform::OpenBrowser() to launch a download, you need to call TGameState::SetState(TGameState::kDownloadButton).
    2. Remove all "Try Again" buttons from your game - at the end of your game the only option should be to quit.
    3. Because MSN has their own high score system, you should not go to the high score screen in game. Instead, you should either go to an upsell screen if the user presses the high score button, or you should remove the high score buttons from your game.
    4. Your game score needs to be sent to TGameState::SetState(TGameState::kScore, score) at least prior to the end of the game, but more often is better.
    5. Whenever a game ends (either by being over or by the user quitting the game in progress), call TGameState::SetState(TGameState::kGameOver). At this point, MSN zone may switch to an upsell screen and then redirect your game to either go to the main menu, or possibly restart the current game mode (you need to poll the TGameState::QueryRestartMode() and TGameState::QueryJumpToMenu() functions to determine whether or not to do this).

- In order for your game to be accepted for use on the AOL.com® site, you will need to do the following:
  - Detect for the string "aol" in TGameState::GetState(TGameState::kSystemMode), which means you are running an AOL service build.
  - If you are in AOL service mode, you need to:
    1. Display the AOL logo on the splash screen. An AOL logo has been included with your Playground delivery, and is located in the addons/webgames/AOL folder (two logos have been provided, one that is baked onto the PlayFirst splash screen, and one that is seperate - you can use whichever file is easier for you). If you are using the DisplaySplash() Lua function to display your splash screen, you can put logic in the Lua script to determine which splash screen to display. For instructions on how to use an overlay with DisplaySplash, refer to the DisplaySplash documentation.
    2. Only have download buttons on 2 screens: the main menu, and the upsell screen.

Chapter 8

Game Footprint

8.1 Smaller is Better

When shipping a downloadable game, the smaller it is, the more likely it is that people will successfully download it. More than that, bandwidth also costs money, and though you may not see that cost directly, a larger footprint may mean your game gets promoted less than a game with an equivalent conversion rate but with a smaller footprint.

8.2 Shrinking Your Game

There are a number of strategies you can use to reduce footprint without reducing perceived content. Here we'll go into a few of them.

8.2.1 Reducing PNG Footprint

PNG32 files offer less-than-ideal compression for images that have photographic detail or gradients, but sometimes you need a mask layer for an image. There are two common ways to approach this problem.

First, you can run your PNG files through a conversion script that splits the RGB layers from the mask layer, and then place the RGB layers into a much-better-compressed JPG file, leaving the mask in a monochrome PNG8 file. The key advantage of this technique is that you get a high quality image attached to a lossless mask at a much better compression ratio. Masks tend to compress especially well in PNG. The disadvantage of this technique is that your source code (or at least Lua code) needs to change to load the masks–it's not automatic.

Second, you can run your PNG files through a conversion process to convert them to PNG8-with-transparency. This is a bit more difficult than it seems, because most popular tools don't correctly read files saved in this format. Adobe Fireworks is one of the few that can successfully read and write this format, though there is also a free command line tool called pngquant that we use at PlayFirst to reduce PNG32 to PNG8 files. The advantage of this conversion is that your files become 1/4 of their previous size, with no change of source code; the disadvantage that they're reduced to an 8-bit palette, so if the image consists of many different color gradients, it may hurt image quality.

It's easy enough to experiment and figure out which of these two techniques works with each of your image types.

8.2.2 Small Files Can Be Smaller!

Just because you have a directory of 2-4k PNG files that together add up to a megabyte, don't assume that they're already small and shrinking them won't help. Files like that are almost always going to compress well to PNG8-with-transparency, as described above. And we've frequently pulled an extra 768k out of one of those megabyte-sized folders, even though the individual files are small.

8.2.3 Reducing JPEG and OGG Footprint

JPEG and OGG/Vorbis files each have variable compression ratio. During development it's tempting to save each of these files at a high quality, but when it comes time to ship and you're looking for some extra room, it pays to experiment with higher compression settings. Due to the nature of the lossy compression of both JPG and OGG/Vorbis, some files will compress much better than others with no perceivable loss in quality.

8.2.4 Shrinking Your Game the Easy Way!

In early 2008 the Playground SDK™ sidewalk utility got a new purpose in life: In addition to creating animation files, it can walk through your assets hierarchy and compress all graphics files to the format and compression level that comes out smallest for each file, while keeping the quality above a minimum threshold. For more information, see sidewalk: Animation Creation and Asset Compression.

8.2.5 Reducing Image Size in Web Games

If you're producing a Web version of your game, you might want to consider reducing asset dimensions to reduce your footprint even more. While the game will work just fine with original assets, and will automatically shrink down to the smaller dimensions required by web sites, this means that you're sending 800x600 images to the site even though it's only displaying them at 400x300 or 640x480.

One easy way to reduce asset size when the assets are specified in a Lua Bitmap{} call is to add an appropriate "scale" parameter to Bitmap{}. Say you're reducing your assets to 640x480. 800/640 = 1.25, so if you add a scale parameter of 1.25 to each bitmap you downsize, the asset will display at the same logical size in the game. If the web game is actually displayed at 640x480, it will end up displaying the resulting bits at 1:1; if it's displayed smaller (down to 400x300 on some sites) then it still ends up being scaled down, but not as much as if you were sending original 800x600 artwork.

Chapter 9

Utilities

9.1 FirstStage: The Playground Resource Editor

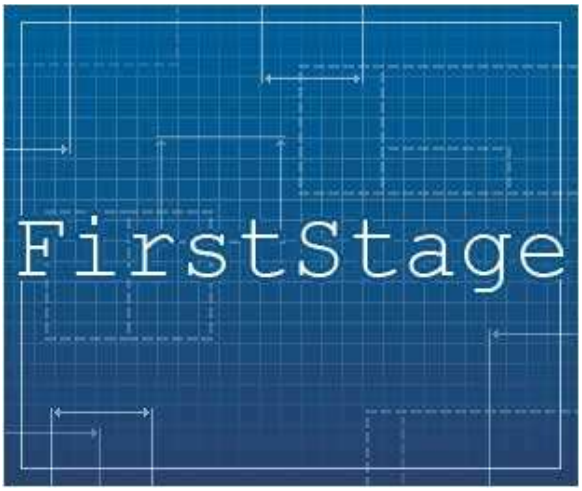


Figure 9.1: FirstStage Logo

9.1.1 Welcome to FirstStage

FirstStage is a RAD editor designed specifically for the Playground SDK which allows you to design window layouts for uses such as dialog boxes, main menus, etc.

By simply clicking on window types on the toolbar, you can rapidly design complex window layouts which would take much longer if editing the Lua files by hand.

Using FirstStage, you can create windows with buttons, images, text fields, text input boxes, and even custom windows.

9.1.2 Getting Started

If you are not already running FirstStage, you can find the program in the bin folder of the Playground SDK. It is called FirstStage.exe.

When you run FirstStage, you will be presented with a screen similar to this one.

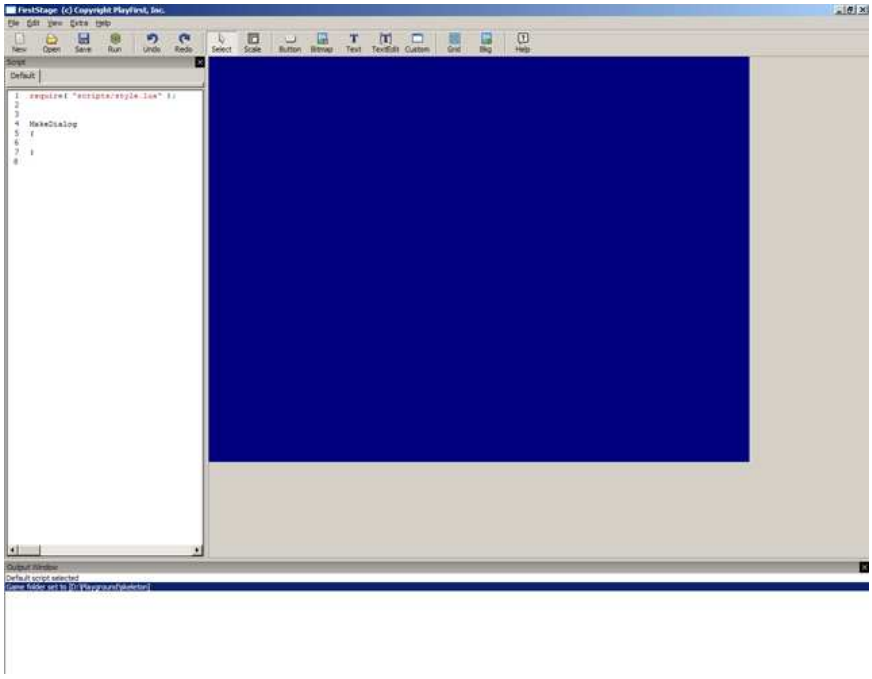


Figure 9.2: FirstStage Default Screen

If this is the first time you have run FirstStage, then you will be presented with a folder selector asking you to select your game folder. Go ahead and point it to your game folder. For the skeleton project, this is usually

```
[PlaygroundSDK folder]\skeleton.
```

Once this is done, it will give confirmation of the assets folder in the output window. Check that it is correct before you proceed.

9.1.3 Your First Dialog

You will notice that by default, there is already some code in the Lua script editor. This is the basic template code that all window layouts use. When we create new window controls, the code for them will appear inside the MakeDialog block.

Let's go ahead and start building a dialog box.

PLEASE NOTE: For this example, we are going to be using assets from the skeleton folder which is included with the Playground SDK distribution. You are free to substitute your own assets in this example if you already have them prepared.

If you want to follow this example and use the skeleton's assets, make sure your game folder is pointing to the skeleton application game folder by selecting **Extra/Select game folder...** in the menu.

Click on the Bitmap button in the tool bar. An image selector dialog will appear with the game folders on the left and thumbnail pictures of your assets on the right.

Go into the backgrounds folder and select dialog.png. Your cursor will change and there will be a rectangle that follows your mouse cursor. This rectangle represents the size of the image.

Move the box until it is properly on screen and click the left mouse button to place the bitmap. A dialog box will then pop up asking you to enter a unique name for this new bitmap. Let's just call it TestWindow.

Press ok and you see your bitmap appear on screen. This bitmap will be our background for the dialog box.

If you move your mouse over the bitmap, you will see that it has a yellow focus rectangle around it. Select the bitmap by clicking on it. It should now have a red box surrounding it indicating that it has been selected. You should also notice that the bitmap code in the Lua window has been highlighted. This is to show you what code is involved in making this window.

Next, let's add some text to the window.

Click on the Text tool in the tool bar. The cursor should change to a text image.

Move the cursor to somewhere inside your bitmap and click and drag out a box. This will be the dimensions of our new text box.

When you release the mouse, a dialog box will appear.

Set the ID Name to gTextBox.

Set the label to HelloMsg. The Label is actually a unique identifier for the translations file which we will explain in more detail below.

Click on the middle button in the Text Alignment and then press OK.

One great feature about the Playground SDK is that it can do automatic translating of strings for you. This means that when you create a text control in FirstStage, you also have to add a translation entry for the new string. Luckily, FirstStage makes this process easy by displaying a translations editor each time you create a control that has text. You will notice that a new entry has been created and selected in the translations table which corresponds with your new text. Go ahead and change the text in the translations column to "Hello There" (without the quotes) and press enter. Then press the OK button to save your new entry.

Your new text box should appear. If you move your mouse over the text area, a highlight rectangle should appear around the area that you specified for the text.

You should have something similar to this on screen:

Figure 9.3: Basic Dialog

Let's add some more stuff!!

A dialog box wouldn't be complete without an OK button.

Click on the **Button** tool.

The button control dialog box will pop up asking for some details.

Set the ID name to gDlgOK.

Set the label to OK. This label already has an entry in the translation file so you won't be prompted to add a new entry.

Select the "default" checkbox.

We will keep the default style button so just press the OK button.

You should now see a box following your mouse pointer around. This is the dimensions of your new button. Use this as a guide to where you want to place the button.

Click the left mouse button to place your new button on your dialog.

You should now see a button appear at the bottom of your dialog.

If the button isn't located in the right place, hover over the new button until it is highlighted and then click your left mouse button. The highlight rectangle should turn red.

Now click and drag with your left mouse button to reposition the button to where you want it.

The end result should look like this:

Figure 9.4: Dialog with button

You have just created your first dialog box. Click on the **Save** button and save your new dialog as my_dialog.lua.

9.1.4 The Main Toolbar

Figure 9.5: FirstStage Toolbar

Here is a brief description of what each tool button does:

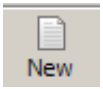


Figure 9.6: New Button

This will create a new blank script. The current list of scripts can be seen in the Scripts window.

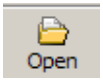


Figure 9.7: Open Button

This will load a script from disk and add it to the Scripts window.

Figure 9.8: Save Button

This will save the currently selected script to disk.

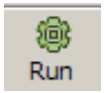


Figure 9.9: Run Button

If you type in any changes in the script window, you will need to press the Run Script button to refresh the changes in the visual display.

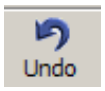


Figure 9.10: Undo Button

Will undo any changes made to the script.

Figure 9.11: Redo Button

Will re-apply any changes that were made to the script.

Figure 9.12: Select Button

FirstStage can be in either of two states when building a dialog. Firstly, there is Select mode which allows you to create new windows and move windows. The other state is Scale Mode which allows you to scale the size of any selected windows.

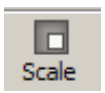


Figure 9.13: Scale Button

If you need to change the size of a window control, like an image, click the Scale Mode button, then select the window(s) that you want to resize, then click and drag the left mouse button to resize the window.

Figure 9.14: Add Button Control Button

If you want to add a push button, a check box, or something similar use the Add Button tool. For more information on the use of Add Button, see Add Button Control.

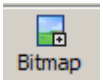


Figure 9.15: Add Bitmap Control Button

Most dialog boxes start with a bitmap as the background and then you layer buttons and text boxes on top of the bitmap. So, adding an image can be seen as the first step in building a new dialog. For more information on the use of Add Image, see Add Bitmap Control.

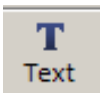


Figure 9.16: Add Text Control Button

You use a static text box when you want to display information to the user. Text is automatically translated through Playground SDK, so all you need to do is enter the text ID and then fill in the translation XML file with the correct translation. For more information on the use of Add Static Text, see Add Static Text Control

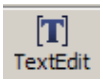


Figure 9.17: Add Text Edit Control Button

If you want the user to be able to type some input on the screen, use a Text Edit control. You can adjust a lot of parameters for the use of the Text Edit when placing the control. For more information on the use of Add Text Edit, see Add Text Edit Control.

Figure 9.18: Add Custom Window Control Button

Custom windows are the basic building blocks of any Playground derived game. In C++, all of your game code resides inside a custom window (a TWindow) so that it can be visualized on screen. Using the custom window

tool, you can define the size and position of a custom window which can later be referenced inside your game. For more information, see Add Custom Window Control.

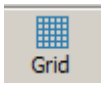


Figure 9.19: Toggle Grid Display

Grid snapping is very useful when you have a lot of controls on your screen and you want to align them so that it looks neat and professional. Pressing the Grid Snapping button will toggle the use of grid snapping on/off. If you want to adjust any aspect of the grid snapping properties, for instance the grid size, just right click the Grid Snapping button and the Grid Properties dialog will be displayed. For more information on the grid properties, see Grid Properties.

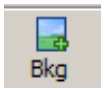


Figure 9.20: Load Background Button

To change the background of the screen, click the Bkg tool button. You will be asked for a valid bitmap. Bitmaps can be any JPG or PNG file. To remove the bitmap from the screen, go to **Extra/Clear Background**.

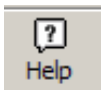


Figure 9.21: Help Button

Clicking this button will display the help that you are reading now.

9.1.5 Add Button Control

Figure 9.22: Add Button Control Button

If you want to add a button to your screen, press the Button tool. You will be presented with the button properties screen.

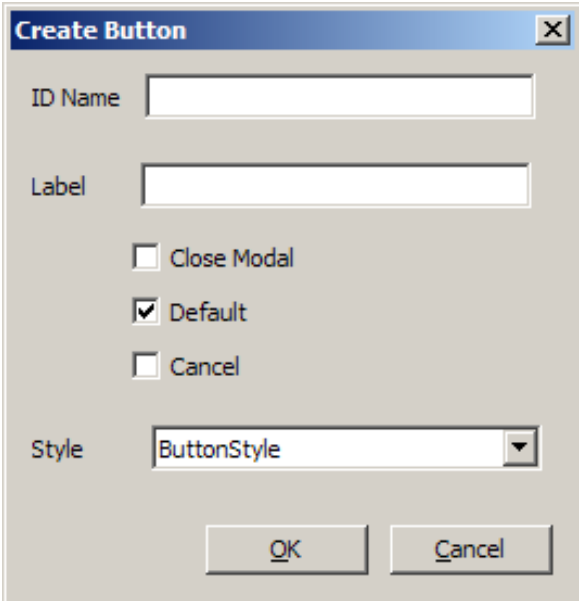


Figure 9.23: Button Control Properties

The **ID name** is a unique name to give to this new control so that Lua can differentiate between the different windows. Typical IDs are names like gOkButton, gFullScreenCheckBox, etc.

The **Label** is the text identifier for this label. Common identifiers can be "OK", "Cancel", "HelloMsg", etc. If the identifier does not exist in the translations file, you will be presented with the translations editor once you hit ok. At that point, you can type in the English equivalent for your identifier. See The Translations Editor for more information.

The **Close Modal** checkbox will tell your game that you want to close the parent dialog when the window is closed.

The **Default** checkbox will tell the game that this is the default button.

The **Cancel** checkbox will tell the game to abort the window.

The **Style** drop down list holds a list of all the global styles for this Lua file. Pick a style that suites the type of button that you want to create. Commonly used styles are "ButtonStyle" and "LongButtonStyle". You can make your own custom button styles which will also show up in this list.

Press the OK button once you have filled in the details.

As mentioned, you wil be taken to the translations editor if your label doesn't already exist.

The cursor will then change to indicate that you are in the Add Button mode. Move your mouse to where you would like to place your button. Use the box outline that is following your mouse as a guide to where you would like to place your button.

9.1.6 Add Bitmap Control

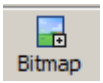


Figure 9.24: Add Bitmap Control Button

Bitmaps are very important to the look and feel of your dialog. Typically, you would use a bitmap for the background of your dialog.

To place a bitmap onto the canvas, first select the Bitmap tool button.

You will be presented with an image selector window. Use the folder tree on the left to navigate to your image. If your folder is not in the tree list, then you possibly may not have configured FirstStage to be pointing to your game folder. Use the menu option **Extra/Select Game to Modify...** to change to the proper game folder and try again. Select the image you want to use and press OK.

Once you have selected your bitmap, move your mouse around the screen. You will notice that the mouse cursor has changed to a bitmap icon and there is a rectangle following your cursor. This rectangle represents the dimensions of your new bitmap. You now have an idea how big your bitmap will be on screen.

At this point, you have two options:

1. If you are happy with the size and position of the bitmap on screen, then press the left mouse button to place your new image.

2. If on the other hand the image is too big or too small, you can hold down the control key and then click and drag a new size for your bitmap. When you are dragging out a size, you will notice that the aspect stays constant. This is because images scale equally in both dimensions. Once you are happy with the new size of your image, release your mouse button to place your image.

You will now be presented with the bitmap properties dialog:

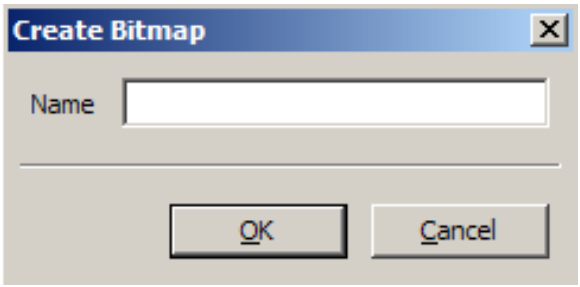


Figure 9.25: Bitmap Control Properties

You only have one parameter that you need to fill in which is the **Name** for this image. Type in a unique name for the image and press OK to see your image on screen.

9.1.7 Add Static Text Control

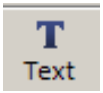


Figure 9.26: Add Text Control Button

Text boxes are great for displaying information to the user. The text displayed in a text box is automatically translated via Playground into the correct language.

To create a text area on screen, click on the Text tool button. The cursor will change to indicate you are in this mode.

Next, with your left mouse button, click and drag out a box where you want your text to be displayed.

You will then be presented with the Text Properties dialog:

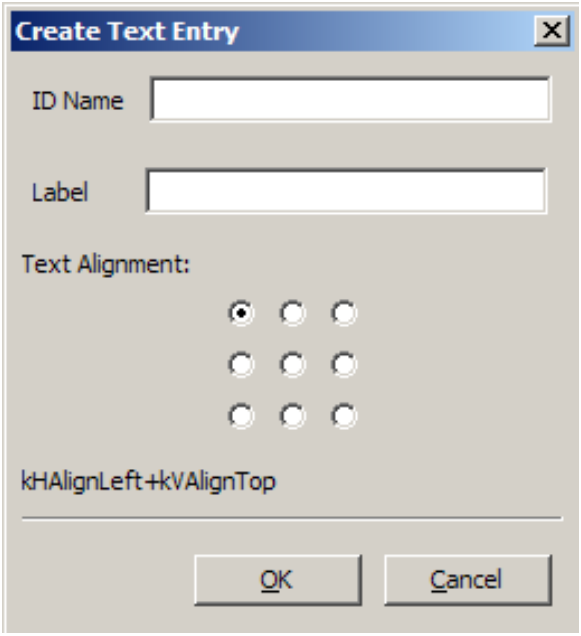


Figure 9.27: Text Control Properties

The **ID name** is a unique name to give to this new control so that Lua can differentiate between the different windows. Typical IDs are names like gTextBox, gInfoBox1, etc.

The **Label** is an identifier into the translations file for the text you want displayed. See The Translations Editor for more information on the translation file editor.

The **Text Alignment** is where you want the text displayed in relation to the boundaries of your box dimensions. Use the grid of buttons to pick how you want your text aligned. The top left button will mean that your text is aligned to the top left of your text box. The middle button would mean your text is aligned to the middle of your text box, etc.

Press the OK button to see your new text box on screen.

9.1.8 Add Text Edit Control

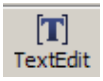


Figure 9.28: Add Text Edit Control Button

A TextEdit control is typically used to allow the user to type a response for use in game. A typical example is a user's name or password.

To create a text edit area on screen, click on the TextEdit tool button. The cursor will change to indicate you are in this mode.

Next, with your left mouse button, click and drag out a box where you want your input box to be displayed.

You will then be presented with the TextEdit Properties dialog:

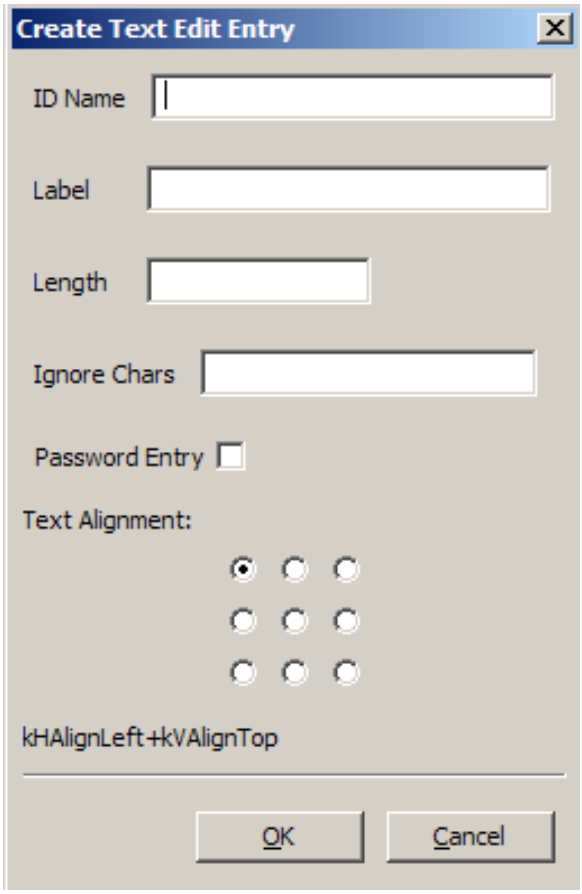


Figure 9.29: Text Edit Control Properties

The **ID name** is a unique name to give to this new control so that Lua can differentiate between the different windows. Typical IDs are names like gPasswordInput, gUserNameInput, etc.

The **Label** is the default text that should be displayed in the edit field.

The **Length** is the maximum number of characters that can be entered.

The **Ignore chars** is a list of letters that you do NOT want the user to be able to use.

The **Password entry** checkbox can be set if you want the user's typed text to be masked for privacy.

The **Text Alignment** is where you want the text displayed in relation to the boundaries of your box dimensions. Use the grid of buttons to pick how you want your text aligned. The top left button will mean that your text is aligned to the top left of your text box. The middle button would mean your text is aligned to the middle of your text box, etc.

Press the OK button to see your new TextEdit box on screen.

9.1.9 Add Custom Window Control

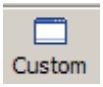


Figure 9.30: Add Custom Window Control Button

Custom windows are used as a canvas for your TWindow derived classes in your game. You need to create a custom window to be able to see your game on screen.

Click on the custom window button and left click and drag out a box on screen in the position and size that you want for your custom window.

You will then be presented with the Custom Window Properties dialog:

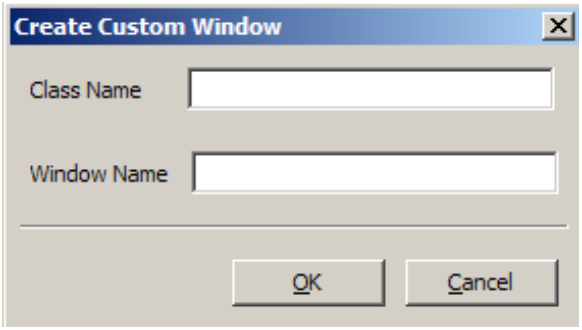


Figure 9.31: Custom Window Properties

The **Class name** is the name of your c++ TWindow derived class that you have defined in game.

The **Window Name** is the unique ID that you want to give to this window.

Press the OK button to add your custom window.

9.1.10 Grid Properties

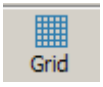


Figure 9.32: Grid Properties Button

The grid snapping properties can be accessed by right clicking the Grid tool button in the tool bar or by selecting **Extra/Change Grid Properties...** in the menu.

The following dialog box will be displayed:

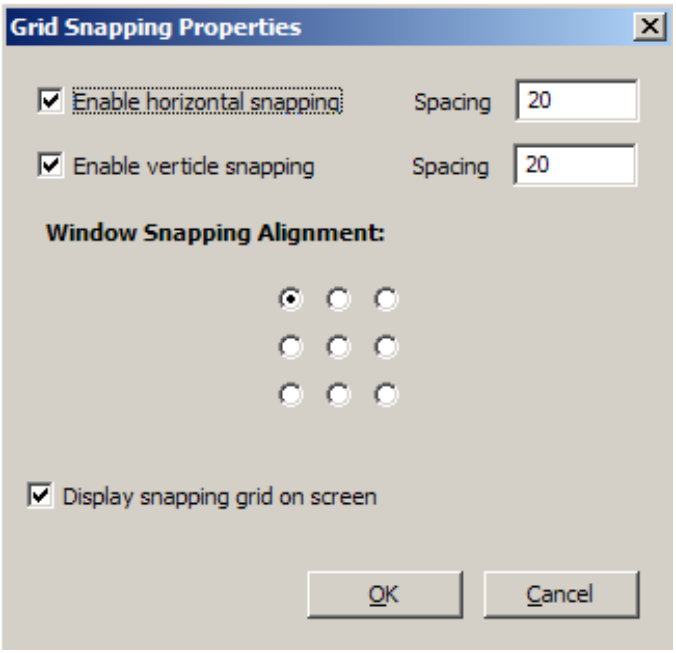


Figure 9.33: Grid Properties Dialog

The **Enable Horizontal Snapping** checkbox allows you to enable/disable snapping in the horizontal direction. The text field following this option is the distance in pixels between each column.

The **Enable Vertical Snapping** checkbox allows you to enable/disable snapping in the vertical direction. The text field following this option is the distance in pixels between each row.

The **Window Snapping Alignment** is the edge that you which your windows to be snapped to. By default, windows will snap to grid lines using the top left edge of the window.

The **Display Snapping Grid on Screen** checkbox allows you to toggle the visibility of the grid on screen. This does not disable snapping.

9.1.11 Context Menu

If you hover over a window, for instance a bitmap or a button on the canvas, and you click your right mouse button, a small popup menu will be displayed similar to this one:

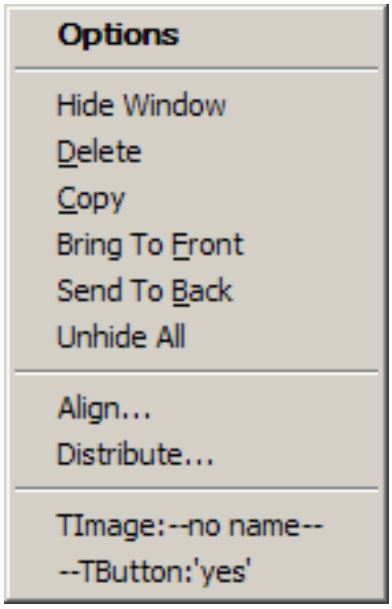


Figure 9.34: Game Screen Context Menu

Hide Window(s) will hide the selected windows.

Delete will delete the code for the selected window from the Lua script.

Copy will copy the window code to the clipboard.

Bring to Front will move the order that the window is drawn so that it will be in front of the other windows.

Sent to Back will reorder the selected window so that all other windows are drawn first.

Unhide All will unhide any windows that were hidden with the "hide window" command.

Align will bring up the alignment dialog. From here, you can snap all selected windows to a certain orientation.

Distribute will bring up the distribute dialog. This will allow you to space the selected windows out in a direction.

The remaining items in the menu are the current list of windows that are beneath the mouse. If you are having problems selecting a window, you can use this list to find and click on the window that you want to select.

9.1.12 Alignment Dialog

When working with a lot of window's controls, like buttons, it is important to have the ability to align a certain edge of these controls so that they look neat. This is very simple to do with the alignment dialog. To access the alignment dialog, first select the windows that you want to align, then either right click on one of the windows and select "Align..." or select **Edit/Align selected objects...** in the menu.

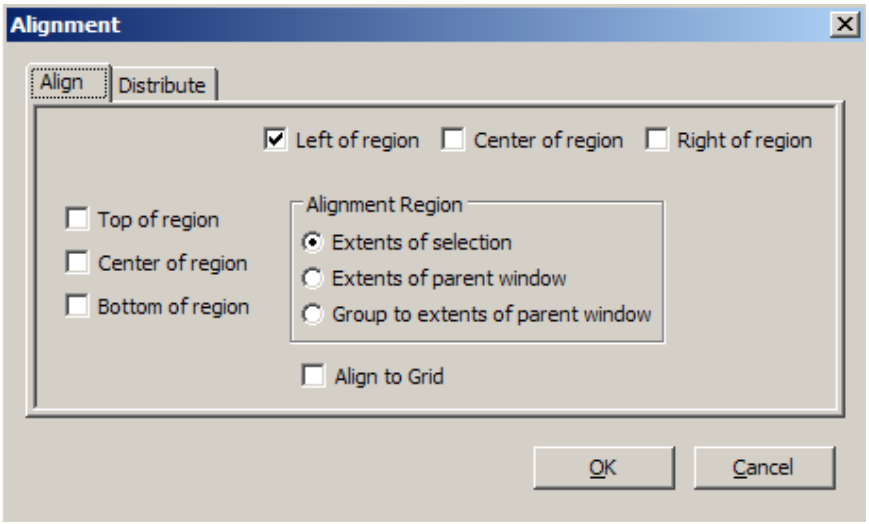


Figure 9.35: Alignment Dialog

There are three parts to the alignment dialog.

1. The Alignment Region - These three options specify where you want the windows aligned.

Extents of Selection - will calculate the bounding region of all selected windows and align to one of the edges.

Extents of parent window - will align the objects to a border of the parent window of the selected windows.

Group to extents of parent window - will align all selected windows as one group to a border of the parent window.

2. The edge to align to - these are the items along the top and side of the dialog. Choose the edges that you want your selected objects to align to.

3. Align to Grid - setting this option will make sure that all of the selected windows will snap to the current grid settings once they have been adjusted.

9.1.13 Distribute Dialog

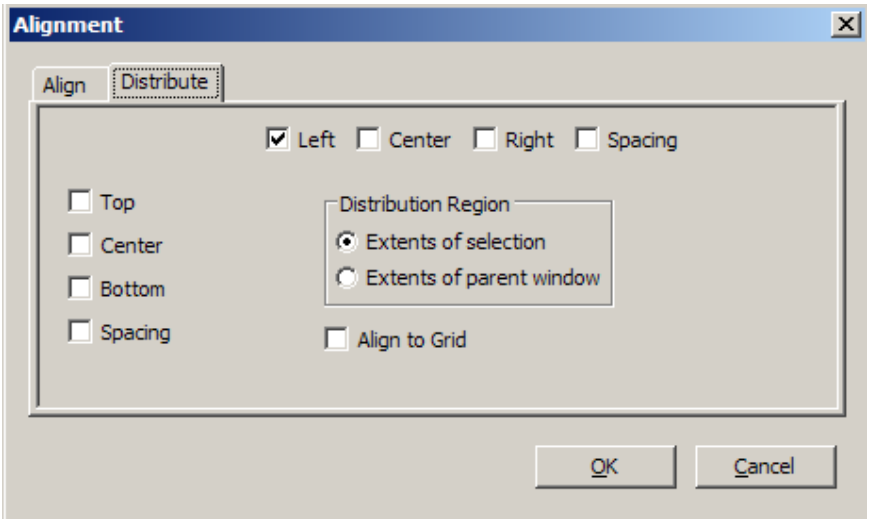


Figure 9.36: Distribute Dialog

The Distribute dialog is very similar to the alignment dialog but with a slight difference. If you distribute the windows, you are affectively spacing the windows out evenly along a certain axis. Typically, you might have three buttons that you want spaced evenly along the bottom of the screen. You would select all three windows and then use the distribute tool to space them out correctly.

Left, Center, Right, Top and Bottom refer to the edge of the controls which you want to distribute. For instance, if you selected to distribute "Top", then all the controls would be evenly spaced based on the tops of the controls.

If you select, "Spacing" as an option, then FirstStage will make sure that the space between each control is exactly the same.

9.1.14 The Translations Editor

The translation editor is a dialog that allows you to enter new values or modify old values of strings which are found inside the strings.xml file located in your assets folder.

String entries are always defined as a key/value pair. So, you will always start with a key string that is a unique identifier for the value string which is the normal text. For example, "HelloMsg" would be the key and "Hello there everybody!" would be the value. The key string is what you enter into your label section when creating a button or text control. The text will then automatically be replaced with the value for that key when it is displayed on screen.

The translations editor can be activated in two ways. Firstly, if you were to create a new button or text control and you entered a label that currently doesn't exist in the strings.xml file, then the translations editor will automatically be opened and your new value added. Alternatively, you can access the editor at any time by selecting **View/Show Strings Editor...** from the main menu.

The editor is broken up into three sections. At the top, you have a filter search box. If you start typing in the mask field, you should notice that your list of displayed strings will be reduced to only those strings that match your typed text. This is a very quick way of finding a string in a huge list of items. The "Filter on ID" option (default) means that it will try and match your mask string with a string in the identifier field and similarly, the "Filter on Translation" will search on the translation field. If you select "Substring" matching (default) then your mask must exist inside the field. Example, if your mask was "ing", it would match strings with "fishing", "cooking", etc. The

wildcard search is a bit different. Using wildcard searches allows you to be more precise with your selections. You could enter a mask like "c*ing" and it would match "cooking" but not "fishing".

The next section is the key/value cells where you can edit your data. The left column is the index number for the key/value in the xml file. The next column is the key and the last column is the translation for that key. All of the columns can be resized by dragging the splitter area between each column. You can double click on a column header to change the sorting order. Double click once for ascending and double click it again for descending order. Double click on a cell to edit the contents of that cell.

The last section has some buttons which help manage your strings. Currently, you can Add and Delete key/value items. Click Add to add a new key/value to the end of the list. If you want to delete an item, first click on either the key or the translation part of the item and then press the delete button.

Once you have finished with your strings, press the OK button to save your changes or press Cancel to abort your changes.

9.1.15 The Image Selector Dialog

The Image Selector Dialog is a visual way of being able to navigate your game folders for assets. The dialog is automatically triggered when you create a new Bitmap control.

The column on the left side of the dialog is a list of all of the folders in your game. Whenever you click on one of these folders, the images in that folder are displayed in the column on the right of the dialog.

Once you have found the image that you want to use, you can either double click on the icon to choose it or you can click once on the icon and then press the OK button.

9.2 Building A Dialog Tutorial

Introduction

FirstStage is a tool that allows non-technical people to create and edit windows and dialogs within Playground games. This tutorial is designed for anybody with limited knowledge of FirstStage who would like to jump right in and create something. We will be working with the Skeleton demo application that is bundled with the Playground SDK. This tutorial assumes that you've unzipped to c:/firststagetest/. If you've unzipped to another path, please adjust the given paths accordingly. What this tutorial will cover is setting up a simple about box with an image, some text, and an OK button.

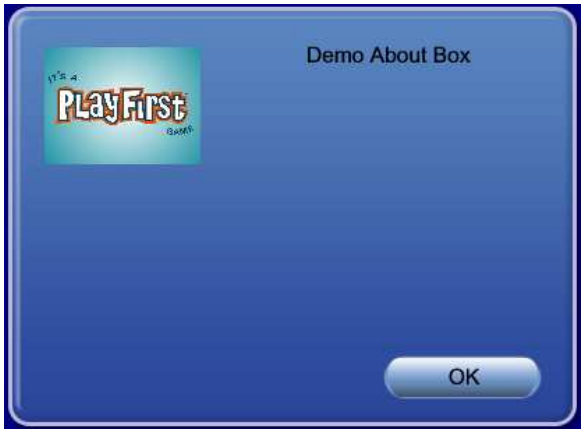


Figure 9.37: Demo About Box

We will then create a new button on the main menu screen that will display this new dialog.

So, let's get started.

Running FirstStage

FirstStage is located in the bin folder of the tutorial archive, similar to c:/firststagetest/bin/FirstStage.exe.

Double click on FirstStage and once it is running, if you've never run FirstStage before, it will pop up with a dialog box informing you that this is your first run and that it has found a game folder at c:/firststagetest/skeleton/ which is exactly what we want.

Figure 9.38: Setting the Path

FirstStage needs to access the files in the game folder, so if it is not pointing to the correct location, you will need to change it to do so. If for some reason it can't find the game folder, it will pop up a folder requester dialog and ask you to find the game folder. If you have run FirstStage before, and it doesn't ask you for a folder name, then for this tutorial please select the Extra menu, "Select Game to Modify", and select the skeleton folder that you unzipped (e.g., c:/firststagetest/skeleton).

Let's build a Dialog!

1. Select your Dialogue Background:

Click on the Bitmap button in the toolbar. An image selector dialog will appear. On the left hand side is the folder hierarchy of your game folder. If you can't see it properly, resize the image selector dialog until you can. You should see a folder called "backgrounds". Click on this folder.

After a brief pause, the right side of the image selector should display all of the bitmaps in that folder.

Scroll down until you see an image called "dialog.png".

Double click on the image.

2. Place Dialogue on Screen:

You now should be back at the game screen and your cursor would have changed to a little image icon informing you that you are now in **AddBitmap** mode. You will also notice that there is a rather large box following your cursor. This is the dimensions of your new image.

Place the image where you want it by clicking your left mouse button. You can reposition the dialogue later if you want. We want our about box to be roughly in the middle of the screen so move your mouse until the box is roughly centered and click the left mouse button.

If all is going well, you should see a new dialog box appear on screen and the code to create this new box in the Lua window on the left hand side.

Great! Let's not stop there.

3. Add Some Text:

Click on the **Text** button in the tool bar. Your cursor will change to the shape of a text tool to inform you that you

are in **AddText** mode.

We now need to tell FirstStage where we want this new text displayed. To do this, we need to click and drag out a box to the size of the new text dimensions.

All of our text will automatically wrap to the dimensions of this new box. Now, click and drag a box similar to this:

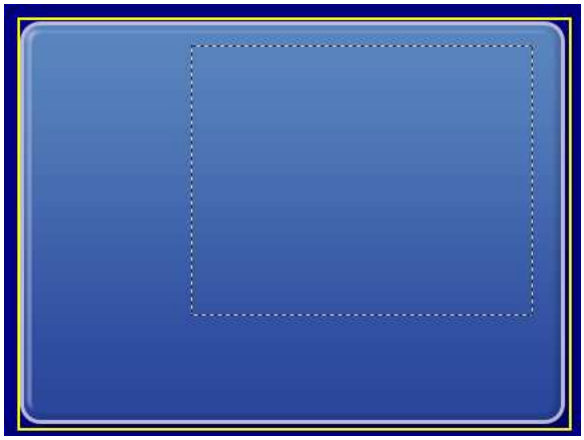


Figure 9.39: Dragging Out a Text Box

Once you release your mouse, a "Create Text Entry" dialog box will appear with lots of fields.

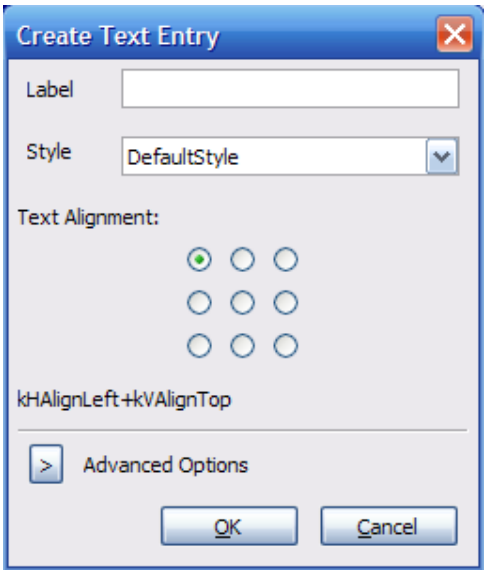


Figure 9.40: Text Properties Box

The "Label" is the text identifier that will be used to look up a string in the string table, which will be the actual string displayed on screen. Any text in a Playground game needs to be stored in the string table to make translation easier. Type "DemoAboutBoxText" into the label field.

The "text alignment" is how you want your text to be displayed. By default, it will be displayed in the top left corner of the text box. For this demo, we will use Top and Center so click on the top center button like so:

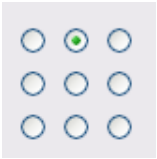


Figure 9.41: Text Alignment Grid

Now click OK.

After a pause, the Translations Editor dialog will appear. This dialog has all of the text that appears in your game. You never type text directly into Lua code because it won't be able to be translated into other languages. You can see that our new text entry label has been added to the bottom of the translations dialog. If you can't see it, try scrolling down the list of entries to the bottom of the list.

Double click on the translation cell (second column) for our new entry and replace the existing text with "Demo About Box" (no quotes) and press the enter key.

Now press the OK button to save your changes.

You will now be returned back to the game screen and you will see your new text displayed in the dialog.

4. Add an "OK" Button:

Onward we go. What would an about box be without an OK button, so that is next.

Click on the **Button** button in the tool bar. A "Create Button" dialog box will appear.

Type "OK" in the "Label" field, and choose "Close Modal" to tell Playground to close the dialog when this button is pressed.

Select the "Default" checkbox to make this button the Default. The "Default" button is the one that's pressed when the user hits Enter.

We'll leave the "Style" as "ButtonStyle". The "Style" field allows us to change the look of the button. "ButtonStyle" is the default style for buttons but you could choose a different one if appropriate. For this demo, we will leave it as "ButtonStyle".

The "Create Button" dialogue should now look like this:

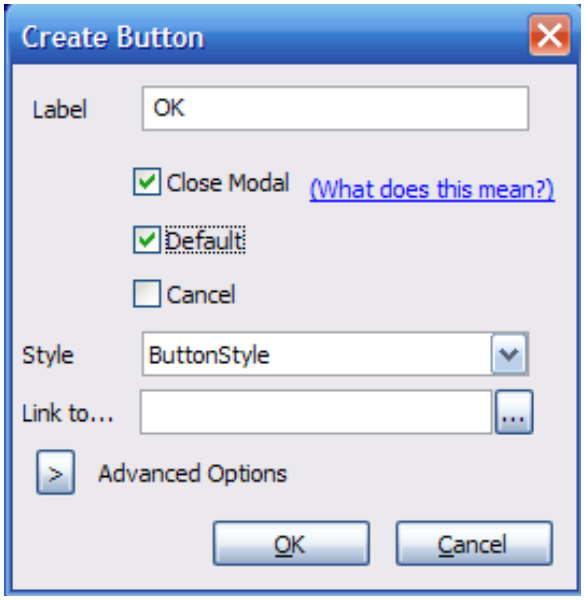


Figure 9.42: Button Properties Box

This button doesn't need to activate another dialog so we will leave the "Link to" option alone.

Click "OK" on the "Create Button" dialogue. That's it for the button options so press the OK button.

The cursor should change to be a button cursor which is informing you that you are in **AddButton** mode. Similar to adding a new bitmap, the button dimensions are shown with a box that follows the mouse.

Place our new button in the bottom left corner of the dialog box, like so:

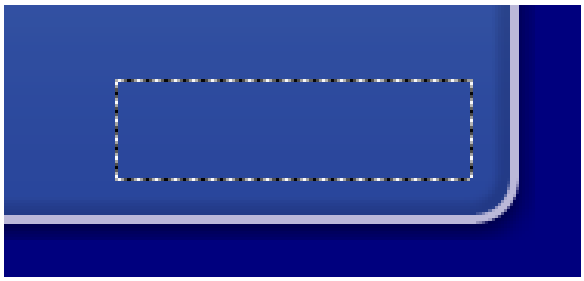


Figure 9.43: Button Dimensions

Like magic, your OK button should appear.

5. Add an Image Graphic:

Use your paint package to create a logo image. For this demo, we created a 128x128 pixel PNG file. Now save that new image into your assets/splash folder. It should be something like c:/firststagetest/skeleton/assets/splash.

Now switch back to FirstStage and click on the **Bitmap** button in the tool bar. When the image selector dialog appears, find the "splash" folder and select it. You should see your new logo as one of the images in the image selector.

Double click on your logo and you will be taken back to the game screen like before.

Now, this time instead of just plonking down the image at its default size, move your mouse to the top left area of the dialog then hold down your control key and click and drag out a box like if you were creating a text field. Notice how the box stays in the correct aspect for the image as you expand the box.

When the box is the size you want for the logo, release your mouse and your logo should appear there.

That should do it.

6. Save Your Work:

Press on the **Save** button on the tool bar and you will be prompted for a name for this Lua file.

Make sure you are in the "scripts" folder and save the Lua file to "myaboutbox.lua"

Displaying the About Dialog Box

Creating an about box is not much good if you can't see it running in game. What we will do is add a button on the Skeleton menu screen so that when you click the button, it will display our about box.

Click on the **Open** button on the tool bar in FirstStage.

Go into the scripts folder and find a file called "mainmenu.lua" and open it. After a pause, you should see the standard main menu screen for the Skeleton demo application. We need to add a new button on to the screen.

Click the **Button** button on the tool bar.

Type "about" into the Label field.

Make sure that none of the checkboxes are ticked.

For the Style, click on the arrow and select "LongButtonStyle"

Here's the interesting part. We want this button to display our new about dialog box that we created so we need to add a link to the lua script. Click on the [...] button on the right of the "Link to..." input box. A file requester will appear with lua files. Find our "myaboutbox.lua" file and double click it.

Our button properties window should now look like this:

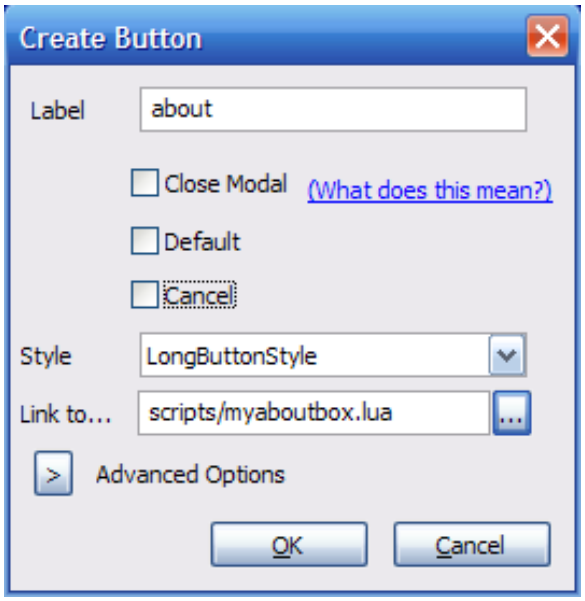


Figure 9.44: Adding a Menu Button

Press the OK button to create our button.

Position your mouse so that the box is above the "Change Player" button and click your mouse.

The translations editor should appear (because the "about" identifier is not in the translations file). We are happy with the default translation so click OK to save the translations file. The button should now appear on screen.

That's it! Save the file by pressing the **Save** button on the tool bar.

Now minimize FirstStage and go to the skeleton folder and run the file "playgroundskeleton.exe"

When the main menu appears, you should see your new button on screen. Click the button and your about box should appear on screen!

Well done! You have created your first dialog box.

9.3 FluidFX 2.0: Interactive Particle System Editor

Figure 9.45: FluidFX Logo

9.3.1 What is FluidFX?

FluidFX is an editor for particle systems specified in the Particle System Shader Language that is supported natively in Playground. It features:

- Colorized editing of Lua file.
- Editing of multiple particle systems at once, and displaying them together.
- Simplified editing of parameters to allow artists to edit and create their own custom particle systems.
- Dynamic data input stream that you can move around by simply dragging it with your mouse.
- Loadable background image.
- Displays the particles as you will see them in the game using the Playground SDK renderer.

9.3.2 How does FluidFX work?

The easiest way to explain how FluidFX works is by loading an existing file.

Run FluidFX, click Open, then find the Playground skeleton folder for some good examples in the assets/fx folder. Start out by loading fire.lua. The first thing you'll probably notice is that there's no Lua code visible on load–just a

set of sliders and other controls. The Lua code is still around–you just need to click on the "Script" tab to see it. But the point is that artists can actually use FluidFX to tweak the code without ever seeing the code–so a programmer can create a template particle system with the appropriate parameters and then let and artist run with it.

9.3.3 Creating Controls in FluidFX

So how do you create those controls? You can look at some examples and create them by hand, or you can have FluidFX create them for you. Start out by loading spinstars.lua, which doesn't have any embedded controls.

First, let's create our own copy by selecting "Save System As..." and giving it a new name. Now when we mess with it, we won't risk breaking the original file.

Next, let's create our first control: Gravity. On line 76, you can see the standard gravity equation. The relevant number here is "400", so select it with the mouse by double-clicking on it. Once the "400" is selected, you can right click on the selected text and you'll get a context menu; select "Slider."

In the slider dialog, fill in the values as you see below:

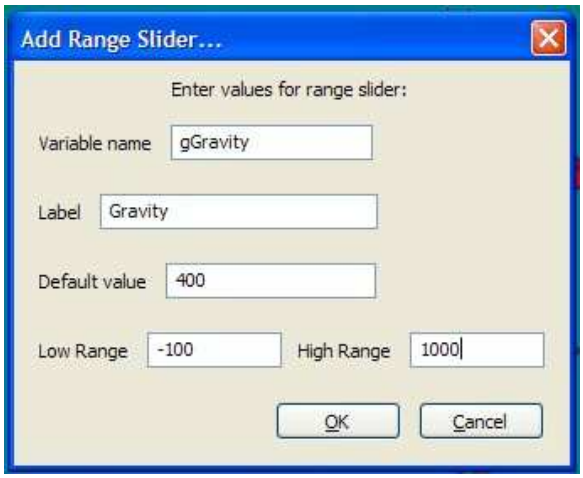


Figure 9.46: Slider Dialog

The first value is the variable name that's used in the script. Technically you don't need to change this, but it's certainly good practice. The **Label** field is the label that's displayed by the slider in the properties window. The default value is the value that the slider will be set to initially–more on that in a bit. Then the range that the slider will set the value to at each extreme. Press OK.

If you look, there's no slider yet on the properties tab, since the new script code that's been created hasn't been executed yet. Scroll to the top of the script pane for a moment to look at it.

Lua
```
gGravity = GUI_Slider(
   400,
   "Gravity",
   -100,1000);
```

That code is what, in FluidFX, will create a slider control. The variable, gGravity, will be modified as you move the slider. In fact, The first value, 400, is actually the value assigned to gGravity, and will be changed in the Lua file as you move the slider. Try it now to see what I mean: Hit the "Go" button, switch to the Properties tab, move the slider, and then switch back to the Script tab. The new value should be visible there in the script.

You can also create a multiple-selection control, and a texture-selection control. Examples of both of these are present in fire.lua, though the principle is the same for both.

For more information on writing new particle systems, see A Lua-Driven Particle System.

9.3.4 Creating Collections of Particle Systems

You can load more than one particle system into FluidFX by clicking the plus icon ("New System") in the Particle Systems window. You will have the option to select a particle system definition file.

When multiple particle systems exist, they all animate simultaneously. You can adjust their relative positions by selecting the "System" mode on the toolbar, and then selecting the system you want to move in the Particle Systems window. Select the "Global" mode to move all the systems at once.

When you have created several systems that you want to save together, you can use the "Save Collection..." option to create a particle system collection. A particle system collection is a Lua file that, when loaded into a TFxSprite, will load several systems at the same relative positions that you set up in FluidFX.

9.3.5 The Graph Control

There are many control types inside FluidFX but one of the most flexible is the graph control. With this control, you can create complex variations to variables over the time span of your particle.

To create a graph control, click on the Graph button in the toolbar which looks like this:

Figure 9.47: Graph Control Button

A dialog should appear like this one:

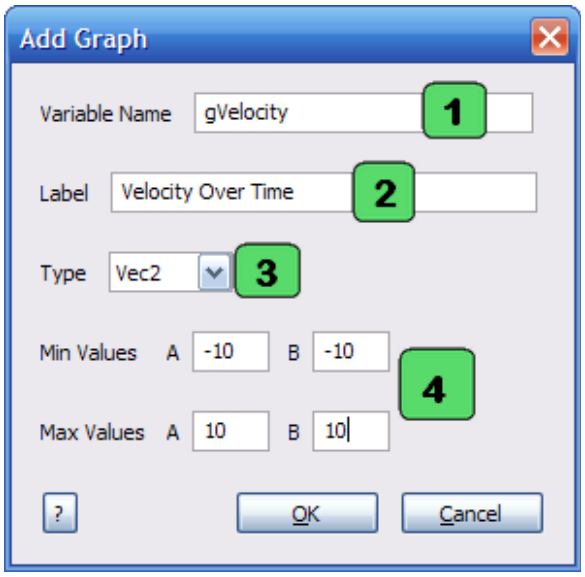


Figure 9.48: Graph Control Dialog

This dialog allows you to set up the default values and ranges for your new graph control.

(1) Variable Name is the name of the variable in the Lua code for this control.

(2) Label is the name that you will see in the properties window for this variable.

(3) Type is what type of variable this graph will display and edit. There are a few different types explained below:

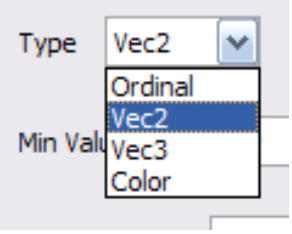


Figure 9.49: Variable Types

An ordinal type represents a single value range. For example 0 to 100 or -10.5 to 20.1. You would typically use an ordinal type for things such as the scale or rotation of the particle. A Vec2 type could be used for variables such as velocity where particles travel in both the x and y direction. A Vec3 type would allow variation on three axis which are common in 3D objects. A Color type is used for controlling the color of the particle. This makes it very easy to make linear blends between a series of colors and alpha values.

(4) The Min and Max values are the extents of the range that you want to set for this variable.

Once you have set up the values, click on the OK button to finish creating your new control and you will be taken back to the normal screen. If you look at the Lua code, you will see the new entry at the very top of the code. Now click on the properties tab and you will see something similar to this:

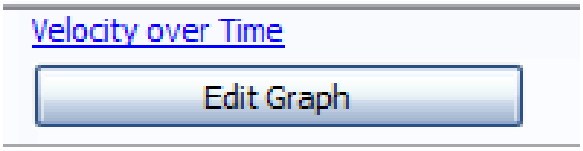


Figure 9.50: Graph Edit Button

The Edit Graph button will bring up a graph editing dialog box which will allow you to visually edit your graph. Press the Edit Graph button and the dialog should pop up.

9.3.6 The Graph Editor Dialog

If you are editing a variable type that isn't a color, you should see something similar to this dialog example:

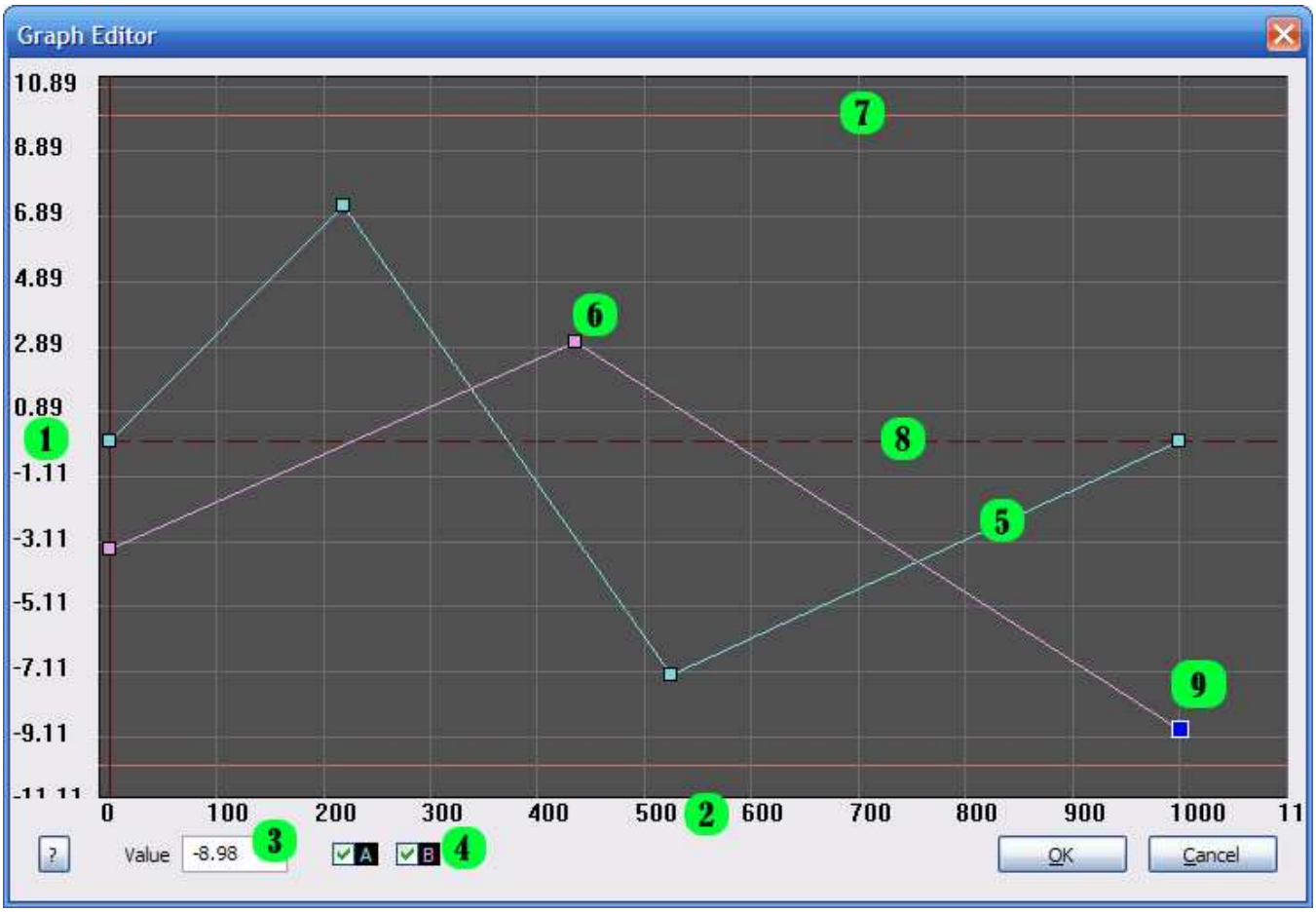


Figure 9.51: Graph Editor Dialog

This editor allows you to visially edit your graph values. The graph editor is very easy to use and comes with a variety of controls to help you edit your values.

(1) The numbers along the left hand side of the graph represent the values that you are editing. For instance, if you were editing the alpha value of a color, the numbers on the left hand side will range between 0.0 to 1.0.

(2) The numbers along the bottom represent time in milliseconds. So, the number 1000 equates to one second. Since you will typically be modifying what the particle will do over the life of the particle, this is how you specify what happens over that period of time.

(3) The value field is a numerical representation of the value of the currently selected node. Say for instance that you are editing the first node in your graph and you want the value to be 1.5 exactly, instead of trying to move the node to the correct value with your mouse, simply type 1.5 into the value box and press enter and the node will jump to the correct location in the graph.

(4) The check boxes allow you to show or hide individual axis of your graph. If you are editing a variable type of Vec2 or Vec3, sometimes it is very handy to hide the axis that you are not editing. The color of the letters, A, B, C are the same colors as the axis splines drawn in the graph window. A, B, and C can also be though of as X, Y, and Z of a Vec3.

(5) These lines represent the linear graduation of values over time. You can add as many points as you like by right clicking the mouse on the line where you want to add a point. Remember though that the more points you add, the more CPU time your particle system will need. It is always best to add just enough points for the level of precision that you need.

(6) These points or "nodes" are how you manipulate your graph. You click and drag them to change the node in time and value.

(7) The red lines at the top and bottom are the value extents that you set when creating your control. You will not be able to move the value of a node passed these lines.

(8) The dotted red line in the center is the "zero" point in your graph. Often, it is good to be able to tell where the value of zero is in relation to your graph.

(9) The current node that you are editing is displayed slightly different to the other nodes. It has a blue center and a white border whereas all other nodes are the same color as the spline with a black border.

If you are editing a "Color" value, the graph looks slightly different:

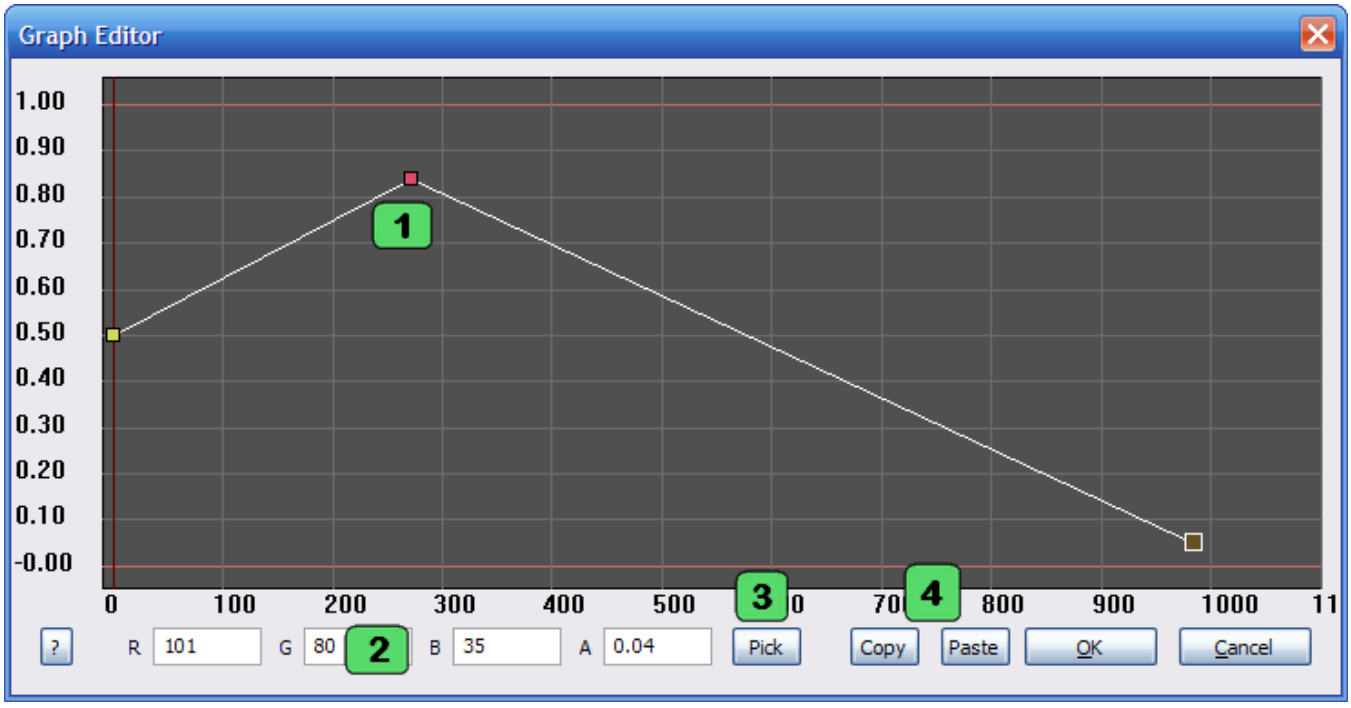


Figure 9.52: Graph Editor Dialog for Color

(1) First off, the nodes in a color graph are the RGB color that you want at that particular point in time. You can edit the color of a node by double clicking it with your mouse. The node's value (the number on the left hand side of the graph) represent the alpha channel of the color.

(2) Down the bottom of the graph, you can see the RGBA values listed. You can manually editor the current node's color be changing these values. R,G, and B are values between 0 to 255 where 0 is off and 255 is full on. The alpha value is between 0 and 1 where 0 is transparent and 1 is fully opaque.

(3) The Pick button brings up the color selector dialog. It is the same as double clicking a node.

(4) The copy and paste buttons allow you to copy the RGB component of one node and paste it into another. Simply select the node that you want to copy and then press the Copy button. Then, click on the node that you want to paste into and press the paste button. You could also press Ctrl-C and Ctrl-V instead of using the buttons.

There are some special keys and mouse actions that aren't obvious when you first look at the graph editor. They are:

Delete Key - You can delete a node by clicking on it and then pressing the delete key.

Ctrl-C and Ctrl-V - Allow you to copy and paste the RGB value of one node to another.

Home Key - Re-center the graph

Space Bar - Deselect the current node.

Backspace Key - Bring up the color requestor dialog for the current node.

With the mouse, you can also do the following.

Left Button - Use the left button to select nodes and while clicking on a node, hold down the left button and drag to drag a node to a new position.

Middle Button - Use the middle mouse button to pan the graph to a different location. Very useful when you have a graph that doesn't fit in the visible area of the screen.

Right Button - You use the right mouse button to create a new node on a spline. Just position the mouse on the part of the line where you want to make a new point and click the right mouse button.

Scroll Wheel - Use the scroll wheel to zoom in and out of the graph. Useful when you want finer control when adjusting values.

Ctrl + Scroll Wheel - Hold down the control key and use the mouse wheel to constrict the zooming to only the value axis. This is useful when you want to get finer control of your values but still want to see all of your graph in time.

9.4 sidewalk: Animation Creation and Asset Compression

The sidewalk command line tool has two distinct functions: Animation creation and asset compression.

When creating an animation, sidewalk takes a sequence of bitmaps as input and converts them to a combined animation that you can load using TAnimatedTexture. The result is an image (or a pair of images) and an XML file with the offsets necessary to reproduce the original sequence.

When using sidewalk to compress an assets folder for texture files, sidewalk will scan for textures in a folder and attempt to recompress them smaller, without losing any quality. Since Playground SDK™ will allow you to omit the file extensions in your source code, this recompression can be done to an assets folder without changing any source code, assuming that you followed the recommendation to omit those extensions.

9.4.1 Sidewalk Command Line Options

If you run sidewalk.exe with no parameters, you'll get the usage statement:

```
sidewalk v4.0.21.1TC
Usage 1/Create XML animation:
  sidewalk [options] animdesc.xml [ folder/fileroot* ]

Options:
  --mask        : Separate output into two layers, image and mask
  --opaque      : No transparency in PNG file.
  --reg=x,y     : Force registration point to x,y.
  --scale=#     : Scale an animation by #, where 1.0 means no change.
  --filter=type : Set the scaling filter type. Options are:
                        Box
                        Triangle
                        Hermite
                        Hanning
                        Hamming
                        Blackman
                        Gaussian
                        Quadratic
                        Cubic
                        Mitchell [default]
                        Lanczos
                        Bessel
  -8            : Output 8 bit png
  -f #          : 'fuzzy' bounding box value, between 0 and 1.
  -o            : Output individual frames as well as combined frames into
```

```
                'frames' folder.
  -p            : Reorder frames for optimal packing. This does not .
                  reorder the way frames are indexed, only reorders them in
                  the output file.
  -i            : Ignore size restrictions. This disables the requirement that
                  all images fit into a 1024x1024 texture. Note that if you use

                  this flag, the resulting image cannot be used directly to draw

                  in a game, but it may be used to hold data, etc.
  -v            : Verbose output
  --noclear     : Don't clear transparent pixels to adjacent color.


For a new animdesc.xml file, you need to specify the source file
wildcard or name.

Usage 2/Compress Images:
  sidewalk crunch [options] [path/to/search]

  defaults: --quality-threshold=70 --samplefactor=2 --output=crunched

Options:
  --nochanges        : Just walk through the tree and report what would be
                       done.
  --summary          : Provide simple summary.
  --fullcsv          : Provide detailed CSV report, including files that can't
                       be optimized.
  --csv              : Provide detailed CSV report of files that can be
                       optimized
  --jpegquality=##   : Set maximum JPEG quality: Any JPEG currently of higher
                       than this will be recompressed to the given quality.
                       Default is 85.
  --quality-threshold=## : Optimize conversions to ## quality. Range 0-100.
                       0=bad. 100=very good quality. good balance=80.
                       Note! This is a CPU intensive operation.
                       This flag and the --jpegquality flag are mutually exclusive.
  --samplefactor=##  : PNG8 resampling factor. Smaller for higher quality.
                       Valid values are [1-10].
  --splitmask        : Split PNG RGBA files into JPG/PNG files. As of
                       Playground 4.0.16, now automatically supported.
                       [default on]
  --nosplitmask      : Disable splitmask feature.
  --norecurse        : Don't recurse into subfolders.
  --splitonly        : Only split images to .jpg/.mask.png.
  --png8only         : Only compress images to png8.
  --jpgonly          : Only compress images to JPEG .jpg format.
  --jp2only          : Only compress images to JPEG2000 .jp2 format.
  --mirror           : Copy all files that do not need processing to output folder.
  --rough-png8       : Turn off quality testing of the png8 export.
  --output=folder    : Set output folder (for new images) to folder. Defaults
                       to "crunched".
  -v                 : Verbose output
```

9.4.2 Sidewalk Animation Creation

Example Animation Usage

To create a new animation, you need to start with a sequence of frames; PNG files are best, because they can have an embedded transparency layer. Note that if they are PNG files stored with an embedded transparency layer,

that layer needs to be non-empty or the resulting PNG file will be completely empty. They should also all be the same size; if your animation moves, the initial frame should be large enough to encompass the entire animation, unless you want the motion to be handled at runtime.

Once you have this sequence, save it in a folder by itself. The frames should alphabetize to be the correct sequence; if they're numbered, the numbers should have leading zeros (0001,0002, etc.) to ensure they come out in the right order if there are more than 10 frames.

9.4.3 Example Animation Usage

Let's say we want to create an animation out of a series of PNG files: frame00.png, frame01.png, etc. To convert these files to a new animation control file "myanim.xml", you would call:

```
C:\dust devil>sidewalk.exe myanim.xml frame*.png
Processing frame: frame00.png
Processing frame: frame01.png
Processing frame: frame03.png
Processing frame: frame04.png
Compositing frame: frame00.png
Compositing frame: frame01.png
Compositing frame: frame03.png
Compositing frame: frame04.png

C:\dust devil>dir myanim*
 Volume in drive C has no label.
 Volume Serial Number is C4A8-160F

 Directory of C:\dust devil

09/12/2006  03:42 PM             2,
270 myanim.png
09/12/2006  03:42 PM               497 myanim.xml
   2 File(s)          2,
767 bytes
```

You can see above that it created myanim.xml and myanim.png. You need to copy these to the same folder in your assets/ tree. All you need to do to load the file is reference myanim.xml, however–it knows what its external files are called.

Here we assume that both of the above files are copied to "assets/anim":

```
   TAnimatedTextureRef anim = TAnimatedTexture::Get("anim/myanim").
```
C++

And that's it, now we have an animation in the game.

Modifying Transparent Pixels

As of v4.0.14, sidewalk now modifies transparent pixels to match adjacent non-transparent colors. As transparent pixels shouldn't actually render, this may seem like a strange feature, but it turns out that these transparent pixels can "bleed" into their adjacent non-transparent pixels.

Remember that the renderer, when given a non-exact pixel location, will average adjacent pixels to determine what color values to draw at a particular location. Say you have a pixel that's completely transparent, with alpha value 0. It still has RGB values of course, but you don't intend them to render. Say the RGB values are 0,0,0, complete black. Also, let's assume that your object is white, RGBA 1,1,1,1.

Now when it attempts to render your object offset by 0.5 pixels, the first pixel it draws will be an average of the transparent pixel, 0,0,0,0, and the opaque one, 1,1,1,1. So you end up with 0.5,0.5,0.5,0.5. This sounds fine–unless the background you're rendering to is a very light color. Then your white object rendered to a white background will have a 50% transparent, 50% grey border.

Sidewalk corrects for this by taking that 0,0,0,0 and looking for an adjacent non-transparent pixel. It then takes the color from that pixel– in our case it would become 1,1,1,0. Now when it averages with its adjacent pixel, it will be come 1,1,1,0.5, which is really what you wanted.

If you don't want this behavior, the option –noclear will disable this clearing of transparent pixels.

9.4.4 The Sidewalk Crunch Command

Sidewalk can also be used to optimize your art folders to hopefully reduce the total amount of disk space that your game needs.

To use this feature, you supply the command "crunch" as the first parameter. There are many command line switches to go with this command that effect the output results. Some are self explanatory and the others will be explained below:

–jpegquality When Sidewalk finds a JPG file, it will firstly identify what quality setting the image was saved at. Then, if it is higher than what you specified as your –jpegquality setting, Sidewalk will recompress the image to the new setting. Make sure you don't specify to low a setting or your images will be blurry.

–jp2rate Same as –jpegquality but for JP2 files. The rate you specify is the percentage of the original size that you would like the image to be. The lower the number will mean that your JP2 could end up being distorted of blurry.

–splitmask This option will take a PNG file with an alpha channel and split it into a jpg file for the color component and a PNG grayscale image for the alpha component. Typically, this has much better savings than a straight PNG file with alpha channel.

–nosplitmask Because Sidewalk will try splitting PNG's by default, this option can turn off the feature if you do not require it.

–splitonly, –png8only, –jpgonly, –jp2only These switches force only a certain type of output. This is very handy when you use Sidewalk in a batch file where you know certain folders only need certain compression techniques.

–mirror This option will automatically copy any file that Sidewalk finds that isn't an image to the output folder. It also will copy any image file that isn't optimized by Sidewalk to the output folder. The result is that after Sidewalk has finished, you should have exactly the same amount of files in your output folder as you do in your input folder. This is great if you want to optimize your game folders and you don't want to have to copy the optimized images by hand over the top of the old files.

–rough-png8 By default, Sidewalk will try an optimization technique for PNG files by trying to convert them to PNG8 files. A PNG8 file tries to use 256 colors for the color component and 1 bit for the alpha component. This can be great for cartoon looking sprites. The downside is that if your original image has more than 256 colors, then PNG8 images can look rather scary. In this case, Sidewalk will automatically detect these bad conversions and reject the image. The –rough-png8 switch will turn off this check. This can be useful when you know certain images will be converted ok, even though they have reduced color quality.

–quality-threshold This switch will give Sidewalk control in determining which compression settings and format to use for the images. You supply a quality level that you are happy with and Sidewalk will try every compression trick it knows to find the best size for the image. This is of course a very time consuming task so be prepared to wait a while for a large folder of graphics to be completed.

9.5 Filmstrip: Creating and Editing Animations

Figure 9.53: Filmstrip

Filmstrip is Playground's animation frame sequencer. You can create new animations or load existing xml or anm animation files.

A variety of animations can be defined using the Lua scripting language to specify the name of the animation and the order of frames to play for that animation.

Also, for each frame, you can adjust a multitude of properties including rotation, scale, alpha and more.

Filmstrip has a group of dockable windows that allow you to control all aspects of the editor. Below is an explanation of each window:

9.5.1 Script Window

The script window holds the Lua script which defines the animations that make up the animation file. A typical script might look like this.

```
function DoAnim()
  delay(20);
    while true do
      frames{
        1,1,1,1,2,2,2,3,4,5,7,4,3,2
      };
    end
  end
```

For more information on how to control animation through Lua, see the "Animation Script Functions" section.

9.5.2 Animations Window

When you define new animations in the script window, the animation names will automatically be displayed in the Animations Window. To play an animation, either click on the name of the animation and press the play button in the window's toolbar, or just double click the animation name. To stop the animation from playing, press the stop button in the window's toolbar. You can advance one frame at a time by pressing the frame advance button and likewise play frames in reverse by pressing the reverse button in the toolbar.

9.5.3 Animation Frames Window

This window holds a list of frame numbers that make up the loaded sprite sheet. All animations are composed from these frames. To see a particular frame, click on the frame number in the list. You can also display multiple frames at once by clicking on the first frame in the sequence and then holding down shift and clicking the last frame in the sequence. On screen, you will see all the frames in the sequence being drawn at once but they will all be ghosted apart from the last frame. The ghost effect is called "onion skinning". More on onion skinning later.

The Animation Frames toolbar has some buttons which can help in modifying your animation frames.

Figure 9.54: Anim Frames Toolbar

From left to right,

- Frame Reverse - go back one frame in the animation
- Frame Advance - go forward one frame in the animation
- Delete Current Frame - Delete the selected frame number from the list.
- Insert Blank Frame - Insert a blank frame before the currently selected frame.
- Duplicate Frame - Insert a copy of the current frame. See below.
- Move Current Frame - Move the selected frame to a different location in the list. You will be presented with an input box. Specify the new location for the frame and press OK.

9.5.4 Duplicating Frames

You have the following options when duplicating a frame:

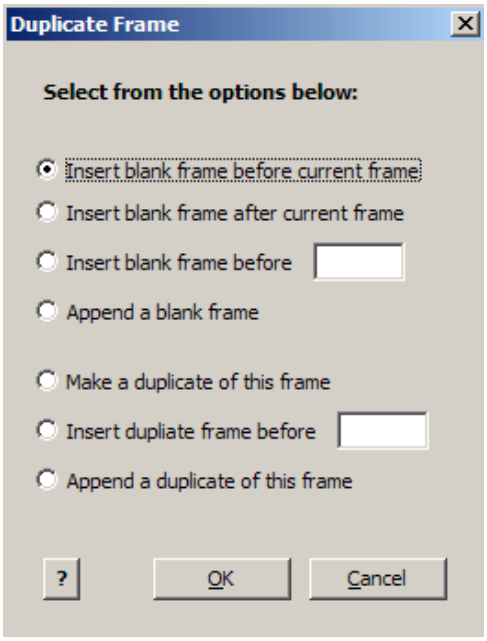


Figure 9.55: Duplicate Frames Dialog

- Insert Blank Before - Insert a blank frame before the currently selected frame.
- Insert Blank After - Insert a blank frame after the currently selected frame.
- Insert Before "x" - Create a blank key frame before the specified frame.
- Append Blank Frame - Add a blank frame to the end of the list of frames.
- Duplicate Frame - Make a copy and insert right after the current frame. You will end up with two exact frames together.
- Insert Duplicate Before "x" - Make a copy of the current frame and insert the new frame before frame "x".
- Append Duplicate - Make a copy of the current frame and add it to the end of the list of frames.

9.5.5 Anchors Window

When dealing with animations, anchors are a great way of retrieving reference points for each frame. See Anchors for more information on anchors. The Anchors Window allows you to manage your anchors. All of the anchors for the loaded animation are displayed as a tree list in the Anchors Window. If you click on the name of the anchor, it will select it. You can then use the Delete button in the toolbar to delete the anchor or press the Rename button to rename the anchor. There are a couple of ways to create new anchors. One way is to press the New Anchor button in the tool bar. It will pop up a dialog asking for a name for this new anchor.

9.5.6 History Window (Hidden by default)

The History Window displays all actions that have been performed inside the editor. You can either use Ctrl-Z and Ctrl-Y to undo and redo or you can click on any item in the history to revert back to when that action was created. This allows you to do a block of undos in one go.

9.5.7 Frame Inspector Window

This window gives you access to all of the editable properties for each frame on your sprite sheet. Click on any frame number in the Animation Frames window and you will see the properties for that frame updated in the Frame Inspector. It might look something like this:

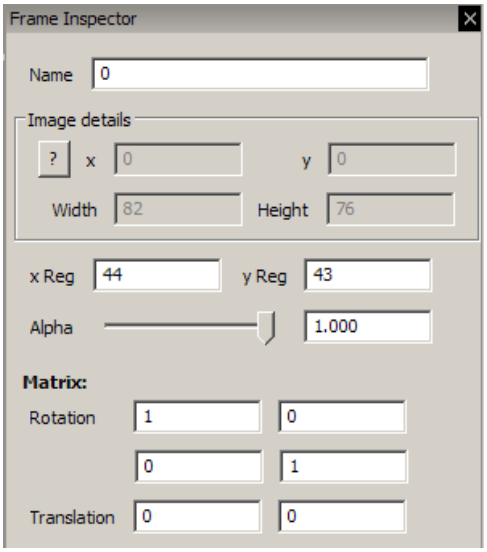


Figure 9.56: Frame Inspector

- The Name field can be used as an identifier for this frame. For instance, instead of frame 10 just being called 10, you could type in "WalkStart" and then in your Lua script, you could refer to this frame as WalkStart.
- The Image Details section is only for display purposes and shows the pixel offset information for this frame in the sprite sheet.
- The x Reg and y Reg fields are the registration values (or offsets) for this frame.
- The Alpha field represents the alpha level for this frame. You can adjust the alpha by using the slider or by typing in a value in the edit box.
- The next section is the matrix for this frame. It is made up of a rotation and translation section. Normally, you would adjust the matrix for a frame by using the Move, Rotate, and Scale tools rather than hand typing in values into the Inspector but if you want to enter exact numbers, you can use these matrix entries to do so.

9.5.8 The Main Tool Bar

The tool bar in Filmstrip has most of the common tasks that you will use when using the program. From left to right, these are:

Figure 9.57: Filmstrip Toolbar

New - Create a new sprite sheet from a series of images. See Creating a New Animation for more information.

Open - Open an XML animation or an ANM animation. ANM files are the binary version of the XML file which helps if you have a lot of data to load. If you want to create an ANM file, first load your XML animation and then choose File/Export to ANM from the main menu.

Save - Save the current animation file back to disk. If you want to save your animation to a different file, choose File/Save as from the main menu.

Undo/Redo - These buttons are the standard button to allow you to undo any mistakes you make.

Select/Move/Rotate/Scale - These buttons are "state" buttons and are used to manipulate the current frame of animation. If you click on either Move, Rotate, or Scale buttons, you will notice that the cursor will change to match the mode that you selected. The move button allows you to move the position of the current frame. Select the move button and then click and drag the left mouse button around the screen. You will notice the frame moves position. The rotate button will rotate the frame. Similar to move, press the rotate button and then click and drag the left mouse button left and right to rotate the image. The scale button will scale the image in both width and height. Click on the scale button and then click and drag the left mouse button to see the image scale. If you hold down the control key, the image will scale uniformly in width and height. Use the Select button when you don't want to make any more modifications to the frame.

Move/Rotate/Scale data entry - If you want to type in exact values for a particular mode, right click on the button and a data entry dialog will appear. Say, for instance, that you want to move the image ten pixels to the left. Right click on the Move tool button and a dialog will appear. Click on the Relative radio button because you want to move the position based on the current position. Type -10 in the x field and press OK. You will notice that the frame has jumped ten pixels to the left.

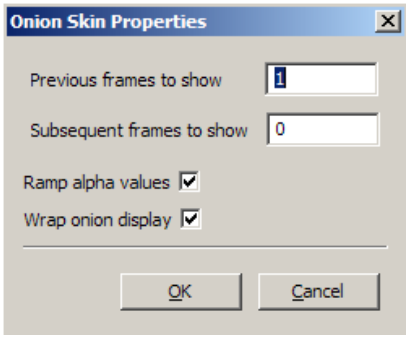


Figure 9.58: Onion Mode Settings Dialog

Onion - This toggles onion skinning mode. Onion skinning was a term adopted by traditional cell animators a long time ago and it means when you can see the last couple of frames of animation while you are editing the current frame. It is most helpful when you are trying to adjust details of the current frame and you need a reference of the last frame to compare it to. If you right click on the Onion toolbar button, you will get a dialog with settings you can adjust.

- Previous frames to show is how many frames you want to see before the current frame.
- Subsequent frames to show is how many frames to draw past the current frame.
- Ramp alpha values means that the previous and subsequent frames will have gradually less alpha the further they are away (in frames) from the current frame number.
- Wrap onion display means that the display of the frames will wrap around the end frames of your animation. For example, say you had an animation with 10 frames and you had your onion skin previous frames set to 5. If you were on frame 2, you would see frames 2,1,10,9,8, and 7. If you had Wrap turned off, you would only see frames 2 and 1.

Grid - Toggle frame grid view. The frame grid view will display all of the current frames in a 2D grid. If there are more frames than can fit on screen, move your mouse to the right of the game window and a vertical scroll bar will appear. Depending on the grid options set, you can also move the mouse to the bottom of the game window and a frame scale slider will appear which allows you to change the display scale for the frames. The current frame has a highlighted background and you can change the current frame by clicking on the cell of the frame you want.

The frame grid settings can be accessed by right clicking the grid tool bar button. Click Grid Settings for more information.

Bkg - This allows you to set a background image for the screen so that you can test your animations against a background.

9.5.9 Creating a New Animation

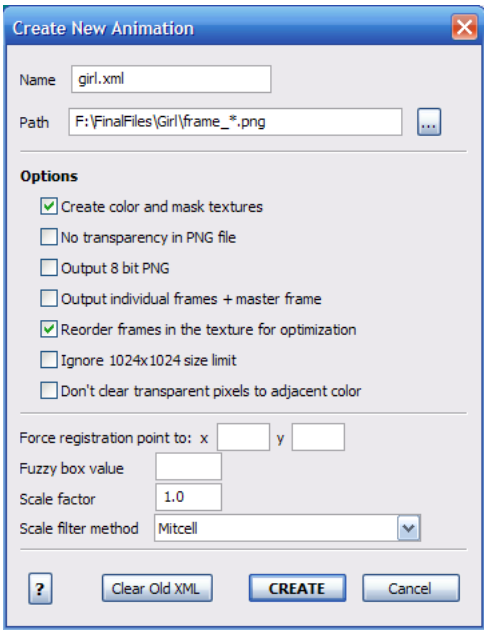


Figure 9.59: New Animation Dialog

Create a new sprite sheet from a series of images. The result will then be loaded into Filmstrip for you to add your animations. When you press New, it will bring up the Create New Animation dialog box which looks like this:

- Name is the name of the xml file to create on disk.
- Path is the path to one of the images that will be in the new animation. If you select the button to the right, you can browse to the folder with your images. Select any one of the images in the sequence you want to build and press Open.
- The Options are the same as the options in Sidewalk which is explained here sidewalk: Animation Creation and Asset Compression
- Clear Old XML button will delete any xml file of the same name before it begins compiling the new images. This is helpful if you want to make sure that the XML has been built from scratch rather than just updated.

- The Create button will start the build process. When you press the Create button, a window will appear on screen which is sidewalk being activated to build your new animation file. Once the process have been completed, the result will appear in the Output Window at the bottom of the screen.

9.5.10 Grid Settings

The frame grid settings can be accessed by right clicking the grid tool bar button. You will be presented with a dialog like this:

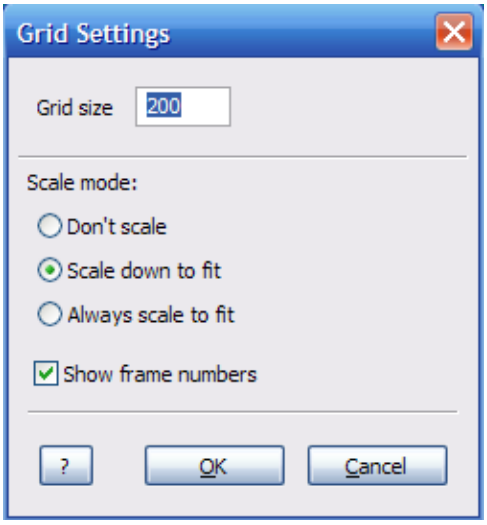


Figure 9.60: Grid Properties Dialog

Grid Size represents the width and height of each cell of the grid.

Scale Mode specifies how to scale each frame in the cell. The options are:

- **Don't Scale** Don't use any automatic scaling technique. The user can manually change the scale with the scale slider.
- **Scale Down to Fit** Reduce the size of the frame only if it doesn't fit inside the cell.
- **Always Scale to Fit** Always rescale the frame so that it fills the cell.

  Show Frame Numbers will display the frame number in the top left of each cell.

9.5.11 Anchors

Anchor points are used to track animated offsets; for instance, the hand of a character that needs to have an object attached might get an animated anchor. There are a couple of ways to create an anchor in Filmstrip. One way is to move your mouse where you want to create an anchor and then click the right mouse button. A menu will pop up. Select "Add anchor here" to create an anchor at that point. The other way to create an anchor is to use the "Add new Anchor" button in the toolbar of your anchors window. You will be prompted for the name of the new anchor. Enter a name and press OK. A new anchor is created and will be automatically located at the top left of the game screen. You will then need to move it to where you want on your animation.

To move an anchor, simply move your mouse until it is over the anchor and then click and drag with your left mouse button to the new location.

If you want to delete an anchor, you can either right click on the anchor and select "Delete this anchor" or you can select the anchor in the Anchors Window and press the delete button in the Anchors window toolbar.

When you move an anchor, it only moves the position of the anchor for the current frame you are on. If you want to move the anchor to a new position for ALL frames, hold down the ALT key while dragging your anchor.

9.5.12 Filmstrip and Sidewalk

Filmstrip uses the utility called Sidewalk (found in the Playground SDK bin folder) to create new animations. As such, the first time you want to create a new animation, you might need to direct Filmstrip to where your copy of sidewalk can be found. Filmstrip tries to do an intelligent search for a version of sidewalk, but if it can not find it you will be prompted to locate it yourself. Once Sidewalk has been located, you shouldn't be asked again. You can redirect Filmstrip to a different version of Sidewalk by using the menu item "Extra/Change Sidewalk location" in the main menu.

9.5.13 File Associations

If you develop under Windows XP, you can tell Filmstrip to associate all XML and ANM files with Filmstrip so that if you double click on an XML or ANM file in Windows Explorer, it will automatically run Filmstrip and load that animation. To tell Filmstrip to associate XML and ANM files, use the menu item "Extra/Associate XML/ANM files" from the main menu.

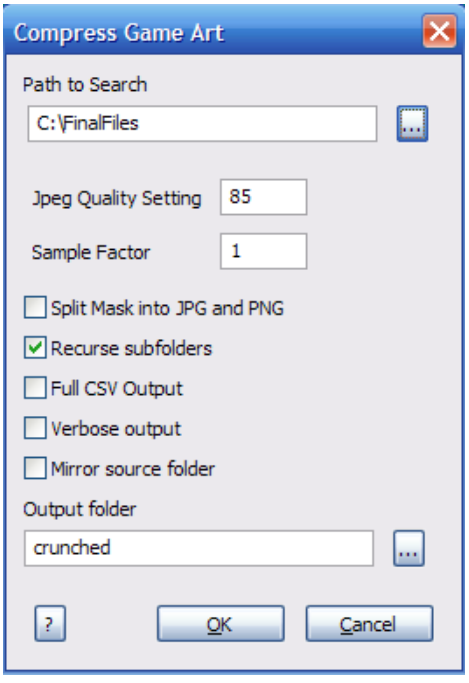


Figure 9.61: Compress Graphics Dialog

9.5.14 Compress "Crunch" Game Art

Filmstrip supplies a handy shortcut to the crunch command built into Sidewalk. All of the crunch features are selectable through the compress dialog found under the **Extras/Compress Graphics...** main menu

Refer to sidewalk: Animation Creation and Asset Compression for an explanation of the crunch options.

9.6 3dsconvert: Creating 3d Models For Playground

To create models for use in Playground, you convert .3ds files into a .mesh file. The utility for doing this is called 3dsconvert. Options for this utility are:

- -i [input]: name of .3ds file to convert.
- -id [inputdir]: name of directory to convert all .3ds files from.
- -o [outputfile]: name of file to output (default uses name of .3ds file) If there is more than one mesh in the file, and -m is not used, the files will be named [outputfile]1.mesh [outputfile]2.mesh ...
- -m: use names of .3ds meshes for output file name if both -m and -o are used, output names will be of form [outputfile]_[meshname].mesh.
- -od [outputdir]: name of directory to output files in.
- -s [amount]: how much to scale the model by (default 1.0).
- -center [xyz]: recenter across an axis - specify any combination of xyz. This happens before the yzflip.
- -invert [xyz]: invert across an axis - specify any combination of xyz. This happens before the yz flip, so -invert z first inverts the z axis, then flips it with y.
- -uvflip [uv]: This will flip texture coordinates from 0-1 to 1-0 - specify any combination of uv.
- -yzfilp: this will flip the y and z axis (default off)

9.7 Creating a Playground Font

To create a font you need the Macromedia® Flash® software. In the Playground SDK™ distribution is a bin folder, where you can find two different font template .fla files, one for Flash® MX and one for Flash® 8.

To create the font, first, from within the Flash® software:

1. Open the appropriate font template in your version of Flash®.
2. Select the text field and change it to be the font you want.
3. Publish the swf.

After you've completed the above, you need to drag and drop the resulting swf file onto the swf2mvec.exe program, also within the bin distribution. That program will emit an mvec file next to the swf file. Name that mvec file and add it to your assets folder, and then you can load it as a font.

9.8 axtool: Testing Your Game in a Browser

The axtool utility can help you configure your system to be able to run your game in an Internet Explorer browser on your system. Here are the basic steps you need to follow:

1. Create two files in your game build directory, activex.bat and activextest.bat, based on the example files included under the bin\axtool\samplefiles folder. Copy these files into your game folder and add the appropriate information for your game. You can generate the GUIDs using a tool that ships with the Microsoft® Developer Studio® development system called guidgen.exe, which is installed in the Common7\Tools folder in your Developer Studio installation folder.
2. Create an empty text file at c:\pfaxdebug.txt. Without this text file at the root of your C drive, the game will not run.
3. Open up a command prompt window (in Vista, ensure that you're running the command prompt as administrator).
4. Run the axtool.bat script that is in your utilities\bin\axtool folder. Run the utility like this:

```
axtool [gamefolder] test debug
```
C++

   - The gamefolder should be the path to your gamefolder (the folder that contains the activex.bat file, etc.)
   - **test** and **debug** are required parameters for testing the game locally.
   - Note that this script assumes you have Visual Studio® 2005 installed in the default location. If you do not, you can override where it looks for Visual Studio by defining a VSNETPATH environment variable to the path. For example:

```
set VSNETPATH="C:\\Program␣Files\\Microsoft␣Visual␣Studio␣2005"
```
C++

   or

```
set VSNETPATH="C:\\msdev\\install"
```
C++

After you've completed the above steps, you'll have an html file in your game folder. To test your game in a window:

1. Open up the html file in Internet Explorer (it will not run in other browsers).
2. Internet Explorer may ask you if you want to install the ActiveX® control; allow it to install.
3. A window will appear that will say something like "Run with -wnd=123456." This means your game is now ready to run in the window. You can now launch your game in the debugger, and instead of running in its own window, it will show up in the Internet Explorer window.

9.9 xml2anm: Convert XML to ANM

The ANM format was created to help speed the loading of animation files. As a binary format it can be loaded by Playground much more quickly than the more verbose and human readable XML format.

Currently you need to modify your game source to look for ANM files instead of XML files, so you'll need to perform the conversion in your development builds–it can't be done automatically at build time. We're considering changing this in a future release.

To use, you call it on the command line with one or more of the following options:

```
xml2anm: converts an animation XML file into a binary ANM file
flags:
    -f <filename> - converts one file from xml to anm
    -o <outputname> - if using -f, optional argument to specify outputfile
    -d <directoryname> - converts all xml files in directory to anm files
    -r - turns on recusrion for -d option
    -1 - writes old version 1 file
```

Chapter 10

Advanced Features

10.1 Playing With the Big Kids

This section has a number of quick examples and how-tos that cover more advanced concepts. Most Playground SDK™ games contain few or none of these techniques, but some problem domains have very simple solutions when you have the right tools. Here are some of the more sophisticated tools for you to use when the situations arise.

Some of the examples are very brief—after all, if you're reading this section, you're an expert looking for an example of an advanced technique. If you're not sure why you'd want to do some of these things, then you likely don't need to do them.

10.2 Deriving a Custom Sprite Type

As part of a discussion on the forums about attaching text to a sprite, it became obvious that one easy way would be to create a custom sprite type that, instead of (or in addition to) drawing a texture, also drew a line of text.

If you want your own custom-derived sprite, here's a quick approximation of what that code would need to look like. Our derived class will be called TTextSprite.

First a declaration:

C++

```
// Forward declaration
class TTextSprite ;

// Reference counted pointer wrapper
typedef shared_ptr<TTextSprite> TTextSpriteRef ;

class TTextSprite : public TSprite
{
   PFSHAREDTYPEDEF(TSprite);
protected:
   // Internal Constructor. Use TTextSprite::Create() to get a new sprite.
   TTextSprite(int32_t layer);

public:
   // Destructor.
   virtual ~TTextSprite();

   // Factory method
   static TTextSpriteRef Create(int32_t layer=0);
```

```
    // Our Draw call
    virtual void Draw(const TDrawSpec & drawSpec=TDrawSpec(), int32_t depth=-1);

private:
    TTextGraphic mTextGraphic ;
    TRect       mTextRect;
}
```

And then highlights from the implementation:

C++

```
PFTYPEIMPL(TTextSprite);

// Call the protected TSprite constructor
TTextSprite::TTextSprite( int32_t layer ) : TSprite( layer )
{
}

// Public creation call, to ensure all TTextSprites are wrapped
// in TTextSpriteRefs correctly
TTextSpriteRef TTextSprite::Create( int32_t layer )
{
    TTextSprite * as = new TTextSprite(layer);
    TTextSpriteRef ref( as );

    return ref ;
}

// The actual Draw call
void TTextSprite::Draw(const TDrawSpec & drawSpec, int32_t depth)
{
    if (!mEnabled)
    {
        return;
    }
    TDrawSpec localSpec = mDrawSpec.GetRelative(drawSpec);

    // This assumes you have an mTextRect which is the desired rectangle size
    // for your text.
    TRect rect = mTextRect+TPoint(localSpec.mMatrix[2].x,localSpec.mMatrix[2].y) ;
    mTextGraphic->Draw( rect, 1, 0, localSpec.mAlpha );

    TSprite::Draw(drawSpec,depth);
}
```

Note that I didn't include the math for extracting the rotation from the matrix, so this text will draw at the location but not the rotation of a sprite. I leave the extraction of the rotation as an exercise for the reader.

10.3 Sending Custom Application Messages

The TMessage type is intended to encapsulate any complex client message. The Playground SDK™ encapsulates TMessage as a Lua user-type, so you can pass a TMessage object around within Lua. If you want to have Lua pass complex message objects back to your C++ game, you can easily have a Lua object send a message to your game window where you can interpret its contents and trigger the correct game actions.

To start, derive your message type from TMessage and add any extra information to your derived class. For example, say you have a script that triggers an explosion animation that requires some additional processing in C++ code:

C++

```
// A custom user message with information about an explosion
class TriggerExplosion : public TMessage
{
    // This allows the class to have run time type id information
    PFTYPEDEF_DC(TriggerExplosion,TMessage);
```

```
public:
    int x,y;
    float size ;
};

...

// In a C++ file, define the runtime type id
PFTYPEIMPL(TriggerExplosion);
```

Now we have a new custom message type, and we need a way to create it from within Lua. Since we can use ScriptRegisterDirect() to expose a C++ function to Lua, we can create an instance in C++ and return it to Lua directly:

C++
```
// A function that creates a new explosion message
TMessage * NewExplosion( int x, int y, float size )
{
    TriggerExplosion * te = new TriggerExplosion;
    te.x = x ;
    te.y = y ;
    te.size = size ;
    te.mName ="TriggerExplosion";
    return te ;
}
```

The TMessage* will be deleted by the Lua garbage collection when it is no longer used, so you won't need to delete it yourself.

Here's how you might use the above code:

C++
```
// The main game class (excerpts of relevant parts)
class MainGameWindow
{
public:
    MainGameWindow()
    {
        // ...
        ScriptRegisterDirect(
            TWindowManager::GetInstance()->GetScript(),
            "NewExplosion",
            NewExplosion);

        // Make sure that messages are routed to us.
        TWindowManager::GetInstance()->GetTopModalWindow()->SetDefaultFocus(this);
        // ...
    }

    // ...
    virtual bool OnMessage(TMessage * message)
    {
        TriggerExplosion * te = message->GetCast<TriggerExplosion>() ;

        if (te) // this is a TriggerExplosion, so process it
        {
            x = te->x ;
            y = te->y ;
            size = te->size ;
            HandleExplosion(x,y,size) ; // This would do the extra work
            return true ;
        }

        // Check for other message types here...

        // ...

        return false ; // We didn't handle the message
    }
```

```
    // ...
}
```

Then, in a Lua script:

```Lua
    -- Send a message that indicates we want an explosion at this location
    PostMessage( NewExplosion(12,32,0.232) ) ;
```

This will send a message that will be delivered to the "default focus" window (see TModalWindow::SetDefaultFocus), and from there can be handled by your custom game window class. If you need a message to go to more than one destination, then you should set up a default focus window that can dispatch the message to the appropriate destination. The TMessage will be deleted by Lua in a normal garbage collection pass.

10.4 Calling a Lua Function from C++

For a simple example of calling a Lua function from C++,see the following.

```Lua
function DoSomething()
    DebugOut("I've been called");
end

Foo = { fn=DoSomething };
```

```C++
    TWindowManager::GetInstance()->GetScript()->RunScript("test.lua");
```

```C++
   TLuaTable * table = TWindowManager::GetInstance()->GetScript()->GetGlobalTable("Foo");
   TLuaFunction * fn = table->GetFunction("fn");
   fn->Call();
   delete fn;
   delete table;
```

This prints out "Lua: I've been called" in the debug log. You should be able to keep around the fn pointer indefinitely–I just delete it here since I'm creating it in a local pointer.

If your routine returns values, they will be pushed onto the stack in order–and this Call function only supports one result, so you'll only get the first one. You can use the TScript::PopString(), TScript::PopNumber(), or direct Lua C stack access functions to extract the return value from the stack.

10.5 Substituting a Custom TModalWindow

Here's a trick that's useful to replace TModalWindow as the default modal window that Lua's scripts push. Remember that this created and pushed every time your game calls DoModal(), DisplayDialog(), or even just PushModal(), so you probably want it to behave, in general, similarly to a TModalWindow.

To start with, create a class that derives from TModalWindow:

```C++
// In the header
class TMyModalWindow : public TModalWindow
{
    PFTYPEDEF(TMyModalWindow,TModalWindow);
    ...
}

// In the implementation file
PFTYPEIMPL(TMyModalWindow);
```

Then you need a piece of code that acts like the Lua version of PushModal():

```C++
str PushMyModal( str name )
{
    TMyModalWindow * window = new TMyModalWindow();
    window->SetName(name);
    TWindowManager::GetInstance()->PushModal(window);

    return window->GetID();
}
```

Once you have that function set up, you simply need to register it:

```C++
    ScriptRegisterDirect( TWindowManager::GetInstance()->GetScript(), "PushModal", PushMyModal );
```

Once you've done that, then every TModalWindow created by Lua will instead be one of your derived TModalWindow classes.

Like most of the advanced tips, this one comes with caveats. Be careful having your window draw anything–the current TModalWindow is intentionally invisible. And for obvious reasons, the class TDialog will not suddenly derive from your TModalWindow–and the function TWindowManager::DisplayDialog() creates a TDialog window, so any dialogs created using that function won't use your new TModalWindow.

The other reason this is in the advanced section is that it really would be a bad idea to abuse this trick: One might be tempted to swap in various kinds of TModalWindows depending on what game mode you were currently in, for example. And that might even work. But considering that Lua executes effectively as a "light" thread, and it's hard to know what its current state is at any one time–and things can synchronize differently depending on the speed of your computer. So use caution.

Part II

Reference

Chapter 11

Windowing Reference

11.1 Windowing and Widget Functionality

Collaboration diagram for Windowing and Widget Functionality:

11.1.1 Detailed Description

Everything related to windows and their descendents: buttons, sliders, image windows, and custom windows.

Modules

- Ignore This Group Please
  Interface for class T2dParticleRenderer.

Classes

- class TButton
  Encapsulation for button functionality.
- class TDialog
  A generic modal dialog.
- class TImage
  The TImage class is a TWindow that contains and draws a TTexture.
- class TLayeredWindow
  A TLayeredWindow is a TWindow with multiple layers which can be switched between.
- class TScreen

The base level modal window.

- class TText

  A text window.

- class TTextEdit

  The TTextEdit class represents an editable text TWindow.

- class TWindow

  The TWindow class is the base class of any object that needs to draw to the screen.

- class TWindowSpider

  A class used with TWindow::ForEachChild to iterate over the children of a window with a single "callback" function.

- class TWindowHoverHandler

  A callback that receives notification that a window has had the mouse hover over it.

- class TWindowManager

  The TWindowManager class manages, controls, and delegates messages to the window system.

Chapter 12

Lua Reference

12.1 Lua-Related Documentation

12.1.1 Detailed Description

Documentation on predefined Lua constants and functions, as well as C++ interfaces to Lua.
See the section on Lua Scripting for more information.

Files

- file luapluscd.h

  Playfirst-modified LuaPlus Call Dispatcher.

Classes

- class TLuaTable

  A wrapper for Lua table access in C++.

- class TLuaObjectWrapper

  Wrap a Lua object for use within C++ code.

- class TLuaFunction

  A wrapper for a Lua function.

- class TScript

  An encapsulation for a Lua script context.

- class TScriptCode

  An encapsulation of a compiled Lua source file.

Defines

- #define ScriptRegisterMemberFunctor(script, name, ptr, functor)

  Register a member function with the standard Lua signature.

- #define ScriptRegisterFunctor(script, name, functor)

  Register a function with the standard Lua signature.

- #define ScriptRegisterMemberDirect(script, name, ptr, directfunctor)

  Register a "Direct" called function: A function that will be called by Lua directly with appropriate parameters.

- #define ScriptRegisterDirect(script, name, directfunctor)

  Register a "Direct" called function: A function that will be called by Lua directly with appropriate parameters.

- #define ScriptUnregisterFunction(script, name)

  Unregister a function that was previously registered using ScriptRegisterDirect(), ScriptRegisterMemberDirect(), ScriptRegisterFunctor() or ScriptRegisterMemberFunctor().

Functions

- template<typename Func>
  void lua_pushdirectclosure (lua_State *L, Func func, unsigned int nupvalues)

  Push a function on the Lua stack that will be called "directly" with a custom parameter list and return value ("directly").

- template<typename Callee, typename Func>
  void lua_pushdirectclosure (lua_State *L, Callee *callee, Func func, unsigned int nupvalues)

  Push a function on the Lua stack that will be called "directly" with a custom parameter list and return value ("directly").

- template<typename Callee>
  void lua_pushfunctorclosure (lua_State *L, Callee *callee, int(Callee::*func)(lua_State *), unsigned int nupvalues)

  Push a member function on the Lua stack that will be called as a standard Lua callback function.

12.1.2 Define Documentation

#define ScriptRegisterDirect(script, name, directfunctor)

Register a "Direct" called function: A function that will be called by Lua directly with appropriate parameters.

This can be contrasted with ScriptRegisterMemberFunctor(), which calls a function with the standard Lua function signature.

Supported parameter types include:

- bool
- [unsigned] char
- [unsigned] [short] int (unsigned and short are optional)
- [unsigned] long

- lua_Number
- float
- const char*
- str
- const LuaNil&
- lua_CFunction
- TLuaTable * (see below for important notes)
- TMessage*
- const void* (return from Lua only)
- const LuaLightUserData& (return from Lua only)

For TLuaTable * as parameter, the pointer will exist for the duration of your function, but will be deleted in the next application event loop, so don't keep it around.

For TLuaTable * as a return type, you will need to return a TLuaTable pointer that persists past the end of the function; you are still responsible for deleting the pointer.

See also:

Using Lua to Script Your Game

#define ScriptRegisterFunctor(script, name, functor)

Register a function with the standard Lua signature.

The function signature should match:

int FN(lua_State * L);

Parameters:

script Script to add functor to.
name Lua name of function.
functor The function to call.

See also:

Using Lua to Script Your Game

#define ScriptRegisterMemberDirect(script, name, ptr, directfunctor)

Register a "Direct" called function: A function that will be called by Lua directly with appropriate parameters.

Parameters:

script Script to add functor to.
name Lua name of function.
ptr Pointer to "this" in the class we're binding to.
directfunctor The member function to call.

See ScriptRegisterDirect for supported parameter types.

See also:

ScriptRegisterDirect
Using Lua to Script Your Game

#define ScriptRegisterMemberFunctor(script, name, ptr, functor)

Register a member function with the standard Lua signature.

The function signature should match:

int FN(lua_State * L);

Parameters:

script Script to add functor to.
name Lua name of function.
ptr Pointer to "this" in the class we're binding to.
functor The member function to call.

See also:

Using Lua to Script Your Game

12.1.3 Function Documentation

template<typename Callee, typename Func> void lua_pushdirectclosure (lua_State * *L*, Callee * *callee*, Func *func*, unsigned int *nupvalues*)

Push a function on the Lua stack that will be called "directly" with a custom parameter list and return value ("directly").

More details are available in lua_pushdirectclosure(lua_State* L, Func func, unsigned int nupvalues)

See ScriptRegisterMemberDirect() for a simplified wrapper to this function.

Advanced user function; can be safely ignored by most users.

See the disclaimer on the LuaPlusCD.h description page.

Parameters:

L Lua state.
callee A pointer to the class instance you want to bind your caller to.
func Member function to call.
nupvalues Number of upvalues (usually 0; see Lua docs)

See also:

lua_pushdirectclosure(lua_State* L, Func func, unsigned int nupvalues)
ScriptRegisterMemberDirect

template<typename Func> void lua_pushdirectclosure (lua_State * *L*, Func *func*, unsigned int *nupvalues*)

Push a function on the Lua stack that will be called "directly" with a custom parameter list and return value ("directly").

See ScriptRegisterDirect() for a simplified wrapper macro.

Advanced user function; can be safely ignored by most users.

See the disclaimer on the LuaPlusCD.h description page.

With this function you can expose *any* class member function that uses the supported parameter and return value types to a Lua script.

See ScriptRegisterDirect() for supported parameter types.

Parameters:

L Lua state.
func Function to call.
nupvalues Number of upvalues (usually 0; see Lua docs)

See also:

ScriptRegisterDirect

template<typename Callee> void lua_pushfunctorclosure (lua_State * *L*, Callee * *callee*, int(Callee::*)(lua_State *) *func*, unsigned int *nupvalues*)

Push a member function on the Lua stack that will be called as a standard Lua callback function.

The standard function signature takes a parameter lua_State* and returns int.

This will allow you to create a member function and have Lua call it directly.

Advanced user function; can be safely ignored by most users.

See the disclaimer on the LuaPlusCD.h description page.

Parameters:

L Lua state.
callee A pointer to the class that your function is a member of.
func Function to call.
nupvalues Number of upvalues (usually 0; see Lua docs)

12.2 Query Values for Current Configuration in Lua.

12.2.1 Detailed Description

These values are passed to GetConfig() to query various configuration states.
See GetConfig() for more information.

Constants

- luaConstant kCheatMode = "cheatmode"
  Cheat mode enabled?
- luaConstant kComputerId = "computerid"
  This computer's unique ID.
- luaConstant kInstallKey = "installkey"
  The key that indicates how this game was installed.
- luaConstant kGameName = "gamename"
  The name of the game.
- luaConstant kGameVersion = "version"
  The version number of this build.
- luaConstant kEncryptionKey = "encryptionkey"
  The encryption key.
- luaConstant kHiscoreLocalOnly = "hiscorelocal"
  This is a local hiscore build.
- luaConstant kHiscoreAnonymous = "hiscoreanon"
  This is an anonymous hiscore build.

12.2.2 Constant Documentation

luaConstant kCheatMode = "cheatmode"

Cheat mode enabled?

luaConstant kHiscoreAnonymous = "hiscoreanon"

This is an anonymous hiscore build.

luaConstant kHiscoreLocalOnly = "hiscorelocal"

This is a local hiscore build.

12.3 GUI-Related Constants in Lua.

12.3.1 Detailed Description

These constants are available in the Lua GUI script.

Constants

- luaConstant kPush = 0
  Button Type: Push button.
- luaConstant kToggle = 1
  Button Type: Toggle button.
- luaConstant kRadio = 2
  Button Type: Radio button.
- luaConstant kAllLayers = -1
  SelectLayer() constant to select all layers for edit.
- luaConstant kCenter = 80000
  Position-relative-to-center: Add this constant to an x/y position to make it relative to a centered object.
- luaConstant kMax = 160000
  Width or position maximum.
- luaConstant kDefault = 128
  Button Text Alignment: Default for type of button.

12.3.2 Constant Documentation

luaConstant kCenter = 80000

Position-relative-to-center: Add this constant to an x/y position to make it relative to a centered object.

In other words, if you make x=kCenter, the object will be centered horizontally. If you add one (x=kCenter+1), the object will be one pixel to the right of where it would have been centered.

luaConstant kMax = 160000

Width or position maximum.

In the case of position, you can subtract an amount from kMax to make the position relative to the opposite edge. For a width or height, you can use -kMax to indicate the window should "grow" in the opposite direction of the x or y position: w=-kMax means that x specifies the right edge of the window, and that it should grow to the left edge of the parent.

12.4 Text and Window Alignment.

12.4.1 Detailed Description

These Lua constants are used both as text alignment in Text() windows, and general Window() alignment using the align tag.

Combine flags in Lua using normal addition (+), since Lua doesn't support bitwise or.

Constants

- luaConstant kHAlignLeft = 0
  Horizontal alignment: Left.
- luaConstant kHAlignCenter = 1
  Horizonal alignment: Center.
- luaConstant kHAlignRight = 2
  Horizonal alignment: Right.
- luaConstant kVAlignTop = 0
  Vertical alignment: Top.
- luaConstant kVAlignCenter = 4
  Vertical alignment: Center.
- luaConstant kVAlignBottom = 8
  Vertical alignment: Bottom.

12.5 Defined Message Types in Lua.

Constants

- luaConstant kGeneric = 0

  A generic message.

- luaConstant kCloseWindow = kGeneric+1

  A request to close a window.

- luaConstant kDefaultAction = kCloseWindow+1

  A request for the default window action.

- luaConstant kButtonPress = kDefaultAction+1

  A button was pressed.

- luaConstant kPressAnyKey = kButtonPress+1

  An "any key" was pressed.

- luaConstant kQuitNow = kPressAnyKey+1

  A request to terminate the application with prejudice.

- luaConstant kModalClosed = kQuitNow

  A notification that a modal window was closed.

- luaConstant kTextEditChanged = kModalClosed+1

  New information has been typed into/removed from a text edit field.

- luaConstant kCommandOnly = kTextEditChanged+1

  This message is empty; it's being sent to run the accompanying command.

- luaConstant kSliderValChanged = kCommandOnly+1

  A slider value changed.

- luaConstant kSliderMouseUp = kSliderValChanged+1

  The mouse has been released on a slider.

- luaConstant kSliderPageUp = kSliderMouseUp+1

  Someone clicked above the slider handle to create a virtual page-up.

- luaConstant kSliderPageDown = kSliderPageUp+1

  Someone clicked below the slider handle to create a virtual page-down.

- luaConstant kUserMessageBase = 1000

  First ID available for client applications.

12.5.1 Constant Documentation

luaConstant kCloseWindow = kGeneric+1

A request to close a window.
Name of message must be window ID to close (TWindow::GetID())

luaConstant kGeneric = 0

A generic message.

luaConstant kSliderMouseUp = kSliderValChanged+1

The mouse has been released on a slider.

luaConstant kSliderValChanged = kCommandOnly+1

A slider value changed.

luaConstant kTextEditChanged = kModalClosed+1

New information has been typed into/removed from a text edit field.

12.6 Animation Script Functions

12.6.1 Detailed Description

Supporting Lua functions for the animation system.

Functions

- function delay (ms)

  Set the delay time that will be used by showFrame.

- function pause (ms)

  Pause the animation for the given number of milliseconds.

- function showFrame ()

  Show the current frame and pause for the delay given in the last call to delay().

- function fadeout (numberOfFrames)

  Do an algorithmic fade-out over the number of frames given.

- function fadein (numberOfFrames)

  Do an algorithmic fade-in over the number of frames given.

- function setAlpha (a)

  Set the current alpha of this animation.

- function frames (framelist)

  Display the frames in the given list, once each, with the delay between frames given by delay().

- function loop (times, framelist)

  Loop the given frame list.

- function run (table)

  Run an animation in a different XML file: Loads the XML file and associated images and script, and then executes the given function.

- function die ()

  Kill this animation.

- function halt ()

  Pause the script indefinitely: The script and animation still exists, but won't be updated again until an external event changes its state to non-paused.

- function isPlaying (depth)

  Return true if a child is playing or paused.

- function stop (depth)

Stop this animation, and children animations as deep as the given depth parameter.

- function random (min, max)

  Return a random number between min and max inclusive.

12.6.2 Function Documentation

function delay (ms)

Set the delay time that will be used by showFrame.
Alternately you can call pause(ms) to pause for a given number of milliseconds.

Parameters:

ms Delay in Milliseconds

function die ()

Kill this animation.
The sprite still exists, but with no running script.

function fadein (numberOfFrames)

Do an algorithmic fade-in over the number of frames given.
Must call frames() with a framelist to make this work, since that's where the fade-out logic lives.

function fadeout (numberOfFrames)

Do an algorithmic fade-out over the number of frames given.
Must call frames() with a framelist to make this work, since that's where the fade-out logic lives.

function frames (framelist)

Display the frames in the given list, once each, with the delay between frames given by delay().

Parameters:

framelist A table with a list of frames.

function isPlaying (depth)

Return true if a child is playing or paused.
Won't search past the given depth; -1 means to keep searching.

Parameters:

depth Levels of children to test for playing.

function loop (times, framelist)

Loop the given frame list.

Parameters:

times Number of times to loop
framelist A table with a list of frames.

function pause (ms)

Pause the animation for the given number of milliseconds.

Parameters:

ms Delay in Milliseconds

function random (min, max)

Return a random number between min and max inclusive.

Parameters:

min The lowest number it might return.
max The highest number it might return.

function run (table)

Run an animation in a different XML file: Loads the XML file and associated images and script, and then executes the given function.

Expected call pattern is run{ "file.xml" } with optional named parameters

Parameters:

table A table consisting of a file to read and several optional parameters:

- function The name of a function to call to start the animations (default DoAnim)
- image The name of an image to override the default image.
- mask The name of a mask image to override the default mask image.

Example: run{ "file.xml", function="DoFirstAnim", image="usethisimage.png" };

function showFrame ()

Show the current frame and pause for the delay given in the last call to delay().

function stop (depth)

Stop this animation, and children animations as deep as the given depth parameter.

Parameters:

depth How deep to stop children.

12.7 Lua Particle System Reference

Collaboration diagram for Lua Particle System Reference:

12.7.1 Detailed Description

Supporting Lua functions for the particle system.
These functions and constants are available in any TLuaParticleSystem specification Lua script.
For more information about using the Lua Particle System, see A Lua-Driven Particle System.

Modules

- Blending Constants.
  These constants are only available in TLuaParticleSystem scripts.

Functions

- function fDistance (a, b)
  Return the distance between two points.
- function fOpenCycle (lower, upper, steps)
  Cycles the ouput value from low value to high value, in a given number of steps.
- function fClosedCycle (lower, upper, steps)
  Cycles the ouput value from low value to high value, in a given number of steps.
- function fAlpha (color, alpha)
  Replace (or set) the alpha value of a color.
- function fDebug (...)
  Write out debug info.
- function fFade (age, state1, time1, state2,{time2, state3,...})
  Interpolate a value between states over time.
- function fPick (value1{, value2...})
  Pick one of several values randomly.
- function fRange (value1, value2)
  Pick a random value in a range.
- function fExpire (value)

Kill a particle if the given test returns a non-zero value.

- function f2dRadius (radius{, center})

  Pick a random point on a circle with a given radius.

- function f2dRotation (radians)

  Calculate the x,y point on a unit circle corresponding to the given value in radians.

- function fAge ()

  Return elapsed time.

- function fTimeScale (value)

  Scale a value by elapsed time.

- function fLess (a, b)

  Test for a less than b.

- function fLessEqual (a, b)

  Test for a less than or equal to b.

- function fGreater (a, b)

  Test for a greater than b.

- function fGreaterEqual (a, b)

  Test for a greater than or equal to b.

- function GetParticleCount ()

  Get number of current active particles.

- function Done ()

  Signal to the Lua particle engine that this particle system considers itself to be done.

- function RandomFloat ()

  Return a random floating point value, 0-1.

- function RandomRange (low, high)

  Return a random value.

- function Allocate (size)

  Allocate a particle register.

- function CreateParticles (particles)

  Create particles.

- function Value (x)

  Wrap a scalar value with the proper Lua metatable.

- function Vec2 (x, y)

  Create a 2d vector.

- function Vec3 (x, y, z)
  Create a 3d vector.
- function Color (r, g, b, a)
  An RGBA color made up of floating point values from 0-1.
- function NewParticleGenerator ()
  Return a new particle-time calculator.

12.7.2 Function Documentation

function Allocate (size)

Allocate a particle register.

Parameters:

size (number) Number of floats in this register

Returns:

A particle register handle.

function CreateParticles (particles)

Create particles.

Takes a floating point value which is accumulated, so that if you tell it to create 1.5 particles twice, it will end up producing one particle the first time and two the second time.

Parameters:

particles (number) Number of particles to create; can be fractional.

function f2dRadius ()

Pick a random point on a circle with a given radius.

- radius The radius of the circle.
- center [optional] The center of the circle. Defaults to (0,0)
- Return a Vec2() point on the given circle.

function f2dRotation (radians)

Calculate the x,y point on a unit circle corresponding to the given value in radians.

- radians The point, on the circle, in radians.

- Return a point on the unit circle.

function fAge ()

Return elapsed time.

- Return elapsed time.

function fAlpha (color, alpha)

Replace (or set) the alpha value of a color.
color A color or Vec3(). alpha New alpha value. Return A color with the given alpha value.

function fClosedCycle (lower, upper, steps)

Cycles the ouput value from low value to high value, in a given number of steps.
In a "closed" cycle, the exact top value is reached on the last step.
Each time the function is called the next step in the cycle is returned.
For example:

```
fClosedCycle(0,1,3)
```
Lua

...will return, on successive calls: 0,0.5,1,0,0.5,1

```
fClosedCycle( Vec2(0,0), Vec2(1,1), 3)
```
Lua

...will return, on successive calls: (0,0),(0.5,0.5),(1.0,1.0),(0,0),(0.5,0.5),(1.0,1.0)

- lower Lower bound.
- upper Upper bound, which is never quite reached.
- steps Steps in the cycle.

function fDebug (...)

Write out debug info.
Any parameters given to this function will have their current values written out to the debug log. All parameters must evaluate to a particle value, i.e. a scalar, Vec2(), Vec3(), or Color().

Warning:

> Very slow–there is even overhead in a release build! You'll want to comment these out before shipping your game.

function fDistance (a, b)

Return the distance between two points.

"Points" can be from 1-4 dimensions.

- a First point (used to determine number of dimensions)
- b Second point
- Return scalar $\sqrt[2]{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2+(w_1-w_2)^2}$, where any dimensions not included in the given points are treated as zero.

function fExpire (value)

Kill a particle if the given test returns a non-zero value.

- value Value to test. If non-zero, then kill this particle.
- No Return Value

function fFade (age, state1, time1, state2)

Interpolate a value between states over time.

- state1..n The values to interpolate between.
- time1..n-1 The time to transition between stateN and stateN+1.
- Return an interpolated value. Value will have the same dimensions as input.

function fGreater (a, b)

Test for a greater than b.

- a First operand.
- b Second operand.
- Return true if a>b.

function fGreaterEqual (a, b)

Test for a greater than or equal to b.

- a First operand.
- b Second operand.
- Return true if a>=b.

function fLess (a, b)

Test for a less than b.

- a First operand.
- b Second operand.
- Return true if a<b.

function fLessEqual (a, b)

Test for a less than or equal to b.

- a First operand.
- b Second operand.
- Return true if a<=b.

function fOpenCycle (lower, upper, steps)

Cycles the ouput value from low value to high value, in a given number of steps.
In an "open" cycle, the exact top value is never actually returned.
Each time the function is called the next step in the cycle is returned.
For example:

```Lua
fOpenCycle(0,1,3)
```

...will return, on successive calls: 0,0.333,0.666,0,0.333,0.666

```Lua
fOpenCycle( Vec2(0,0), Vec2(1,1), 3)
```

...will return, on successive calls: (0,0),(0.333,0.333),(0.666,0.666),(0,0),(0.333,0.333),(0.666,0.666)
For example if you wanted to to spawn twenty particles that explode like a firework from a central point (dLocus):

```Lua
pPosition:Init( dLocus );
pVelocity:Init( f2dRotation( fOpenCycle( 0,6.28,32 ) ) );
pPosition:Anim( pPosition + fTimeScale(pVelocity) );
```

- lower Lower bound.
- upper Upper bound, which is never quite reached.
- steps Steps in the cycle.

function fPick ()

Pick one of several values randomly.

- value1..n The values to choose.

- Return One of the given values.

function fRange (value1, value2)

Pick a random value in a range.

The values can be any number of dimensions from 1-4. For dimensions greater than one, the result is calculated memberwise, so result[1] becomes a random value in the range value1[1]-value2[1], etc.

- value1 First bound.
- value2 Second bound.
- Return a value within the range of given values, with the same dimensionality.

function fTimeScale (value)

Scale a value by elapsed time.

Useful for adding a fractional velocity to position, e.g., so that velocity can be stored in pixels/second.

- value Value to scale, which is in units/second.
- Return a value in actual units.

function GetParticleCount ()

Get number of current active particles.

Returns:

The current number of particles active.

function NewParticleGenerator ()

Return a new particle-time calculator.

Returns a function that can be used as follows:

Lua

```
NewParticles = NewParticleGenerator()
PPS = 100 -- 100 particles per second
...
function Update( ms )
  newParticles= NewParticles( PPS, ms )
  CreateParticles( newParticles )
...
```

function RandomFloat ()

Return a random floating point value, 0-1.

Note that a value returned by this function will be generated once when the Lua code is executed–more than once if a Lua loop is involved, of course–but not once each time a particle is created or processed. In other words, any value returned by this function and passed into a particle function will remain constant for the duration of the particle system.

Returns:

A random value between zero and one.

function RandomRange (low, high)

Return a random value.

See the notes in RandomFloat() about random numbers.

Parameters:

low (number) Low end of range.
high (number) High end of range.

Returns:

A random value between low and high, inclusive.

function Value (x)

Wrap a scalar value with the proper Lua metatable.

Shouldn't be necessary for most isolated values, but can be useful if you want to have equations of the form Value(100)-pAge, since having 100-pAge won't work because of the way Lua resolves operator handling.

12.8 Lua GUI Command Reference

Lua GUI script function and constant documentation.

These functions and constants are available in the Lua script available from TWindowManager::GetScript().

Functions

- function Color (r, g, b, a)

  Return a color given r,g,b and optionally alpha.

- function FColor (r, g, b, a)

  Return a color given floating point r,g,b and optionally alpha.

- function DoWindow (window)
- function MakeDialog (dialogcommands)

  Create a dialog.

- function Window (table)

  Create a generic window.

- function Text (table)

  Create a TText window.

- function TextEdit (table)

  Create an editable text (TTextEdit) window.

- function StripKeyFromLabel (label)

  Search a label for & characters, and when found, return the label with the & removed, and the hot-key the & preceeded.

- function Button (window)

  Create a TButton.

- function Bitmap (table)

  Create a bitmap window (TImage).

- function EnableWindow (name, enable)

  Enable or disable a window by name.

- function BeginGroup ()

  Begin a group of radio buttons.

- function GetTag (tab, tag,...)

  Get a tag from the table or environment.

- function SetDefaultStyle (style)

  Set the current default style at a global level.

- function SetStyle (style)

  Set the default style within a window definition.

- function AppendStyle (style)
  Add a table of traits to the current style.
- function SetFocus (name)
  Set the focus to be window "name".
- function WaitForCloseMessage (id)
  Pause to wait for a particular modal window to close.
- function DoModal (fileToRead)
  Push a modal window onto the window stack.
- function ModalReturn (value)
  Return a value from a modal dialog.
- function DisplaySplash (splashMovie, splashGraphic, time)
  Display a splash movie or bitmap.
- function Yield ()
  Yield control to C++ code.
- function ReadFile (fileToRead)
  Read and run a Lua file in the assets folder.
- function CloseWindow (param)
  Ask the current level of modal window to close and return.
- function CustomCreator (s)
  A function to call with any custom window types, to allow FirstStage to parse the window description.
- function GetString (id, p1, p2, p3, p4, p5)
  Get a string from the string localization table.
- function CreateNamedMessage (type, name)
  Create a named message.
- function PostMessage (message)
  Post a message to the current message listeners, the current focus, or failing those options, the top modal window.
- function PostMessageToParent (message)
  Post a message to the parent of the current top modal window.
- function GetTimer ()
  Get the global timer value (TPlatform::Timer).
- function PushModal (name)
  Push a modal window onto the modal window stack.
- function GetLabel (name)

Get a label from a TText- or TTextEdit-derived window.

- function SetLabel (name, label)
  Set a label on a TText- or TTextEdit-derived window.
- function SetCommand (name, command)
  Set a command on a TButton-derived window.
- function GetButtonToggleState (name)
  Get the toggle state of a button.
- function SetButtonToggleState (name, state)
  Set the toggle state of a button
  - *name Name of the button to query*
  - *state True to set the button to "On" (TButton::SetOn), false for "Off".*
- function TagHotKey (hotkey, label)
  Find the first instance of the letter referred to in the given hotkey spec, and tag it in the label so that it comes out underlined.
- function SetBitmap (name, image, scale)
  Set an image on a TImage-derived window.
- function PopModal (r)
  Pop the a particular modal window NOW.
- function SwapToModal (name)
  Swap the contents of the top level modal window with the window elements contained in an external definition.
- function DisplayDialog (t)
  Present a modal dialog.
- function SelectLayer (layer)
  Select a layer in a TLayeredWindow (like a TButton).
- function Group (t)
  Group a set of windows together.
- function FitToChildren ()
  Cause a window to be resize to encompass current children.
- function Pause (time, waitForKey)
  Pause for a specified amount of time.
- function RegisterHotkey (hotkey, buttonName)
  Register a hot key with the current top modal window.

12.8.1 Function Documentation

function AppendStyle (style)

Add a table of traits to the current style.

```lua
-- Add two traits to the current active style
AppendStyle{ font="fonts/myfont.mvec", x=123 };
```

The added elements are only added locally; when another style is selected, the appended elements are discarded.

If you want to permanently change a named style, it's actually pretty easy: Styles are actually Lua tables, which are passed around as references, so modifications are always to the original table. So, to add a trait to a style MyDefaultStyle, you would just use the Lua member accessor:

```lua
MyDefaultStyle.font = "fonts/myfont.mvec";
```

...or...

```lua
MyDefaultStyle["font"] = "fonts/myfont.mvec";
```

Parameters:

style (table) A table that describes a style.

function BeginGroup ()

Begin a group of radio buttons.

Use before the first radio button in a group.

function Bitmap (table)

Create a bitmap window (TImage).

Remarks:

Supported tags include:

- alpha (boolean) True to force the image to have an alpha channel. (default=false)
- hflip (boolean) True to horizontally flip the image. (default=false)
- image (string) Name of the file to load.
- mask (string) Optional image mask (transparency layer) to apply.
- mipmap (boolean) True to force the image to be created with mipmaps. (default=false)
- rotate (boolean) Rotate the image by 90 degrees.
- scale (number) Scale to apply to the image. 1.0==normal image size. (default=1.0)
- vflip (boolean) True to vertically flip the image. (default=false)
- autoscale (boolean) True to scale image to screen size. Will NOT change aspect ratio of image if it is different from the screen. Ignored if scale parameter != 1. (default=false)
- Other generic window tags.

See also:

Window()

Parameters:

table (table) The table that describes the bitmap.

Returns:

A function that does the actual window creation.

function Button (window)

Create a TButton.

This function is defined, by default, in the style.lua file that is included with the Playground Skeleton application.

Remarks:

Supported tags include:

- beginGroup (boolean) This button is the first in a radio-button group.
- close (boolean) This button closes its window/dialog if true.
- command (function) The function to call when the button is clicked. During the function you can call GetButtonName() to determine the name of the calling button.
- default (boolean) Button is a default button.
- flags (number) Button label text alignment.
- graphics (table) An array of up to four images for the button: Three for push-buttons (Up, RollOver, Down), Four for toggle and radio buttons (Up, RollOver-Up, Down, RollOver-Down). If the array has fewer than 3 or 4 images, additional images are duplicated from the last given image.
- hflip (boolean) Horizontally flip the button images.
- label (string) Default button text label.
- mask (string) Specify a click mask for the button.
- on (boolean) True if the (toggle or radio) button should default to being "on".
- rotate (boolean) Rotate the button image by 90 degrees.
- sendToParent (boolean) Send any button message to the parent of the current top modal window.
- scale (number) Scale to apply to the button graphics.
- sound (string) Name of sound to play when button pressed.
- rolloversound (string) Name of sound to play when mouse rolls over the button.
- type (number) Button type (kPush, kToggle, kRadio).
- vflip (boolean) Vertically flip the button images.
- Other generic window tags.

Parameters:

window (table) The table that describes the button.

Returns:

A function that does the actual window creation.

See also:

Window()
BeginGroup()

function CloseWindow (param)

Ask the current level of modal window to close and return.

Parameters:

param (string or number) The value that should be returned from the DoModal() that spawned this window.

function Color (r, g, b, a)

Return a color given r,g,b and optionally alpha.

Values should range from 0-255.

Parameters:

r (number) Red value, 0-255.
g (number) Green value, 0-255.
b (number) Blue value, 0-255.
a (number) (optional) Alpha value, 0-255.

Returns:

A table that can be used to represent a color.

function CreateNamedMessage (type, name)

Create a named message.

Parameters:

type (number) The integer type of the message.
name (string) The name of the message.

Returns:

A TMessage * wrapped as a Lua object. Pass to PostMessage or PostMessageToParent.

See also:

PostMessage
kCloseWindow
kDefaultAction
kButtonPress
kPressAnyKey
kQuitNow
kModalClosed

function CustomCreator (s)

A function to call with any custom window types, to allow FirstStage to parse the window description.

Parameters:

s (string) Name of custom window creator.

function DisplayDialog (t)

Present a modal dialog.

Parameters:

t (table) A table, with a string as the first array element that names a Lua file that describes the dialog, and additional optional parameters to pass to the dialog. Parameters are passed in global gDialogTable.

function DisplaySplash (splashMovie, splashGraphic, time)

Display a splash movie or bitmap.

For more information on how to format your Flash content to display correctly on Windows and Mac, see TFlashHost.

Note: It is possible to have an overlay display on top of the splash screen bitmap (i.e. for a distributor logo). This can only be done on top of a bitmap, not a SWF. To do this, you should have a definition for a variable called "splashoverlay" inside your current default style. For example:

Lua
```
SplashOverlayStyle = {
  splashoverlay= Bitmap{ x=30, y=50, image="playfirstlogo" };
};
Then before you call DisplaySplash, call:
SetDefaultStyle(SplashOverlayStyle);
```

Note that you can also use the default style to control the overall scale of your splash screen graphic (i.e. scale it up or down):

Lua
```
SplashOverlayStyle = {
  splashoverlay= Bitmap{ x=30, y=50, image="playfirstlogo" };
  scale=1.3333333
};
```

There are three tags you can specify in the style before displaying a splash: disableAbort - (default is false) - if this is true, the user cannot click through the splash screen. translate - (default is false) - if this is true, the movie will trigger the Flash translation pipeline (see TFlashHost::Play). allowInput - (default is false) - if this is true, the flash movie will display the system cursor and accept mouse clicks.

Parameters:

splashMovie (string) Name of SWF file to play.
splashGraphic (string) Name of bitmap to display if Flash fails.
time (number) Time to show bitmap.

function DoModal (fileToRead)

Push a modal window onto the window stack.

Parameters:

fileToRead (string) The name of a Lua file to read that describes the window to push.

function DoWindow (window)

Note:

This function is for advanced users only.

Internal function that actually creates the window. This is the function that the script commands ultimately call to create the window.

Parameters:

window (table) Table that describes the window.

function EnableWindow (name, enable)

Enable or disable a window by name.

Parameters:

name (string) Name of the window to enable or disable.
enable (boolean) True to enable, false to disable window.

Returns:

True if window was found and enabled/disabled, false if no window by this name found.

function FColor (r, g, b, a)

Return a color given floating point r,g,b and optionally alpha.

Values should run from 0-1.

Parameters:

r (number) Red value, 0-1.
g (number) Green value, 0-1.
b (number) Blue value, 0-1.
a (number) (optional) Alpha value, 0-1.

Returns:

A table that can be used to represent a color.

function FitToChildren ()

Cause a window to be resize to encompass current children.

Intended to be used in a MakeDialog() context: Actually returns a function that, when called, will resize the top window on the stack.

function GetButtonToggleState (name)

Get the toggle state of a button.

Parameters:

name (string) Name of the button to query.

Returns:

True if the button is "On", false if "Off".

function GetLabel (name)

Get a label from a TText- or TTextEdit-derived window.

Parameters:

name (string) The name of the window to retrieve the label from.

function GetString (id, p1, p2, p3, p4, p5)

Get a string from the string localization table.

Parameters:

id (string) String to look up.
p1..5 (optional string) Optional parameters for string substitution. See TStringTable::GetString for more information.

Returns:

A string from the string table, or ##### if no string is found

function GetTag (tab, tag, ...)

Get a tag from the table or environment.

In normal usage, pass only two parameters.

Parameters:

tab (table) Table with window definition.
tag (string) The tag to query.

Returns:

The tag, either from the table or from the current style. Returns nil if tag isn't found.

function GetTimer ()

Get the global timer value (TPlatform::Timer).

Returns:

Number of milliseconds since the game was started.

function Group (t)

Group a set of windows together.

- t The table containing the windows to group.

Example

Code that creates two buttons based on a common name.

Lua

```
function TwoButtons( name )
      -- First create a table of the window parts using name
      t = {
         Button { name=name.."buttona", ... },
         Button { name=name.."buttonb", ... }
      };
      -- Return the table
      return Group(t);
end
...
-- Then use your function
MakeDialog
{
      Bitmap
      {
         x=0,y=0,
         TwoButtons( "test" ),
         ... -- The rest of the dialog definition can go here
      }
}
```

Parameters:

t (table) A table of window-creation command results.

function MakeDialog (dialogcommands)

Create a dialog.

Parses through a table (dialogcommands) which is made up of creator functions like Bitmap and Button.

The internal operation of MakeDialog assumes that actual function closures are in the table. Window creation functions like Bitmap() actually return a function closure that is added to the table.

Parameters:

dialogcommands (table) A table of window creation commands.

function ModalReturn (value)

Return a value from a modal dialog.

Parameters:

value (string or number) A value to return from a modal window.

function Pause (time, waitForKey)

Pause for a specified amount of time.

Parameters:

time (number) Number of milliseconds to wait.
waitForKey (optional boolean) True to abort on a keypress. Defaults to false.

function PopModal (r)

Pop the a particular modal window NOW.

Will pop that window and *any modal windows that were pushed on top of that window.*

Parameters:

r (string) Window ID or name to pop.

function PostMessage (message)

Post a message to the current message listeners, the current focus, or failing those options, the top modal window.

Parameters:

message (TMessage*) A message created with CreateNamedMessage or a user function that returns a TMessage *.

See also:

CreateNamedMessage

function PostMessageToParent (message)

Post a message to the parent of the current top modal window.

Parameters:

message (TMessage*) A message created with CreateNamedMessage or a user function that returns a TMessage *.

See also:

CreateNamedMessage

function PushModal (name)

Push a modal window onto the modal window stack.

Name the window.

- name (string) Name of the new modal window

Parameters:

name (string) The name of the new modal window.

Returns:

An id that can be passed to WaitForCloseMessage().

function ReadFile (fileToRead)

Read and run a Lua file in the assets folder.

Parameters:

fileToRead (string) Lua file to read.

function RegisterHotkey (hotkey, buttonName)

Register a hot key with the current top modal window.

When the key is pressed, it will act as if the button with the given name has been pressed.

Parameters:

hotkey (string) The key to tie to this button.
buttonName (string) The name of the button to press.

See also:

TModalWindow::AddHotKey()

function SelectLayer (layer)

Select a layer in a TLayeredWindow (like a TButton).

Parameters:

layer (number) The layer number to select.

function SetBitmap (name, image, scale)

Set an image on a TImage-derived window.

Parameters:

name (string) Name of the TImage window.
image (string) New image name.
scale (optional number) Optional scale value.

function SetButtonToggleState (name, state)

Set the toggle state of a button

- name Name of the button to query
- state True to set the button to "On" (TButton::SetOn), false for "Off".

Parameters:

name (string) Name of the button to query.
label (boolean) True to set the button to "On" (TButton::SetOn), false for "Off".

function SetCommand (name, command)

Set a command on a TButton-derived window.

Parameters:

name (string) The name of the button.
command (function) The new command to assign..

function SetDefaultStyle (style)

Set the current default style at a global level.

Do not call within a window definition.

Parameters:

style (table) A table that describes a style.

function SetFocus (name)

Set the focus to be window "name".

Parameters:

name (string) Name of window to receive the input focus. Usually a TextEdit window.

function SetLabel (name, label)

Set a label on a TText- or TTextEdit-derived window.

Parameters:

name (string) The name of the TText or TTextEdit window to modify.
label (string) The new label value to set.

function SetStyle (style)

Set the default style within a window definition.

The style is selected in the current layer only; after the closing brace of the current layer, the previously selected style will be restored.

A style is a Lua table with the following form:

Lua

```
MyStyle =
{
  parent=DefaultStyle, --- Optionally inherit from style table "DefaultStyle".
  tag=value,         --- Set tag to value. Any window tag can be set in a style.
  tag2=value2,       --- etc...
  tag3=value3,
};
```

When the window creation code searches for a tag, it first searches the window creation script table, then the current style, then parent of the style, and so forth until it finds the tag. Most tags have a default value that they fall back to when not defined.

Parameters:

style (table) A table that describes a style.

function StripKeyFromLabel (label)

Search a label for & characters, and when found, return the label with the & removed, and the hot-key the & preceeded.

Parameters:

label (string) The label to be parsed.

Returns:

cleanedLabel,key Two parameters are returned: the cleaned label and the specified key, assuming that the & preceeded a valid key character (low-ASCII).

function SwapToModal (name)

Swap the contents of the top level modal window with the window elements contained in an external definition.

Parameters:

name (string) Name of Lua file to load with new definition.

function TagHotKey (hotkey, label)

Find the first instance of the letter referred to in the given hotkey spec, and tag it in the label so that it comes out underlined.

Parameters:

hotkey (string) The key to find.
label (string) The label to add an underline tag to.

Returns:

The label with markup to underline the appropriate hot key.

function Text (table)

Create a TText window.

Remarks:

Supported tags include:

- flags Text alignment flags.
- defflags Default text alignment flags. Use these if no flags found.
- label Text to render.
- padding Line padding.
- rotation Degrees of rotation.
- rotationOriginX X coordinate of rotation origin.

- rotationOriginY Y coordinate of rotation origin.
- rotationAlign How to align rotated text; uses Text and Window Alignment..
- Other generic window tags.

Parameters:

table (table) The table that describes the control.

Returns:

A function that does the actual window creation.

See also:

Window()
GUI-Related Constants in Lua.

function TextEdit (table)

Create an editable text (TTextEdit) window.

Remarks:

Supported tags include:

- flags Text alignment flags.
- label Initial text.
- password A password field that should be shown as '*'s
- length Maximum length of editable field.
- ignore Characters to disallow in edit field.
- utf8 Allow all UTF-8 characters in edit field.
- Other generic window and text tags.

Parameters:

table (table) The table that describes the control.

Returns:

A function that does the actual window creation.

See also:

Text()
Window()
GUI-Related Constants in Lua.

function WaitForCloseMessage (id)

Pause to wait for a particular modal window to close.

Pass in the id that was returned from PushModal(), and when that modal window closes this routine will fall out.

Parameters:

id (string) The modal window id, returned from PushModal().

function Window (table)

Create a generic window.

Remarks:

Tags are Lua table entries that contain data that define how the window will be created. Tags that are not specified in a particular window table are sought in the current style and its parents.

Supported tags include:

- x, y Window position. Can be specified as an offset from kCenter or kMax.
- w, h Window width and height. Can be specified as an offset from kMax.
- name Window name (used as a handle to search for window and identify it at runtime).
- align Window alignment: A combination of horizontal and vertical alignment flags. See Text and Window Alignment.. Disables special processing of negative position values.

To center a window, specify its x or y position as kCenter, or specify align=kHAlignCenter or align=kVAlignCenter. To have a window fill its maximum width or height, specify kMax for that dimension. You can add offsets to either kCenter or kMax to specify a position relative to that logical anchor. See kCenter or kMax documentation for details.

Using kCenter will override any alignment flags for that axis.

Window position can be specified as negative, which indicates an offset from the opposite edge (similar to kMax-position). *Negative x and y use is deprecated, and will be removed from 4.1. This feature is disabled if you add an align tag to the window.*

Window width and height can be specified as negative, which indicates that x and y are describing the right or bottom edge of the window.

Parameters:

table (table) The table that describes the control.

Returns:

A function that does the actual window creation.

See also:

kCenter
kMax

function Yield ()

Yield control to C++ code.

Returns any parameters passed to Resume().

Returns:

Any values passed to Resume()

Chapter 13

Vertex Rendering Reference

13.1 Vertex Support for Triangle Rendering

13.1.1 Detailed Description

When rendering to triangles or lines using TRenderer::DrawVertices(), you need to set up your vertices using functionality provided in this section.

Classes

- struct TVert

  3d untransformed, unlit vertex.

- struct TLitVert

  3d untransformed, lit vertex.

- struct TTransformedLitVert

  2d Transformed and lit vertex.

- class TVertexSet

  A helper/wrapper for the Vertex Types which allows TRenderer::DrawVertices to identify the vertex type being passed in without making the vertex types polymorphic.

Functions

- void CreateVertsFromRect (const TVec2 &pos, TTransformedLitVert *vertices, TVec2 *corners, const TVec2 *uv, const TMat3 &matrix, float alpha, const TColor &tint)

  Create some vertices based on a rectangle and some transformation information.

13.1.2 Function Documentation

void CreateVertsFromRect (const TVec2 & *pos*, TTransformedLitVert * *vertices*, TVec2 * *corners*, const TVec2 * *uv*, const TMat3 & *matrix*, float *alpha*, const TColor & *tint*)

Create some vertices based on a rectangle and some transformation information.

This is a utility function that is used by several parts of the library internally, but has been exposed because of its general usefulness.

The resulting vertices can be passed to TRenderer::DrawVertices() with the rendering type of TRenderer::kDrawTriFan.

Parameters:

pos Position on screen to anchor rectangle.
vertices An array of four vertices to fill.
corners An array of four corners (first is upper left, then clockwise) relative to x,y that define the rectangle to create vertices for. Will be transformed by CreateVertsFromRect.
uv An array of two uv coordinates (upper left/lower right).
matrix A transformation matrix to apply to the rectangle.
alpha An alpha value to encode in the vertices.
tint A tint value to encode into the vertices.

Chapter 14

Font Reference

14.1 Introduction

The MVEC File Format is Playground SDK's font file format. Each glyph is stored as a series of vector line and curve drawing commands. Because of the vector representation, the glyphs can be smoothly scaled to many sizes.

Playground's font rendering is somewhat unconventional, but is designed to make it easy to get predictable results with a wide variety of fonts and when localizing games to many languages.

14.2 Quadratic Bezier Curves

MVEC's use quadric Bezier curves, which are defined by 3 control points. The **CURVETO** command uses p_0 as the current point, and takes additional points as data used as p_1 and p_2 . The path of the curve B(t) is defined by:

$$B(t) = (1-t)^2 p_0 + 2t(1-t)p_1 + t^2 p_2, t \in [0,1]$$

14.3 Data Types

The MVEC file format uses the following data types:

BYTE	1 byte
ULONG	4 bytes: a 32-bit unsigned long in little-endian byte order
USHORT	2 bytes: a 16-bit unsigned short in little-endian byte order
VALUE	2 bytes: a 16-bit signed short in little-endian byte order used to represent a real number. It's a 16-bit fixed point representation with 9-bits of integer and 7-bits of decimal. So, if x is float, then VALUE = (short)(x$*$128).

14.4 Vector Commands

The MVEC file format uses the following commands to control how a glyph is drawn. All command arguments are of type VALUE.

START (2 bytes)	Start the glyph	0x01 0x05
MOVETO (5 bytes)	Move the "pen" to an x, y coordinate	0x04 x y
LINETO (5 bytes)	Draw a line from the current pen position to an x, y coordinate	0x05 x y
CURVETO (9 bytes)	Draw a curved line (quadratic Bezier) from the current pen position to x2, y2 with x1, y1 as the control point	0x06 x1 y1 x2 y2
STOP (1 byte)	End the glyph	0x08

14.5 File Format Specification

Header (4 bytes)

```
MVC4
```

Glyph Count (2-bytes)

```
USHORT
UNUSED VALUES (6-bytes) reserved for future use
```

Glyph Table (18-bytes per glyph)
For each glyph in the font:

```
ULONG (Unicode value for the glyph)
VALUE (minimum X coordinate for the glyph)
VALUE (maximum X coordinate for the glyph)
VALUE (minimum Y coordinate for the glyph)
VALUE (maximum Y coordinate for the glyph)
VALUE (x value to advance before drawing the next glyph)
ULONG (offset from the beginning of the file for this glyph's vector commands)
```

The remainder of the file is the vector commands for each glyph.

Note: The "advance" value is similar to "kerning" but rather than it being the space between adjacent Glyphs, it's the space between the starting points for adjacent glyphs. MVEC currently does not support Kerning tables where character spacing is specialized based on which glyphs are neighboring each other.

14.6 Font Rasterization

When drawing a glyph, the vectors are rasterized and then filled in using an odd/even fill rule. That is, for each scan line, from left to right, the first scan line encountered starts the fill, the next scan line stops the fill, and so on.

14.7 Notes on Font Sizes and Line Height

Fonts are a whole world in and of themselves. There is a rich history and great reasons for a wealth of complexity. Playground SDK tries to simplify and make fonts practical from the perspective of a programmer who thinks about fonts as "text that fits in a rectangle".

As such, Playground SDK makes the notion of "line height" primary. All scaling issues are derived from that constraint. The concepts of "point size" and "leading" are therefore secondary. So in Playground SDK, the size at which a font is rendered is specified as its line height, where line height is the exact height in pixels needed to display a line of text at that size, including line-spacing. (Note a feature is provided to add additional line-spacing when desired for visual effect, but it is not necessary to use this for text to be adequately spaced).

Playground defines the "tallest" ascender as the glyph for the letter "W".

It defines the "deepest" descender as the deepest descender in the set of "low-ASCII" (UNICODE values < 128) excluding the glyph for underscore (glyph 95: _).

A line is then defined as follows, 15% of the line will be "above the W", and 5% of the line will be below the deepest descender (frequently this is "y"). Note that some novelty fonts don't have typical descenders (e.g. they are all caps style) — thus, the more algorithmic approach to selecting the deepest descender. In the other direction, due to more variability in styles, "W" proves to be a more reliable option.

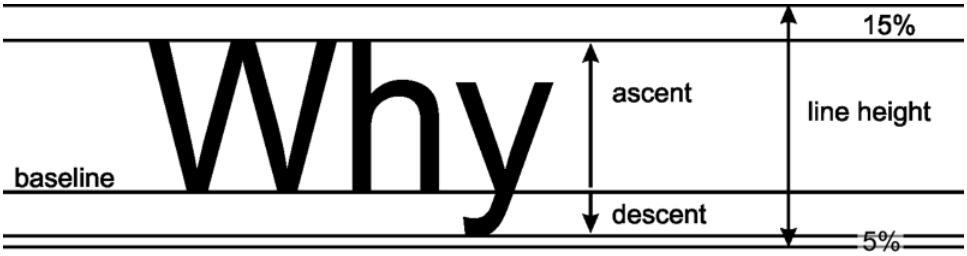


Figure 14.1: Font Example

Given this definition, a scale factor is selected to be applied to the font's glyphs for the requested line-height.

After these determinations are made, all glyphs in the font are analyzed. If a particular glyph would ascend past the top of the line, or descend past the bottom of the line, it is assigned a y-axis-only scale-factor that will "squish" it to fit.

This scaling approach shows an English language bias, but has been proven to yield beautifully localized games in languages including Western and Eastern European languages (e.g. Dutch, Russian) to Asian languages (e.g. Japanese, Korean) to Middle Eastern Languages (e.g. Arabic. Note that for Arabic, Playground has a string table pre-processing tool to handle UNISCRIBE issues for right-to-left and cursive texts.)

This scaling approach was derived experimentally, so it "just works" by taking into account things that were actually encountered across many fonts and many games. For example, the top margin is bigger than the bottom margin to better accommodate Western European extended glyphs, e.g., $\acute{A}$ or $\dot{E}$.

14.8 Call for Help

We'd LOVE it if someone wrote a direct TrueType to MVEC font conversion tool. :-) Especially one that allowed for easy select of different subsets of glyphs, e.g., Latin-1, Latin-1 + Korean, etc.

Currently, MVEC fonts are most easily created with the swf2mvec utility supplied with Playground. The Adobe SWF file format specifies font vectors in a very similar manner, so the swf2mvec utility is able to extract MVECs from swfs. However, a more direct conversion tool would be nice to have.

Chapter 15

Class and File Reference

15.1 str Class Reference

`#include <pf/str.h>`

15.1.1 Detailed Description

Reference-counted string class.

By default, str treats its string as a null-terminated string of ASCII characters. Sorting (collation) is handled, by default, in ASCII-value order. On the Windows platform, however, there's a way to insert an alternate string compare operator: If you add a DLL with the name pflocale.dll to the folder with the game, it will attempt to import a string compare function from that DLL named pfCollate() that takes two char* parameters and returns -1, 0, or 1, as per str::compare(). A UTF-8-aware comparison function is included in pflocale.dll in the bin folder of the distribution.

If you embed a null in a string using the str(char*,int) constructor, then some functions will not work as expected; str::length() in particular.

Public Types

- enum eFlags { kCaseInsensitive = 1, kReplaceAll = 2, kReverse = 4 }
  Flags for str::find and str::replace.

Public Member Functions

- str ()
  Create an empty string.
- str (const char *s)
  Create a string from a null-terminated string.
- str (const char *s, size_t len)
  Create a string from a buffer and length.

- str (const str &s)
  Copy constructor.
- ~str ()
  Destructor.
- int_fast8_t compare (const str &s) const
  Compare this string with another.
- uint32_t length () const
  Get the current string length.
- uint32_t size () const
  Get the current string length.
- bool empty () const
  Test whether the string is empty.
- str & assign (const char *s, size_t len)
  Assign a certain number of characters to str.
- str & append (const char *s, size_t len)
  Append a certain number of characters to str.
- str substr (uint32_t pos, int32_t length=npos) const
  Extract a substring.
- int32_t to_int () const
  Convert a string to an integer.
- TReal to_float () const
  Convert a string to a floating point value.
- void reserve (uint32_t size)
  Reserve at least size bytes in the internal string buffer.
- const char * c_str () const
  *Get a const char *.*
- void format (const char *formatstring,...)
  Format a string using a printf-style format string.
- int32_t find (str searchString, uint32_t flags=0, uint32_t start=0) const
  Find a substring.
- int32_t find (char searchChar, uint32_t flags=0, uint32_t start=0) const
  Find a character in this string.
- str & replace (str searchString, str replaceString, uint32_t flags=0, uint32_t start=0)

Search-and-replace a substring.

- void erase (uint32_t start, int32_t count=npos)
  Erase a range of characters in the string.
- void unique ()
  Force this instance of this string to be unique; prepare for modification.
- uint32_t overlay (int32_t start, const char *buffer, uint32_t count, bool bTerminate=true)
  Overlay a string into the current string.
- void downcase ()
  Convert this string to lower-case in place.
- unsigned int find_first_of (str s, unsigned int start=0)
  Find the first character that matches the set given by s.
- unsigned int find_first_not_of (str s, unsigned int start=0)
  Find the first character that doesn't match any in the set given by s.

Operator Definitions

- const char operator[] (uint32_t i) const
  Read-only character access.
- str & operator= (const str &s)
  Assignment operator.
- str & operator= (const char *p)
  Assignment operator.
- str operator+ (const str &) const
  Concatenation.
- str operator+ (char) const
  Concatenation.
- bool operator== (const str &) const
  Equality/inequality.
- bool **operator**< (const str &) const
- bool **operator**> (const str &) const
- bool **operator**<**=** (const str &) const
- bool **operator**>**=** (const str &) const
- bool operator!= (const str &s) const
  Equality/inequality.
- str & operator+= (const str &s)
  Concatenation.
- str & **operator+=** (const char c)

Static Public Member Functions

- static str dupchar (uint32_t number, char c=' ')
  Create a string consisting of a number of identical characters.
- static str getFormatted (const char *formatstring,...)
  Create a formatted string using a printf-style format.
- static str getFormattedV (const char *formatstring, va_list va)
  Create a formatted string using a printf-style format.
- static int sizeof_utf8_char (const char *s)
  Calculate the size of a UTF8 character.
- static uint32_t utf8length (const char *p)
  Calculate the length, in characters, of a UTF-8 string.
- static str from_utf32 (uint32_t utf)
  Convert a UTF-32 character code point to UTF-8.
- static uint32_t to_utf32 (const char *utf8)
  Convert a UTF-8 character to a UTF-32 character.

Static Public Attributes

- static const int32_t npos = -1
  Non-position in string. Similar to STL.

Classes

- class **TStringData**

15.1.2 Member Enumeration Documentation

enum str::eFlags

Flags for str::find and str::replace.

Enumerator:

kCaseInsensitive Case insensitive search.
kReplaceAll Replace all instead of just the first match.
kReverse Search starting from end of string and working backwards. start in this case means characters from end of string.

15.1.3 Constructor & Destructor Documentation

str::str (const char * *s*)

Create a string from a null-terminated string.

Parameters:

s Pointer to a null terminated string.

str::str (const char * *s*, size_t *len*)

Create a string from a buffer and length.

Embedded zeros in the source buffer will cause length() to return shorter than len.

Parameters:

s Pointer to a buffer.
len Length of buffer. String will be created at this length.

str::str (const str & *s*)

Copy constructor.

Parameters:

s String to copy and add a reference to.

15.1.4 Member Function Documentation

int_fast8_t str::compare (const str & *s*) const

Compare this string with another.

Parameters:

s String to compare with.

Returns:

Similar to strcmp

- -1 when this string is less than other string
- 0 when strings are equal
- 1 when this string is greater than other string

str& str::operator= (const str & *s*)

Assignment operator.

Parameters:

s Source string.

Returns:

A reference to this string.

str& str::operator= (const char * *p*)

Assignment operator.

Parameters:

p Source char * (c-style string).

Returns:

A reference to this string.

bool str::operator!= (const str & *s*) const

Equality/inequality.

Parameters:

s String to compare against.

uint32_t str::length () const

Get the current string length.

Returns:

Length of string not counting null character.

uint32_t str::size () const

Get the current string length.

Returns:

Length of string not counting null character.

bool str::empty () const

Test whether the string is empty.

Returns:

True if empty.

str& str::assign (const char * *s*, size_t *len*)

Assign a certain number of characters to str.

Parameters:

s Base of string to copy.
len Number of characters.

Returns:

A reference to this.

str& str::append (const char * *s*, size_t *len*)

Append a certain number of characters to str.

Parameters:

s Base of string to copy.
len Number of characters.

Returns:

A reference to this.

str str::substr (uint32_t *pos*, int32_t *length* = npos) const

Extract a substring.

Parameters:

pos Position to start extracting.
length Number of characters to extract.

Returns:

A new string with the specified characters.

int32_t str::to_int () const

Convert a string to an integer.

Returns:

An integer conversion of the string. If the string begins with non-digits, returns 0.

TReal str::to_float () const

Convert a string to a floating point value.

Returns:

A float conversion of the string. If the string begins with non-digits, returns 0.

void str::reserve (uint32_t *size*)

Reserve at least size bytes in the internal string buffer.

Parameters:

size Number of bytes to reserve.

const char* str::c_str () const

Get a const char *.

Returns:

A const char * to the internal data.

Referenced by TScript::PopBool().

void str::format (const char * *formatstring*, ...)

Format a string using a printf-style format string.

Warning:

Strings for %s must be passed as char* arguments!

Parameters:

formatstring Format string.

static str str::dupchar (uint32_t *number*, char *c* = ' ') [static]

Create a string consisting of a number of identical characters.

Parameters:

number Number of characters.
c Character to duplicate.

Returns:

A str with the requested duplicated characters.

static str str::getFormatted (const char * *formatstring*, ...) [static]

Create a formatted string using a printf-style format.

Warning:

Strings for %s must be passed as char* arguments!

Parameters:

formatstring Format string.

Referenced by TWindow::GetID().

static str str::getFormattedV (const char * *formatstring*, va_list *va*) [static]

Create a formatted string using a printf-style format.

Warning:

Strings for %s must be passed as char* arguments!

Parameters:

formatstring Format string.
va var-args argument list.

int32_t str::find (str *searchString*, uint32_t *flags* = 0, uint32_t *start* = 0) const

Find a substring.

Parameters:

searchString String to find.
flags eFlags for options.
start Search start.

Returns:

An offset into the string where found; npos if not found.

int32_t str::find (char *searchChar*, uint32_t *flags* = 0, uint32_t *start* = 0) const

Find a character in this string.

Parameters:

searchChar Character to search for.
flags eFlags for options.
start Start search position.

Returns:

An offset into the string where found; npos if not found.

str& str::replace (str *searchString*, str *replaceString*, uint32_t *flags* = 0, uint32_t *start* = 0)

Search-and-replace a substring.

Parameters:

searchString String to find.
replaceString String to replace found string with (can be empty).
flags eFlags for options.
start Search start.

Returns:

A reference to this str.

void str::erase (uint32_t *start*, int32_t *count* = npos)

Erase a range of characters in the string.

Parameters:

start First character to erase.
count Number of characters to erase. npos for the rest of the string.

uint32_t str::overlay (int32_t *start*, const char * *buffer*, uint32_t *count*, bool *bTerminate* = `true`)

Overlay a string into the current string.

Parameters:

start Start of the new overlay as an index into the current string. Can be (or extend) beyond the end of the string. Can also be npos, to indicate the end of the string.
buffer String to overlay. Is 8-bit safe (can contain NULL bytes).
count Size of string to overlay.
bTerminate True to add a NULL character in the str at the end of this overlay.

Returns:

The character index past the end of the overlayed characters.

unsigned int str::find_first_of (str *s*, unsigned int *start* = `0`)

Find the first character that matches the set given by s.

Parameters:

s Set of characters to search for.
start First character to inspect.

Returns:

The offset of the first character that matches, or str::length() no characters match.

unsigned int str::find_first_not_of (str *s*, unsigned int *start* = `0`)

Find the first character that doesn't match any in the set given by s.

Parameters:

s Set of characters to compare with.
start First character to inspect.

Returns:

The offset of the first character that does not match, or str::length() if all characters match.

static int str::sizeof_utf8_char (const char * *s*) `[static]`

Calculate the size of a UTF8 character.

Parameters:

s Pointer to the character.

Returns:

Number of bytes in this character. Zero if the character is a null terminator.

Referenced by utf8length().

static uint32_t str::utf8length (const char * *p*) `[static]`

Calculate the length, in characters, of a UTF-8 string.

Parameters:

p Pointer to const char * string of UTF-8 characters.

Returns:

The number of UTF-8 characters in a string.

References sizeof_utf8_char().

static str str::from_utf32 (uint32_t *utf*) `[static]`

Convert a UTF-32 character code point to UTF-8.

Parameters:

utf UTF-32 value.

Returns:

A str with a single UTF-8 character.

static uint32_t str::to_utf32 (const char * *utf8*) `[static]`

Convert a UTF-8 character to a UTF-32 character.

Parameters:

utf8 Pointer to a 1-4 byte UTF-8 character.

Returns:

A UTF-32 character.

15.2 T2dParticle Class Reference

`#include <pf/2dparticlerenderer.h>`

15.2.1 Detailed Description

Basic Particle Values.

See also:

T2dParticleRenderer

Public Attributes

- TVec2 mPosition
  Particle position.
- TVec2 mUp
  Current up vector of particle. Referenced as pUp.
- TReal mScale
  Current scale of particle. Referenced as pScale.
- TColor mColor
  Current particle color. Referenced as pColor.
- TReal mFrame
  Current frame of the particle animation (as int). Referenced as pFrame.

15.2.2 Member Data Documentation

TVec2 T2dParticle::mPosition

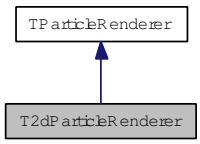
Particle position.

Referenced as pPosition in a Lua particle script.

15.3 T2dParticleRenderer Class Reference

`#include <pf/2dparticlerenderer.h>`

Inheritance diagram for T2dParticleRenderer:



15.3.1 Detailed Description

A particle renderer that expects 2d particles.

This is the default particle renderer used by TLuaParticleSystem.

T2dParticleRenderer defines five particle registers:

- pPosition [2] The 2d position of a particle.
- pUp [2] The 2d "up" vector of a particle.
- pScale [1] The current particle scale.
- pColor [4] The current particle color.
- mFrame [1] The current particle frame (when using a TAnimatedTexture).

See also:

T2dParticle
TLuaParticleSystem

Public Member Functions

- T2dParticleRenderer ()
  Default Constructor.
- virtual ~T2dParticleRenderer ()
  Destructor.
- virtual void Draw (const TVec3 &at, TReal alpha, const ParticleList &particles, int maxParticles)
  Render the particles.
- void SetParticleSize (const TVec2 &size)
  Size of the particle object to render.
- void SetBlendMode (TRenderer::EBlendMode mode)
  Set the blend mode for a particular layer.
- virtual void SetTexture (TTextureRef texture)

Set the texture for the particle.

- virtual void SetRendererOption (str option, const TReal(&value)[4])

  Set a renderer-specific option.

- virtual uint32_t GetPrototypeParticleSize ()

  Size of the array of TReals returned by GetPrototypeParticle.

- virtual TReal * GetPrototypeParticle ()

  Get an initialized particle that will be copied over each particle after creation but before running initializers.

- virtual str GetLuaInitString ()

  Get a TLuaParticleSystem initialization string.

15.3.2 Member Function Documentation

virtual void T2dParticleRenderer::Draw (const TVec3 & *at*, TReal *alpha*, const ParticleList & *particles*, int *maxParticles*) `[virtual]`

Render the particles.

Parameters:

at Location to render particles.
alpha Alpha to render particles with.
particles The list of particles to render.
maxParticles The maximum number of particles this particle system is expecting to render. MUST be greater than the number of particles or Bad Things will happen.

Implements TParticleRenderer.

void T2dParticleRenderer::SetParticleSize (const TVec2 & *size*)

Size of the particle object to render.

Parameters:

size Width and height of particle square on screen in pixels.

void T2dParticleRenderer::SetBlendMode (TRenderer::EBlendMode *mode*)

Set the blend mode for a particular layer.

Parameters:

mode Blend mode

virtual void T2dParticleRenderer::SetTexture (TTextureRef *texture*) `[virtual]`

Set the texture for the particle.

Parameters:

texture Texture to use.

Implements TParticleRenderer.

virtual void T2dParticleRenderer::SetRendererOption (str *option*, const TReal(&) *value*[4]) `[virtual]`

Set a renderer-specific option.

Parameters:

option Option to set.
value Value to set option to, in the form of an array of TReals. Not all values in array are relevant for all options.

Implements TParticleRenderer.

virtual uint32_t T2dParticleRenderer::GetPrototypeParticleSize () `[virtual]`

Size of the array of TReals returned by GetPrototypeParticle.

Returns:

Number of reals.

Implements TParticleRenderer.

virtual TReal* T2dParticleRenderer::GetPrototypeParticle () `[virtual]`

Get an initialized particle that will be copied over each particle after creation but before running initializers.

Returns:

A pointer to an array of TReals.

Implements TParticleRenderer.

virtual str T2dParticleRenderer::GetLuaInitString () `[virtual]`

Get a TLuaParticleSystem initialization string.

Returns:

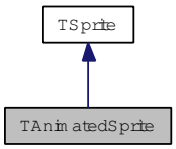
A valid Lua string that will be executed by a TLuaParticleSystem that uses this renderer.

Implements TParticleRenderer.

15.4 TAnimatedSprite Class Reference

`#include <pf/animatedsprite.h>`

Inheritance diagram for TAnimatedSprite:



15.4.1 Detailed Description

A TSprite with an attached TScript.

Similar to TSprite, a TAnimatedSprite should only ever be stored as a reference, but it will work to store one in either a TAnimatedSpriteRef or a TSpriteRef.

Typically you will assign a TAnimatedTexture to a TAnimatedSprite; however, it is legal to assign a normal TTexture to a TAnimatedSprite instead. Obviously the animation script will be unable to change "frames" if a normal TTexture is attached, however.

Initialization/Destruction

- void SetClock (TClock *clock)

  Set the animation to use the passed in clock as its timer.

- virtual ~TAnimatedSprite ()

  Destructor.

- static TAnimatedSpriteRef Create (int32_t layer=0)

  Factory.

Public Member Functions

- virtual bool HitTest (const TPoint &at, const TDrawSpec &parentContext, int32_t opacity=-1, int32_t depth=-1)

  Test to see if a point is within our sprite.

- virtual TRect GetRect (const TDrawSpec &parentContext, int32_t depth=-1)

  Get the rect of this sprite.

Drawing

- virtual void Draw (const TDrawSpec &drawSpec=TDrawSpec(), int32_t depth=-1)

  Draw the sprite and its children.

Animation Control

- void Play (str functionName="DoAnim")
  Plays an animation script.
- void Stop ()
  Stops an animation script that's already playing.
- void Pause (bool pause, int32_t depth=-1)
  Pauses/un-pauses a specific animation.
- void Die ()
  Stop and eradicate the script associated with an animation.
- bool IsPlaying (int32_t depth=-1)
  Test whether or not the animation is currently playing.
- bool IsDone (int32_t depth=-1)
  Return whether or not the animation has signalled it's really done by calling die().

Frame Access

- void SetCurrentFrame (int32_t frame)
  Set the current animation frame.
- int32_t GetCurrentFrame ()
  Get the current animation frame.

Script and Texture Access

- TScript * GetScript ()
  Gets the current script associated with this animation.
- void NewScript ()
  Reset the script to a virgin one that has been initialized with the proper animation functions.
- void **NewThread** ()
- virtual void SetTexture (TTextureRef texture)
  Set the texture of the sprite object.
- TAnimatedTextureRef GetAnimatedTexture ()
  Return an animated texture, if one is attached.
- uint32_t GetNumAnchors ()
  Get the number of anchors in the bound animation.
- uint32_t GetNumFrames ()
  Get the number of frames in the bound animation.

Utility

- TAnimatedSpriteRef GetRef ()
  Get the TAnimatedSpriteRef for this TAnimatedSprite.

Protected Member Functions

- TAnimatedSprite (int32_t layer)

  Default Constructor.

15.4.2 Constructor & Destructor Documentation

TAnimatedSprite::TAnimatedSprite (int32_t *layer*) `[protected]`

Default Constructor.

Parameters:

layer The initial layer for the sprite.

15.4.3 Member Function Documentation

virtual bool TAnimatedSprite::HitTest (const TPoint & *at*, const TDrawSpec & *parentContext*, int32_t *opacity* = −1, int32_t *depth* = −1) `[virtual]`

Test to see if a point is within our sprite.

Parameters:

at Point to test.
parentContext The parent context to test within–where is this sprite being drawn, and with what matrix? Alpha and color information is ignored.
opacity Level of opacity to test for; -1 for a simple bounding box test, or 0-255 for alpha color value, where 0 is transparent (and will therefore always succeed).
depth Depth of children to test. Zero means only test this sprite. -1 means test

Returns:

true if point hits us.

Reimplemented from TSprite.

virtual TRect TAnimatedSprite::GetRect (const TDrawSpec & *parentContext*, int32_t *depth* = −1) `[virtual]`

Get the rect of this sprite.

Parameters:

parentContext The parent context to test within–where is this sprite being drawn, and with what matrix? Alpha and color information is ignored.
depth Depth of children to test

Returns:

Rectangle that includes this sprite.

Reimplemented from TSprite.

static TAnimatedSpriteRef TAnimatedSprite::Create (int32_t *layer* = 0) `[static]`

Factory.

Parameters:

layer Layer of sprite.

Returns:

A reference to a new sprite.

void TAnimatedSprite::SetClock (TClock * *clock*)

Set the animation to use the passed in clock as its timer.

To be effective, must be called before a call to Play() is issued.

Parameters:

clock Clock to use for timing. If NULL, then the global timer is used.

virtual void TAnimatedSprite::Draw (const TDrawSpec & *drawSpec* = TDrawSpec(), int32_t *depth* = −1) `[virtual]`

Draw the sprite and its children.

Parameters:

drawSpec The 'parent' drawspec–the frame of reference that this sprite is to be rendered in. Defaults to a default-constructed TDrawSpec. See TDrawSpec for more details on what is inherited.

depth How many generations of children to draw; -1 means all children.

See also:

TDrawSpec

Reimplemented from TSprite.

void TAnimatedSprite::Play (str *functionName* = "DoAnim")

Plays an animation script.

Parameters:

functionName Name of lua function to run. By default this is "DoAnim"

void TAnimatedSprite::Stop ()

Stops an animation script that's already playing.

Scripts will retain global variable settings even after they're stopped.

void TAnimatedSprite::Pause (bool *pause*, int32_t *depth* = −1)

Pauses/un-pauses a specific animation.

An alternative to calling Pause() on several different TAnimatedTextures is to have them all use the same TClock, and pause the clock instead.

Parameters:

pause true to pause/false to un-pause
depth How many levels to recurse in modifying child animations.

void TAnimatedSprite::Die ()

Stop and eradicate the script associated with an animation.

Any variables saved in the global environment will be erased. The next time you call Play() or GetScript() it will reload any associated TAnimatedTexture() script.

bool TAnimatedSprite::IsPlaying (int32_t *depth* = −1)

Test whether or not the animation is currently playing.

Parameters:

depth How many levels to recurse in testing child sprite animations to see if they're playing. Default is -1 which means to recurse with no limit.

Returns:

True if the animation or one of its children is playing.

bool TAnimatedSprite::IsDone (int32_t *depth* = −1)

Return whether or not the animation has signalled it's really done by calling die().

Parameters:

depth How many levels to recurse in testing child animations.

Returns:

true if the animation is done.

void TAnimatedSprite::SetCurrentFrame (int32_t *frame*)

Set the current animation frame.

Does not affect children.

Parameters:

frame Frame number.

int32_t TAnimatedSprite::GetCurrentFrame ()

Get the current animation frame.

Returns:

Current animation frame number.

TScript* TAnimatedSprite::GetScript ()

Gets the current script associated with this animation.

Creates a script if one doesn't exist already.

DO NOT CACHE this pointer: this script can change over time. As long as you don't call Die() on this animation or reload it from a file, it will copy its environment across scripts, so you can set global variables and be reasonably assured that when you hit "Play" it will retain them–even if it's running in a different TScript (technically a different thread).

Returns:

The current script.

void TAnimatedSprite::NewScript ()

Reset the script to a virgin one that has been initialized with the proper animation functions.

Not necessary to call prior to LoadScript or InitXML, as these functions will call it if no script has been loaded. However, if you want a clean interpreter state, you can call this function.

virtual void TAnimatedSprite::SetTexture (TTextureRef *texture*) `[virtual]`

Set the texture of the sprite object.

Can be a TAnimatedTexture or a regular TTexture; if it's a TTexture, the script and animation control functions will only be relevant to any child sprites that have TAnimatedTextures assigned.

Parameters:

texture Texture to use.

Reimplemented from TSprite.

TAnimatedTextureRef TAnimatedSprite::GetAnimatedTexture ()

Return an animated texture, if one is attached.

If no texture is attached, or if a normal TTexture is attached, return an empty reference.

Returns:

A TAnimatedTextureRef, if one is bound to this sprite.

uint32_t TAnimatedSprite::GetNumAnchors ()

Get the number of anchors in the bound animation.

Returns:

Number of anchors. Returns zero if there is no animated texture attached.

uint32_t TAnimatedSprite::GetNumFrames ()

Get the number of frames in the bound animation.

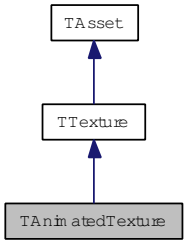
Returns:

Number of frames in the animation. Returns one (1) if there is no animated texture attached.

15.5 TAnimatedTexture Class Reference

`#include <pf/animatedtexture.h>`

Inheritance diagram for TAnimatedTexture:



15.5.1 Detailed Description

This class encapsulates the concept of an animated texture.

This kind of texture is placed directly in video RAM, so be conservative about how large you allow concurrent game animations to become: Consider that a 1024x1024x32 bit texture takes up 4Mb of video RAM, so if you are trying to display more than 10 such textures simultaneously you may start getting performance problems as textures are swapped in and out of video RAM.

A TAnimatedTexture loads in an xml (or anm) file describing the animation, and an image file with multiple source frames.

Additionally, the xml file can describe hot spots, where other textures can be attached.

A Lua script can be embedded in the TAnimatedTexture which can be played back by a TAnimatedSprite. The TAnimatedTexture doesn't have playback capability itself because it represents the raw texture object, and you can have multiple instances of it on the screen at once.

When you use TTexture::Lock() on a TAnimatedTexture, you will lock the full texture surface–which means that TAnimatedTexture::GetWidth() and TAnimatedTexture::GetHeight() will return the wrong values. Instead, you should call TTexture::GetWidth() and TTexture::GetHeight(). For example:

```C++
TAnimatedTextureRef at = ...;

uint32_t w = at->TTexture::GetWidth() ; // This gives you the real texture width
```

Factory Methods/Destruction

- static TAnimatedTextureRef Get (str assetName, str imageOverride="", str maskOverride="")

  Get a TAnimatedTextureRef given an animation name.

- static str GetHandle (str assetName, str imageOverride="", str maskOverride="")

  Get the handle that an animated texture will be registered in the global asset manager as.

- virtual ~TAnimatedTexture ()

  Destructor.

Public Types

- enum { **kNoFrame** = -1 }

Public Member Functions

Drawing Methods

- virtual void CopyPixels (int32_t x, int32_t y, const TRect *sourceRect, TTextureRef _dst)
  Delegates to TTexture::CopyPixels(), which means that it will copy from the multi-frame source image, not from the individual frame.
- virtual void DrawSprite (TReal x, TReal y, TReal alpha, TReal scale, TReal rotRad, uint32_t flags)
  Draw a normal texture to a render target surface as a sprite.
- virtual void DrawSprite (const TDrawSpec &drawSpec)
  Draw an animated texture.

Frame Sequence Information/Update

- uint32_t GetNumFrames ()
  Get number of frames in this animation.
- void SetCurrentFrame (int32_t frame)
  Set the current frame of the animation.
- int32_t GetCurrentFrame ()
  Get the current frame of the animation.
- int32_t GetFrameByName (str name)
  Get a frame by the frame's name.
- TRect GetAnimationBoundingBox ()
  Get the bounding rectangle for this animation.
- TRect GetBoundingBox (int32_t frame)
  Get the bounding rectangle for a specific animation frame.
- TRect GetFrameRect (int32_t frame)
  Get the location of a specific frame inside the animation texture.
- TReal GetFrameAlpha (int32_t frame)
  Get the alpha value associated with a frame.

Anchor Point Support (not needed for most animations).

Anchor points are used to track animated offsets; for instance, the hand of a character that needs to have an object attached might get an animated anchor.

- bool GetAnchorPoint (str name, int32_t *x, int32_t *y)
  Get anchor point information for the current frame of the animation.
- bool GetAnchorPoint (int32_t frame, str name, int32_t *x, int32_t *y)
  Get anchor point information for the selected frame of the animation.

- uint32_t GetNumAnchors ()
  Get number of anchors in this frame of animation.
- bool GetAnchorInfo (int32_t anchorNum, int32_t *x, int32_t *y, str *name)
  Get anchor information.

Image and Frame Data

The data that specifies how each frame needs to be rendered.

- TPoint GetRegistrationPoint ()
  Get the initial image registration point.
- TPoint GetRegistrationPoint (int32_t frame)
  Get the registration point of a frame.
- virtual uint32_t GetWidth ()
  Get the width of the texture.
- virtual uint32_t GetHeight ()
  Get the height of the texture.

Utility Functions

- str GetScript ()
  Get the animation script that was embedded in the XML file.
- str GetPath ()
  Get the path to the source data file.
- TAnimatedTextureRef GetRef ()
  Get a shared pointer (TAnimatedTextureRef) to this texture.

Protected Member Functions

- TAnimatedTexture ()
  Construction is through the factory method.
- virtual void Restore ()
  Restore an asset.

Protected Attributes

- TAnimatedTextureData * mATData
  Internal implementation data.

15.5.2 Member Function Documentation

static TAnimatedTextureRef TAnimatedTexture::Get (str *assetName*, str *imageOverride* = "", str *maskOverride* = "") [static]

Get a TAnimatedTextureRef given an animation name.

The animation name should be an xml or anm file describing the animation.

Parameters:

assetName Name of the xml or anm file which contains the animation data. You can leave off the extension and it will search first for .xml, and then for .anm.
imageOverride Optional image file name to load for this texture, instead of using image named in the xml file
maskOverride Optional mask file name to load for this texture, instead of using mask named in the xml file

Returns:

A TAnimatedTextureRef. This ref will be NULL if it is unable to load.

static str TAnimatedTexture::GetHandle (str *assetName*, str *imageOverride* = "", str *maskOverride* = "") [static]

Get the handle that an animated texture will be registered in the global asset manager as.

Parameters:

assetName Name of asset.
imageOverride Image Override (if any)
maskOverride Mask Override (if any)

Returns:

A string that represents the handle of the object.

virtual void TAnimatedTexture::CopyPixels (int32_t *x*, int32_t *y*, const TRect * *sourceRect*, TTextureRef *_dst*) [virtual]

Delegates to TTexture::CopyPixels(), which means that it will copy from the multi-frame source image, not from the individual frame.

Parameters:

x Left side of resulting rectangle.
y Top edge of resulting rectangle.
sourceRect Source rectangle to blit. NULL to blit the entire surface.
_dst Destination texture. NULL to draw to back buffer.

See also:

TTexture::CopyPixels

Reimplemented from TTexture.

virtual void TAnimatedTexture::DrawSprite (TReal *x*, TReal *y*, TReal *alpha*, TReal *scale*, TReal *rotRad*, uint32_t *flags*) [virtual]

Draw a normal texture to a render target surface as a sprite.

This draws a texture with optional rotation and scaling. Only capable of drawing an entire surface–not a sub-rectangle.

Will draw the sprite within the currently active viewport. X and Y are relative to the upper left corner of the current viewport.

DrawSprite can be called inside TWindow::Draw() or a BeginRenderTarget/EndRenderTarget block.

Parameters:

x x Position where center of sprite is to be drawn.
y y Position where center of sprite is to be drawn.
alpha Alpha to apply to the entire texture. Set to a negative value to entirely disable alpha during blit, including alpha within the source TTexture.
scale Scaling to apply to the texture. 1.0 is no scaling.
rotRad Rotation in radians.
flags Define how textures are drawn. Use ETextureDrawFlags for the flags. Default behavior is eDefaultDraw.

Reimplemented from TTexture.

virtual void TAnimatedTexture::DrawSprite (const TDrawSpec & *drawSpec*) `[virtual]`

Draw an animated texture.

Draws the current frame of the animated texture.

Parameters:

drawSpec The TDrawSpec to use to draw the sprite.

See also:

TTexture::DrawSprite

Reimplemented from TTexture.

uint32_t TAnimatedTexture::GetNumFrames ()

Get number of frames in this animation.

Returns:

Number of frames in animation.

void TAnimatedTexture::SetCurrentFrame (int32_t *frame*)

Set the current frame of the animation.

Parameters:

frame Frame to set animation to, from 0 to n-1, where n is the number of frames.

int32_t TAnimatedTexture::GetCurrentFrame ()

Get the current frame of the animation.

Returns:

frame Current frame of animation, from 0 to n-1, where n is the number of frames.

int32_t TAnimatedTexture::GetFrameByName (str *name*)

Get a frame by the frame's name.

Parameters:

name Name of frame to retrieve.

Returns:

frame Frame number, or -1 if it doesn't exist;

TRect TAnimatedTexture::GetAnimationBoundingBox ()

Get the bounding rectangle for this animation.

Returns:

a TRect that would contain the entire animation.

TRect TAnimatedTexture::GetBoundingBox (int32_t *frame*)

Get the bounding rectangle for a specific animation frame.

Returns:

a TRect that would contain the current frame of the animation

Parameters:

frame Frame to query.

Returns:

The bounding box of this frame.

TRect TAnimatedTexture::GetFrameRect (int32_t *frame*)

Get the location of a specific frame inside the animation texture.

Returns:

a TRect that would contain the frame inside the animation texture

Parameters:

frame Frame to query.

Returns:

The location of this frame inside the animation texture.

TReal TAnimatedTexture::GetFrameAlpha (int32_t *frame*)

Get the alpha value associated with a frame.

Parameters:

frame Frame to retrieve the alpha value from.

Returns:

0.0 (transparent) - 1.0 (opaque).

bool TAnimatedTexture::GetAnchorPoint (str *name*, int32_t * *x*, int32_t * *y*)

Get anchor point information for the current frame of the animation.

Parameters:

name Name of anchor point to retrieve
x [out] Fills in with x coordinate of anchor point if it exists
y [out] Fills in with y coordinate of anchor point if it exists

Returns:

true if the anchor point exists, false otherwise

bool TAnimatedTexture::GetAnchorPoint (int32_t *frame*, str *name*, int32_t * *x*, int32_t * *y*)

Get anchor point information for the selected frame of the animation.

Parameters:

frame Frame index of frame to query.
name Name of anchor point to retrieve
x [out] Fills in with x coordinate of anchor point if it exists
y [out] Fills in with y coordinate of anchor point if it exists

Returns:

true if the anchor point exists, false otherwise

uint32_t TAnimatedTexture::GetNumAnchors ()

Get number of anchors in this frame of animation.

Returns:

Number of anchors in this frame of animation.

bool TAnimatedTexture::GetAnchorInfo (int32_t *anchorNum*, int32_t * *x*, int32_t * *y*, str * *name*)

Get anchor information.

Parameters:

anchorNum Which anchor to fetch
x [out] Fills in with x coordinate of anchor point if it exists
y [out] Fills in with y coordinate of anchor point if it exists
name [out] Fills in name of anchor

Returns:

true if anchorNum is valid, false otherwise

TPoint TAnimatedTexture::GetRegistrationPoint ()

Get the initial image registration point.

The registration point is point from which the image coordinates are calculated.

Returns:

Registration point

TPoint TAnimatedTexture::GetRegistrationPoint (int32_t *frame*)

Get the registration point of a frame.

Parameters:

frame Frame number

Returns:

Registration point

virtual uint32_t TAnimatedTexture::GetWidth () `[virtual]`

Get the width of the texture.

This gets the size of the texture as requested at creation or load; the actual internal size of the texture may vary. If you're using this texture as a source for TRenderer::DrawVertices, see GetInternalSize.

Returns:

Width of the texture in pixels.

Reimplemented from TTexture.

virtual uint32_t TAnimatedTexture::GetHeight () `[virtual]`

Get the height of the texture.

This gets the size of the texture as requested at creation or load; the actual internal size of the texture may vary. If you're using this texture as a source for TRenderer::DrawVertices, see GetInternalSize.

Returns:

Height of the texture in pixels.

Reimplemented from TTexture.

str TAnimatedTexture::GetScript ()

Get the animation script that was embedded in the XML file.

Returns:

Animation script

str TAnimatedTexture::GetPath ()

Get the path to the source data file.

Returns:

A string that represents the path to the source data file.

TAnimatedTextureRef TAnimatedTexture::GetRef ()

Get a shared pointer (TAnimatedTextureRef) to this texture.

Returns:

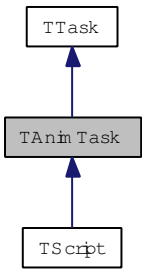
A TAnimatedTextureRef that shares ownership with other Refs to this texture.

Reimplemented from TTexture.

15.6 TAnimTask Class Reference

`#include <pf/animtask.h>`

Inheritance diagram for TAnimTask:



15.6.1 Detailed Description

The TAnimTask interface.

Used as a "callback" for animation or other timed repeating tasks. Can also be used as a simple delayed task: You can call TPlatform::AdoptTask() to adopt a TAnimTask-derived class that will trigger after its delay expires, and then just have its Animate() call return false to let it be destroyed.

See also:

TTask

Public Member Functions

- TAnimTask (TClock *clock=NULL)
  Constructor.
- void SetDelay (uint32_t delay, bool autoRepeat=true, bool resetTime=true, bool forceFrequency=false)
  Set the animation delay.
- void Pause ()
  Pause the current task.
- uint32_t GetTimeUntilReady ()
  Get the number of milliseconds before this task will be ready to trigger again.
- virtual bool Animate ()=0
  Define this function to add the actual animation task.
- void RunOnDraw (bool enable)
  Enable the "Run on Draw" feature, which executes this task prior to every actual screen update.
- void SetClock (TClock *clock)
  Set the reference clock.

- TClock * GetClock ()

  Get the reference clock.

- uint32_t GetTime ()

  Get the current elapsed time of the attached clock.

15.6.2 Constructor & Destructor Documentation

TAnimTask::TAnimTask (TClock * *clock* = `NULL`)

Constructor.

Parameters:

clock Clock that this anim task uses to determine how much time has passed. If this parameter is null then the global clock is used.

15.6.3 Member Function Documentation

void TAnimTask::SetDelay (uint32_t *delay*, bool *autoRepeat* = `true`, bool *resetTime* = `true`, bool *forceFrequency* = `false`)

Set the animation delay.

Parameters:

delay Number of milliseconds to wait to call DoTask().
autoRepeat Repeat this delay after every task.
resetTime Reset the time so that the (initial) delay is counted from *now* rather than from the last task event time.
forceFrequency Force the frequency to be the same as the delay implies; will run the Animate() call multiple times if more than twice the time of delay has elapsed since the last call.

Will not work if resetTime is also true, since resetTime causes the time to be set to the now+delay after the first call, and as such forceFrequency will never cause it to loop.

void TAnimTask::Pause ()

Pause the current task.

Prevents the Animate task from being called.

Resume by calling SetDelay() or RunOnDraw(), as appropriate.

uint32_t TAnimTask::GetTimeUntilReady ()

Get the number of milliseconds before this task will be ready to trigger again.

Returns a negative number if the next call to Ready() would be true immediately.

Returns:

Number of milliseconds before this task is ready. Returns UINT_MAX if task is disabled.

virtual bool TAnimTask::Animate () `[pure virtual]`

Define this function to add the actual animation task.

Returns:

True to continue, false when we're done.

Implemented in TScript.

void TAnimTask::RunOnDraw (bool *enable*)

Enable the "Run on Draw" feature, which executes this task prior to every actual screen update.

If you want the task to run on draw as well as on a timer, set up the timer using SetDelay() normally. If you want to only get called when the screen is about to update, then call Pause() *before* you call RunOnDraw(). Calling Pause after you call RunOnDraw will disable RunOnDraw.

Parameters:

enable True to enable "Run on Draw", false to disable.

void TAnimTask::SetClock (TClock * *clock*)

Set the reference clock.

Useful for having one clock that can be paused and resumed to pause and resume a set or class of TAnimTasks.

Parameters:

clock Clock to use to time animations.

TClock* TAnimTask::GetClock ()

Get the reference clock.

Returns:

A pointer to the currently associated clock.

uint32_t TAnimTask::GetTime ()

Get the current elapsed time of the attached clock.

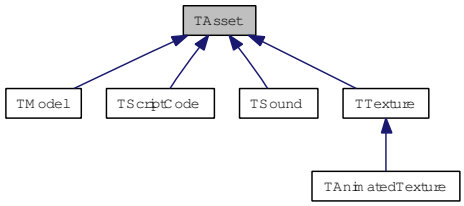
Returns:

Current elapsed time.

15.7 TAsset Class Reference

`#include <pf/asset.h>`

Inheritance diagram for TAsset:



15.7.1 Detailed Description

The interface class for game assets.

Playground-defined TAssets are managed internally by the engine such that if you have a reference to one anywhere in your game, and you request a new one by the same name, it will retrieve the reference rather than reloading the asset.

Public Member Functions

- virtual ~TAsset ()

  Destructor.

- TAssetRef GetRef ()

  Get a reference to this asset. Do not call in a constructor!

Protected Member Functions

- virtual void Release ()

  Release an asset.

- virtual void Restore ()

  Restore an asset.

15.8 TAssetMap Class Reference

`#include <pf/assetmap.h>`

15.8.1 Detailed Description

A collection of assets that simplifies asset reference-holding.

If your game has sections or levels that use a particular set of assets, you can allocate them and store them into a TAssetMap. Then when the level is complete you can release the TAssetMap. This will prevent any glitches from dynamically loading art while the game is running.

Simple usage looks like:

C++
```
// In your class:
TAssetMap mMyMap ;

// In your initialization code:
mMyMap.AddAsset("image1");
mMyMap.AddAsset("image2");

// Later in your game code:
TTextureRef myImage = mMyMap.GetTexture("image1");
```

After you've referenced an image, calling TTexture::Get() on that image will retrieve the reference; however, calling TAssetMap::GetTexture will help you in one of two ways, depending on the state of TAssetMap::SetAutoLoad():

- If SetAutoLoad is false (default), it will ASSERT in debug build that the texture isn't found if you forgot to add it to the map.
- If SetAutoLoad is true, it will load it and *hold* the reference until the map is destroyed. That way the next time you reference it, it will be able to retrieve the already loaded version, even if you've released the in-game reference.

See also:

Game Assets
TAsset

Public Member Functions

- TAssetMap ()
  Default Constructor.
- virtual ~TAssetMap ()
  Destructor.
- void SetAutoLoad (bool autoLoad)
  Activate auto-loading of textures.
- void AddAsset (str assetHandle, TAssetRef asset=TAssetRef(), bool silent=false)
  Add an asset to the map.

- void AddAssets (const char *assets[])

  Add an array of assets to the map.

- TSoundRef GetSound (str asset)

  Get a sound asset from the map.

- TModelRef GetModel (str asset)

  Get a model asset from the map.

- TTextureRef GetTexture (str asset)

  Get a texture asset from the map.

- TAnimatedTextureRef GetAnimatedTexture (str asset)

  Get an animated texture asset from the map.

- TScriptCodeRef GetScriptCode (str asset)

  Get a reference to a pre-loaded, pre-compiled Lua script.

- bool QueryAsset (str asset)

  Query the TAssetMap for an asset; will return true if an asset has been registered successfully with the map.

- void Release ()

  Release ALL managed assets.

15.8.2 Member Function Documentation

void TAssetMap::SetAutoLoad (bool *autoLoad*)

Activate auto-loading of textures.

When this flag is false, requesting a texture that doesn't exist will assert. When it's true, it will simply attempt to load it on demand.

Parameters:

autoLoad True to auto-load.

void TAssetMap::AddAsset (str *assetHandle*, TAssetRef *asset* = TAssetRef(), bool *silent* = `false`)

Add an asset to the map.

Parameters:

assetHandle Name of asset to add.
asset Pointer to asset to associate with assetHandle. Optional.
silent True to supress warnings.

void TAssetMap::AddAssets (const char * *assets*[])

Add an array of assets to the map.

Array is NULL terminated.

For example:

C++
```
const char * assets[]=
{
   "checkers/flipper",
   "checkers/exploder",
   "checkers/heavy",
   "checkers/helium",
   "checkers/rowClear",
   NULL
}

TAssetMap map;
map.AddAssets(assets);
```

Note you can add flags to TSound and TTexture references. See TSound::Get() and TTexture::Get() for details.

Parameters:

assets Array of assets to add.

TSoundRef TAssetMap::GetSound (str *asset*)

Get a sound asset from the map.

Parameters:

asset Asset handle to retrieve. Must be exactly the same string that was specified to add the asset.

Returns:

A TSoundRef. Will be empty (NULL) if the asset is not found, or is of an improper type.

TModelRef TAssetMap::GetModel (str *asset*)

Get a model asset from the map.

Parameters:

asset Asset handle to retrieve. Must be exactly the same string that was specified to add the asset.

Returns:

A TModelRef. Will be empty (NULL) if the asset is not found, or is of an improper type.

TTextureRef TAssetMap::GetTexture (str *asset*)

Get a texture asset from the map.

Parameters:

asset Asset handle to retrieve. Must be exactly the same string that was specified to add the asset.

Returns:

A TTextureRef. Will be empty (NULL) if the asset is not found, or is of an improper type. Note that if you use GetTexture() to retrieve a TAnimatedTexture, it will succeed and return the TAnimatedTextureRef cast to a TTextureRef.

TAnimatedTextureRef TAssetMap::GetAnimatedTexture (str *asset*)

Get an animated texture asset from the map.

Parameters:

asset Asset handle to retrieve. Must be exactly the same string that was specified to add the asset.

Returns:

A TAnimatedTextureRef. Will be empty (NULL) if the asset is not found, or is of an improper type.

TScriptCodeRef TAssetMap::GetScriptCode (str *asset*)

Get a reference to a pre-loaded, pre-compiled Lua script.

Parameters:

asset Asset handle to retrieve. Must be exactly the same string that was specified to add the asset.

Returns:

A TScriptCodeRef. Will be empty (NULL) if the asset is not found, or is of an improper type.

bool TAssetMap::QueryAsset (str *asset*)

Query the TAssetMap for an asset; will return true if an asset has been registered successfully with the map.

Parameters:

asset Asset to query.

Returns:

True if found.

void TAssetMap::Release ()

Release ALL managed assets.

All previously added assets are released and their records are purged. After calling this you can again add assets as usual.

Note that assets that are still referenced elsewhere in your code (or by, e.g., existing TSpriteRefs) will not be destroyed.

A release will happen implicitly on destruction of the TAssetMap.

15.9 TBegin2d Class Reference

`#include <pf/renderer.h>`

15.9.1 Detailed Description

Helper class to wrap 2d rendering.

When you construct this class it activates a TRenderer::Begin2d() state, and when it's destroyed, it calls TRenderer::End2d(). You can also stop the state early by calling TBegin2d::Done().

This allows you to create a local stack variable that will automatically disable TRenderer::Begin2d() mode when it leaves scope.

Public Member Functions

- TBegin2d ()
  Default constructor.
- ~TBegin2d ()
  Destructor.
- void Done ()
  We're done with 2d for now!

15.10 TBegin3d Class Reference

`#include <pf/renderer.h>`

15.10.1 Detailed Description

Helper class to wrap 3d rendering.

When you construct this class it activates a TRenderer::Begin3d() state, and when it's destroyed, it calls TRenderer::End3d(). You can also stop the state early by calling TBegin3d::Done().

This allows you to create a local stack variable that will automatically disable TRenderer::Begin3d() mode when it leaves scope.

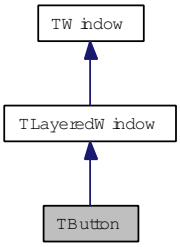
Public Member Functions

- TBegin3d ()
  Default constructor.
- ~TBegin3d ()
  Destructor.
- void Done ()
  We're done with 3d for now!

15.11 TButton Class Reference

`#include <pf/button.h>`

Inheritance diagram for TButton:



15.11.1 Detailed Description

Encapsulation for button functionality.

Handles push-buttons, radio buttons, and toggles. Also handles mouse-over highlighting.

A TButton is implemented as a TLayeredWindow with 3 or 4 layers, depending on the type of button (a kPush button only needs three layers, while the other types need four).

If you define a button in Lua, you can use the special options defined in the documentation of the Lua call Button().

Public Types

- enum EButtonType { kPush = 0, kToggle, kRadio, kNumTypes }

  The type of a TButton.

- enum EButtonLayer { kUp = 0, kDown, kOver, kOverOn }

  The button layers.

- enum EMouseState { kMouseIdle = 0, kMousePush, kMouseOver, kMouseActivated }

  TButton internal dynamic state.

- enum EButtonCreateFlags {
  kGroupStart = 1, kSendMessageToParent = 2, kCloseWindow = 4, kDefaultButton = 8,
  kCancelButton = 16 }

  Button creation flags.

Public Member Functions

- TButton ()

  Constructor.

- ~TButton ()
  Destructor.
- void SetType (EButtonType type)
  Set the button's type.
- void Toggle ()
  Toggle the button's state.
- EButtonType GetType ()
  Get the button's type.
- EMouseState GetState ()
  Get the current mouse state of the button.
- void SetOn (bool on)
  Set a toggle or radio button on.
- bool GetOn ()
  Return whether a particular radio button is on.
- bool IsDefault ()
  Return true if this is a default button.
- bool IsCancel ()
  Return if this is a cancel button.
- void AddFlags (uint32_t flags)
  Add flags to the current button.
- void SetTooltip (str tooltip)
  Set the tooltip for this button.
- void SetLabel (str label)
  Set the label for this button.
- str GetLabel ()
  Get the label for this button.
- virtual bool OnMouseLeave ()
  Notification that the mouse has left the window.
- virtual bool OnMouseUp (const TPoint &point)
  Mouse up handler.
- virtual bool OnMouseDown (const TPoint &point)
  Mouse down handler.
- virtual bool OnMouseMove (const TPoint &point)
  Mouse motion handler.

- virtual bool Press ()

  Press the button from C++: Equivalent to clicking on the button.

- void SetCommand (TButton::Action *action)

  Set the current TButton::Action associated with the button's action.

- bool DoCommand ()

  Do the command associated with this button.

- void SetSound (EMouseState state, str sound)

  Set a sound for a specific button state.

- void SetClickMask (TTextureRef texture)

  Set a click mask for the button.

- virtual void Init (TWindowStyle &style)

  Do Lua initialization.

- virtual bool OnNewParent ()

  Handle any initialization or setup that is required when this window is assigned to a new parent.

- virtual void PostChildrenInit (TWindowStyle &style)

  Do post-children-added initialization when being created from Lua.

Derived Button virtual functions.

Functions that a derived TButton class can override to respond to button state changes with custom behaviors and/or animations.

- virtual void StateUpdate (EMouseState newState, EMouseState previousState)

  Called when the button is activated so that a derived class may trigger an animation or change the button's state.

Classes

- class Action

  An abstract action class for button actions.

- class LuaAction

  A class that wraps a Lua command in an action.

15.11.2 Member Enumeration Documentation

enum TButton::EButtonType

The type of a TButton.

These constants are also available in the Lua GUI thread.

Enumerator:

kPush A normal "push" button.
kToggle A 2-state toggle button.
kRadio One of several "radio" buttons in a group.
Must call BeginGroup() before the first button in the Lua script, or call TButton::AddFlags(kGroupStart) on the first button in the group.
kNumTypes Number of button types (not a real type).

enum TButton::EButtonLayer

The button layers.

Enumerator:

kUp Button is in "up" or "off" state.
kDown Button is in "down" or "on" state.
kOver Button is in roll-over state (and is off for toggles and radio buttons).
kOverOn Button is in roll-over state and is "on".

enum TButton::EMouseState

TButton internal dynamic state.

Enumerator:

kMouseIdle Button mouse state is idle.
kMousePush Button is being pushed by a mouse click.
kMouseOver Button is being rolled over by the mouse.
kMouseActivated A transient state indicating that the button has been activated. Reverts immediately to kMouseIdle.

enum TButton::EButtonCreateFlags

Button creation flags.

Enumerator:

kGroupStart This button is the first in a group.
kSendMessageToParent Send the button's message to the parent modal.
kCloseWindow Automatically send the parent TModalWindow a close message when this button is pushed.
kDefaultButton This button is default button in the local context.
kCancelButton This button is a cancel button in the local context.

15.11.3 Member Function Documentation

void TButton::SetType (EButtonType *type*)

Set the button's type.

Parameters:

type New type for button.

void TButton::Toggle ()

Toggle the button's state.

Button must be of type kToggle, or this method will ASSERT.

EButtonType TButton::GetType ()

Get the button's type.

Returns:

Type of the button.

EMouseState TButton::GetState ()

Get the current mouse state of the button.

Returns:

The state of the button with regards to the mouse.

void TButton::SetOn (bool *on*)

Set a toggle or radio button on.

Parameters:

on True to set to on. False to set to off (toggle only).

bool TButton::GetOn ()

Return whether a particular radio button is on.

Returns:

True for on.

bool TButton::IsDefault ()

Return true if this is a default button.

Returns:

True if default.

bool TButton::IsCancel ()

Return if this is a cancel button.

Returns:

True if this button should abort the window.

void TButton::AddFlags (uint32_t *flags*)

Add flags to the current button.

Parameters:

flags Flags to add. Does not erase flags.

void TButton::SetTooltip (str *tooltip*)

Set the tooltip for this button.

Warning:

Not implemented yet!
Specification subject to change.

Parameters:

tooltip A string for the tooltip.

void TButton::SetLabel (str *label*)

Set the label for this button.

Parameters:

label Text to place in the label for this button

str TButton::GetLabel ()

Get the label for this button.

Returns:

The button's label.

virtual bool TButton::OnMouseUp (const TPoint & *point*) `[virtual]`

Mouse up handler.

Parameters:

point Mouse position.

Reimplemented from TWindow.

virtual bool TButton::OnMouseDown (const TPoint & *point*) `[virtual]`

Mouse down handler.

Parameters:

point Mouse position.

Reimplemented from TWindow.

virtual bool TButton::OnMouseMove (const TPoint & *point*) `[virtual]`

Mouse motion handler.

Parameters:

point Mouse position.

Reimplemented from TWindow.

virtual bool TButton::Press () `[virtual]`

Press the button from C++: Equivalent to clicking on the button.

Returns:

true if the window is still around by the end; false if it's been closed and/or removed from the hierarchy.

void TButton::SetCommand (TButton::Action * *action*)

Set the current TButton::Action associated with the button's action.

Create a TButton::LuaAction to wrap a Lua command. The TButton expects to own the command, and will delete it on TButton destruction.

Parameters:

action Action to bind to button.

bool TButton::DoCommand ()

Do the command associated with this button.

Returns:

True if the window was deleted as a result of the command.

void TButton::SetSound (EMouseState *state*, str *sound*)

Set a sound for a specific button state.

Parameters:

state Which mouse state triggers the sound
sound name of sound file to play

void TButton::SetClickMask (TTextureRef *texture*)

Set a click mask for the button.

A click mask is an image where anything black (all pixels have RGB of less than 20) in the image is considered outside the button, and the mouse will not activate the button if it is in a black area.

Parameters:

texture texture to use as the click mask

virtual void TButton::StateUpdate (EMouseState *newState*, EMouseState *previousState*) `[virtual]`

Called when the button is activated so that a derived class may trigger an animation or change the button's state.

Call the base class if you want to add behavior to the default behavior; otherwise you will need to take responsibility for changing the button's appearance.

Parameters:

newState State we're transitioning to.
previousState State we're transitioning from.

virtual void TButton::Init (TWindowStyle & *style*) `[virtual]`

Do Lua initialization.

Supported tags include:

- close (bool) This button closes its window/dialog if true.
- flags (number) Button label text alignment.
- graphics (table) An array of up to four images for the button: Three for push-buttons (Up, RollOver, Down), Four for toggle and radio buttons (Up, RollOver-Up, Down, RollOver-Down). If the array has fewer than 3 or 4 images, additional images are duplicated from the last given image.
- label (string) Default button text label.
- scale (number) Scale to apply to the button graphics.
- sound (string) Name of sound to play when button pressed.
- rolloversound (string) Name of sound to play when mouse rolls over the button.
- type (number) Button type (kPush, kToggle, kRadio).
- Other generic window tags.

Parameters:

style Style definition for button.

Reimplemented from TWindow.

virtual bool TButton::OnNewParent () `[virtual]`

Handle any initialization or setup that is required when this window is assigned to a new parent.

No initialization of the window has happened prior to this call.

Returns:

True on success; false on failure.

See also:

Init
PostChildrenInit

Reimplemented from TWindow.

virtual void TButton::PostChildrenInit (TWindowStyle & *style*) `[virtual]`

Do post-children-added initialization when being created from Lua.

Any initialization that needs to happen after a window's children have been added can be placed in a derived version of this function.

Warning:

Remember to always call the base class if you're overriding this function.

Parameters:

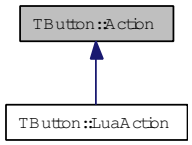
style Current style environment that this window was created in.

Reimplemented from TWindow.

15.12 TButton::Action Class Reference

`#include <pf/button.h>`

Inheritance diagram for TButton::Action:



15.12.1 Detailed Description

An abstract action class for button actions.

Public Member Functions

- virtual void DoAction (TButton *button)=0
  Override this member to perform the action.

15.12.2 Member Function Documentation

virtual void TButton::Action::DoAction (TButton * *button*) **`[pure virtual]`**

Override this member to perform the action.

Parameters:

button A pointer to the button triggering the action.

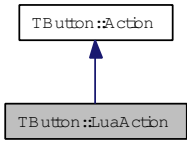
Returns:

Implemented in TButton::LuaAction.

15.13 TButton::LuaAction Class Reference

`#include <pf/button.h>`

Inheritance diagram for TButton::LuaAction:



15.13.1 Detailed Description

A class that wraps a Lua command in an action.

Public Member Functions

- LuaAction (TLuaFunction *action)
  Constructor.
- ~LuaAction ()
  Destructor.
- virtual void DoAction (TButton *button)
  Override this member to perform the action.

Public Attributes

- TLuaFunction * mAction
  The wrapped action.

15.13.2 Constructor & Destructor Documentation

TButton::LuaAction::LuaAction (TLuaFunction * *action*)

Constructor.

Parameters:

action A TLuaFunction to wrap.

15.13.3 Member Function Documentation

virtual void TButton::LuaAction::DoAction (TButton * *button*) `[virtual]`

Override this member to perform the action.

Parameters:

button A pointer to the button triggering the action.

Returns:

Implements TButton::Action.

15.14 TClock Class Reference

`#include <pf/clock.h>`

15.14.1 Detailed Description

The TClock class encapsulates timer functionality.

When you have a need for a time source in your game, you can instantiate a TClock object that has its own Start, Stop, Reset, and Pause controls.

TClocks can be assigned to TAnimTask objects to control when they animate.

For example, if you create a TClock in your game, you can use it to time the animations your game employs. When implementing game-pause functionality, if all animations that should pause reference the game TClock object, then you can pause your game by simply pausing the TClock.

Public Member Functions

- TClock ()

  Constructor.

- virtual ~TClock ()

  Destructor.

- void Start (void)

  Start the timer.

- bool Pause (void)

  Pause the timer.

- void Reset (void)

  Pauses and zeros a timer.

- uint32_t GetTime (void)

  Get the current running millisecond count.

- void SetTime (uint32_t t)

  Set the current clock time.

- bool GetPaused ()

  Return whether the clock is paused.

15.14.2 Member Function Documentation

void TClock::Start (void)

Start the timer.

Has no effect on a running timer.

bool TClock::Pause (void)

Pause the timer.

Will have no effect on a paused timer.

Returns:

true if timer was paused already.

uint32_t TClock::GetTime (void)

Get the current running millisecond count.

Returns:

Number of milliseconds clock has been allowed to run.

void TClock::SetTime (uint32_t *t*)

Set the current clock time.

Resets the clock and sets the current elapsed time.

Parameters:

t Value to set the current time to.

bool TClock::GetPaused ()

Return whether the clock is paused.

Returns:

True if paused.

15.15 TColor Class Reference

`#include <pf/pftypes.h>`

15.15.1 Detailed Description

An RGBA color value.

TColor stores the color values as TReals.

See also:

TColor32

Public Member Functions

- TColor ()

  Constructor.

- TColor (TReal R, TReal G, TReal B, TReal A)

  Construct from floating point values.

- void Init (uint32_t R, uint32_t G, uint32_t B, uint32_t A)

  Integer initializer; expects color values from 0-255.

- bool operator== (const TColor &color) const

  Compare two colors.

- bool operator!= (const TColor &color) const

  Compare two colors.

Public Attributes

- TReal r

  Red value, 0-1.

- TReal g

  Green value, 0-1.

- TReal b

  Blue value, 0-1.

- TReal a

  Alpha value, 0-1.

15.15.2 Constructor & Destructor Documentation

TColor::TColor ()

Constructor.

Zeros colors by default.

TColor::TColor (TReal *R*, TReal *G*, TReal *B*, TReal *A*)

Construct from floating point values.

Values are expected to range from zero to one.

Parameters:

R Red.
G Green.
B Blue.
A Alpha, where 0 is transparent and 1 is opaque.

15.15.3 Member Function Documentation

void TColor::Init (uint32_t *R*, uint32_t *G*, uint32_t *B*, uint32_t *A*)

Integer initializer; expects color values from 0-255.

Parameters:

R Red (0-255)
G Green (0-255)
B Blue (0-255)
A Alpha (0-255)

bool TColor::operator== (const TColor & *color*) const

Compare two colors.

Parameters:

color Right-hand color to compare with.

Returns:

True on equality.

References a, b, g, and r.

bool TColor::operator!= (const TColor & *color*) const

Compare two colors.

Parameters:

color Right-hand color to compare with.

Returns:

True on non-equality.

15.16 TColor32 Struct Reference

`#include <pf/pftypes.h>`

15.16.1 Detailed Description

A 32-bit platform native color value.

Used in some structures to save space and/or map to platform-specific structure layouts. The internal color value should not be accessed directly if you want your code to work cross-platform.

Public Member Functions

- TColor32 ()
  Default constructor.
- TColor32 (const TColor &c)
  Build a TColor32 from a TColor.
- TColor32 (uint8_t r, uint8_t g, uint8_t b, uint8_t a)
  Build a TColor32 from RGBA values.
- TColor32 (uint32_t c)
  Build a TColor32 from an unsigned long ARGB (A is highest byte).
- uint8_t Alpha () const
  Read color alpha (0-255).
- uint8_t Red () const
  Read color red (0-255).
- uint8_t Green () const
  Read color green (0-255).
- uint8_t Blue () const
  Read color blue (0-255).
- void SetAlpha (uint8_t a)
  Write color alpha (0-255).
- void SetRed (uint8_t r)
  Write color red (0-255).
- void SetGreen (uint8_t g)
  Write color green (0-255).
- void SetBlue (uint8_t b)
  Write color blue (0-255).

Public Attributes

- uint32_t color

  Raw, platform-specific color value.

15.16.2 Constructor & Destructor Documentation

TColor32::TColor32 (const TColor & *c*)

Build a TColor32 from a TColor.

Parameters:

c Source color.

TColor32::TColor32 (uint8_t *r*, uint8_t *g*, uint8_t *b*, uint8_t *a*)

Build a TColor32 from RGBA values.

Parameters:

r R (0-255)
g G (0-255)
b B (0-255)
a A (0-255)

TColor32::TColor32 (uint32_t *c*)

Build a TColor32 from an unsigned long ARGB (A is highest byte).

Parameters:

c An "ARGB"-order packed color.

15.16.3 Member Data Documentation

uint32_t TColor32::color

Raw, platform-specific color value.

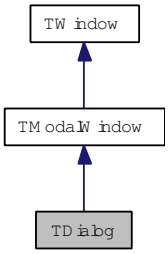
Warning:

Do not modify directly! The meaning of this value may change from video card to video card, or from OS X to Windows, or from Intel to PowerPC Macs.

15.17 TDialog Class Reference

`#include <pf/dialog.h>`

Inheritance diagram for TDialog:



15.17.1 Detailed Description

A generic modal dialog.

Public Types

- enum { **kResponseOK** }

Public Member Functions

- TDialog (str initFileName, str bodyText, str titleText)
  Default Constructor.
- virtual ~TDialog ()
  Destructor.
- virtual bool OnMessage (TMessage *message)
  Handle a message.
- virtual bool OnNewParent ()
  Do the real initialization on being added to the hierarchy.

Protected Types

- enum { **kHttpLink** = 30000 }

15.17.2 Constructor & Destructor Documentation

TDialog::TDialog (str *initFileName*, str *bodyText*, str *titleText*)

Default Constructor.

Parameters:

initFileName Dialog spec file.
bodyText Text for the body of the dialog
titleText Text for the title of the dialog.

15.17.3 Member Function Documentation

virtual bool TDialog::OnMessage (TMessage * *message*) `[virtual]`

Handle a message.

Parameters:

message Payload of message.

Returns:

True if message handled; false otherwise.

Reimplemented from TModalWindow.

virtual bool TDialog::OnNewParent () `[virtual]`

Do the real initialization on being added to the hierarchy.

Returns:

true on success.

Reimplemented from TModalWindow.

15.18 TDrawSpec Class Reference

`#include <pf/drawspec.h>`

15.18.1 Detailed Description

2d drawing parameters for use in DrawSprite.

Public Member Functions

- TDrawSpec (const TVec2 &at=TVec2(), TReal alpha=1, TReal scale=1, uint32_t flags=0)
  Default constructor with optional arguments for convenience.
- TDrawSpec GetRelative (const TDrawSpec &parent) const
  Create a new TDrawSpec that is a combination of this TDrawSpec and a parent reference frame.

Public Attributes

- TMat3 mMatrix
  3x3 matrix including the offset and relative orientation (and scaling) of the sprite.
- TVec2 mCenter
  Logical center of the texture in pixels; can be outside of the texture.
- TColor mTint
  Vertex coloring for the texture.
- TReal mAlpha
  Alpha to use to draw sprite.
- TRect mSourceRect
  Source rectangle to extract image from.
- uint32_t mFlags
  Options for drawing.
- TRenderer::EBlendMode mBlendMode
  Blend mode to use, if any; set to TRenderer::kBlendINVALID to use current mode (default).

Static Public Attributes

- static const int32_t kFlipHorizontal = 1<<0
  Flip texture horizontally.

- static const int32_t kFlipVertical = 1<<1

  Flip texture vertically.

- static const int32_t kUseSourceRect = 1<<2

  Use the source rectangle field.

- static const int32_t kUseCenter = 1<<3

  Use the center field.

15.18.2 Constructor & Destructor Documentation

TDrawSpec::TDrawSpec (const TVec2 & *at* = TVec2 (), TReal *alpha* = 1, TReal *scale* = 1, uint32_t *flags* = 0)

Default constructor with optional arguments for convenience.

Parameters:

at Position to draw the object.
alpha Alpha level of the object, 0-1.
scale The scale of the object, where 1.0 is its actual size.
flags Options flags. See mFlags.

References mMatrix, and TMat3::Scale().

15.18.3 Member Function Documentation

TDrawSpec TDrawSpec::GetRelative (const TDrawSpec & *parent*) const

Create a new TDrawSpec that is a combination of this TDrawSpec and a parent reference frame.

The new TDrawSpec will have a combined matrix, combined alpha, and inherited relative flip flags (kFlipVertical + kFlipVertical = no flip, same for kFlipHorizontal). Blend modes are inherited only if this TDrawSpec is set to kBlendINVALID.

Other parameters stay the same as this class and are NOT inherited.

Parameters:

parent Parent to combine

Returns:

New combined TDrawSpec

References TRenderer::kBlendINVALID, kFlipHorizontal, kFlipVertical, mAlpha, mBlendMode, mFlags, and mMatrix.

15.18.4 Member Data Documentation

const int32_t TDrawSpec::kUseSourceRect = 1<<2 [static]

Use the source rectangle field.

const int32_t TDrawSpec::kUseCenter = 1<<3 [static]

Use the center field.

Warning:

If you set kUseCenter on the TDrawSpec of a TAnimatedSprite, you will disable the automatic positioning of the TAnimatedSprite, and the animation will no longer work as expected.

TMat3 TDrawSpec::mMatrix

3x3 matrix including the offset and relative orientation (and scaling) of the sprite.

Use mMatrix[2] to set the position where the center of sprite is to be drawn.

Use mMatrix.Scale() to set the scale of the sprite.

Referenced by GetRelative(), and TDrawSpec().

TVec2 TDrawSpec::mCenter

Logical center of the texture in pixels; can be outside of the texture.

Drawing of the texture will be done relative to this point: If this point is (0,0), it will draw relative to the upper left corner of the image, for instance.

This is **not** the parameter to use to set the position of an object; for that, see TDrawSpec::mMatrix. If you misuse the mCenter field to position a sprite, you will likely run into problems with inherited positions and TAnimatedTexture drawing. The mCenter field isn't inherited (as it's intended to specify the reference point of **this** sprite), and TAnimatedTexture uses mCenter internally to properly position the animation after sidewalk has cropped all of the frames.

This parameter is **not** inherited when used with TSprite::Draw(), so if you want to modify the center of a sprite you need to modify the sprite's TSprite::GetDrawSpec() and not the environment passed in to TSprite::Draw().

mFlags must have kUseCenter set to cause this field to be valid. Otherwise the actual center of the texture is used.

See also:

kUseCenter

TColor TDrawSpec::mTint

Vertex coloring for the texture.

The alpha channel of this color is ignored. Default value on construction is pure white.

Applies only to this object–for sprites, does NOT get inherited by child sprites.

TReal TDrawSpec::mAlpha

Alpha to use to draw sprite.

Zero is completely transparent and one is completely opaque (subject to blend mode and texture alpha values).

This value is multiplicatively inherited in child sprites using GetRelative().

Referenced by GetRelative().

TRect TDrawSpec::mSourceRect

Source rectangle to extract image from.

This parameter is NOT inherited when used with TSprite::Draw(), so if you want to modify the source rectangle of a sprite you need to modify the sprite's TSprite::GetDrawSpec() and not the environment passed in to TSprite::Draw().

mFlags must have kUseSourceRect set to activate.

uint32_t TDrawSpec::mFlags

Options for drawing.

Inherits flip flags with an XOR, so if a parent and child are, e.g., both horizontally flipped, the child will be drawn normally.

Options include kFlipHorizontal, kFlipVertical, kUseSourceRect, and kUseCenter.

Referenced by GetRelative().

TRenderer::EBlendMode TDrawSpec::mBlendMode

Blend mode to use, if any; set to TRenderer::kBlendINVALID to use current mode (default).

Inherited when parent mode set and child is kBlendINVALID.

Referenced by GetRelative().

15.19 TEncrypt Class Reference

`#include <pf/encrypt.h>`

15.19.1 Detailed Description

A class that encapsulates an encryption engine.

Public Member Functions

- TEncrypt (str encryptionKey)
  Constructor.
- ~TEncrypt ()
  Destructor.
- str EncryptStr (str input)
- str DecryptStr (str input)
- uint32_t EncryptBinary (const void *input, uint32_t inLength, void *output, uint32_t outLength, bool base64)
- uint32_t DecryptBinary (const void *input, uint32_t inLength, void *output, uint32_t outLength, bool base64)
- uint32_t GetLastSize ()
  Get the last decrypted or encrypted data size.

15.19.2 Constructor & Destructor Documentation

TEncrypt::TEncrypt (str *encryptionKey*)

Constructor.

Parameters:

encryptionKey Key to use for encrypting/decrypting. The key should consist of valid Base64 characters(0-9,a-z,A-Z,+,/) and be correctly formatted (i.e. user proper '=' suffix).

15.19.3 Member Function Documentation

str TEncrypt::EncryptStr (str *input*)

Parameters:

input Str to encrypt.

Returns:

returns the input encrypted as a Str.

str TEncrypt::DecryptStr (str *input*)

Parameters:

input Str to decrypt. This str must have been created with EncryptStr to ensure proper decryption.

Returns:

returns the decrypted str.

uint32_t TEncrypt::EncryptBinary (const void * *input*, uint32_t *inLength*, void * *output*, uint32_t *outLength*, bool *base64*)

Parameters:

input Data to encrypt
inLength Length of input in bytes
output Buffer to place encrypted data into.
outLength Size of output buffer.
base64 true to fill the buffer with base64 characters, so it can be passed around as a string. false will fill the buffer with binary data.

Returns:

returns 0 if successful. Otherwise, if output is NULL or outLength is too small for the resulting data, then returns the size in bytes that the output buffer must be to hold the encrypted data.

uint32_t TEncrypt::DecryptBinary (const void * *input*, uint32_t *inLength*, void * *output*, uint32_t *outLength*, bool *base64*)

Parameters:

input Data to decrypt. This data must have been encrypted with EncryptBinary to ensure proper decryption.
inLength Length of input in bytes
output Buffer to place decrypted data into.
outLength Size of output buffer.
base64 true if the incoming buffer is in base64, false if the incoming buffer is in binary

Returns:

returns 0 if successful. Otherwise, if output is NULL or outLength is too small for the resulting data, then returns the size in bytes that the output buffer must be to hold the decrypted data.

uint32_t TEncrypt::GetLastSize ()

Get the last decrypted or encrypted data size.

Returns:

Size of last encrypted or decrypted data.

15.20 TEvent Class Reference

`#include <pf/event.h>`

15.20.1 Detailed Description

System event encapsulation.

Public Types

- enum EEventCode {
  kIdle = 1, kNull, kClose, kQuit,
  kMouseDown, kExtendedMouseEvent, kMouseUp, kMouseMove,
  kMouseLeave, kMouseHover, kKeyDown, kKeyUp,
  kChar, kUTF32Char, kRedraw, kTimer,
  kDisplayModeChange, kActivate, kFullScreenToggle }
  Event codes.
- enum EKeyFlags { kShift = 1, kControl = 2, kAlt = 4, kExtended = 8 }
  Event key flags.

Public Attributes

- int32_t mType
  Type of event.
- int32_t mKey
  Key for keyboard events.
- TPoint mPoint
  Point for mouse events.
- EKeyFlags mKeyFlags
  Flags for key events.

Static Public Attributes

Key Codes

Use these constants to refer to the given keys in your code when processing an OnKeyDown() message.

See also:

TWindow::OnKeyDown

- static int32_t kUp
  Up arrow key.
- static int32_t kDown
  Down arrow key.
- static int32_t kLeft
  Left arrow key.
- static int32_t kRight
  Right arrow key.
- static int32_t kEnter
  Enter key.
- static int32_t kEscape
  The "Esc" key.
- static int32_t kTab
  The Tab key.
- static int32_t kPaste
  The "Paste" keyboard combination.
- static int32_t kPageUp
  The Page-Up key.
- static int32_t kPageDown
  The Page-Down key.
- static int32_t kBackspace
  The backspace key.
- static int32_t kDelete
  The Delete key.

15.20.2 Member Enumeration Documentation

enum TEvent::EEventCode

Event codes.

Enumerator:

kIdle Idle event processing time.
kNull EMPTY event.
kClose A close request (Alt-F4, clicking close button).
kQuit A QUIT NOW event.
kMouseDown Mouse down.
kExtendedMouseEvent Right Mouse Button/Wheel down.
kMouseUp Mouse up.
kMouseMove Mouse move.

kMouseLeave Mouse has left the window. You must TWindowManager::AddMouseListener() the mouse to receive this message; note that TButton-derived classes automatically capture the mouse on mouse-over.

kMouseHover Mouse has hovered over a point on the window.
kKeyDown Key down.
kKeyUp Key up.
kChar translated character event
kUTF32Char translated utf32 event
kRedraw Redraw the screen now.
kTimer A timer has triggered.
kDisplayModeChange The display mode has changed.
kActivate Our application has activated/deactivated. mKey is set to 0 on deactivate, or 1 on activate.
kFullScreenToggle The user has toggled full screen mode. mKey is set to 0 on windowed mode, or non-zero on full screen mode.

enum TEvent::EKeyFlags

Event key flags.

Enumerator:

kShift Shift is pressed.
kControl Control (or command) is pressed.
kAlt Alt key is pressed.
kExtended Reserved.

15.20.3 Member Data Documentation

int32_t TEvent::kEnter `[static]`

Enter key.

On keyboards with more than one key of this description, the same constant is returned for both.

int32_t TEvent::kPaste `[static]`

The "Paste" keyboard combination.

Typically Control-V on Windows, Command-V on OS X.

int32_t TEvent::mType

Type of event.

See also:

EEventCode

15.21 TFile Struct Reference

`#include <pf/file.h>`

15.21.1 Detailed Description

A file reading abstraction.

All file access from Playground games should be handled through this abstraction.

File names must consist ONLY of the following characters:

- a-z : Lowercase letters
- 0-9 : Digits
- - : Hyphen
- _ : Underscore
- . : Dot

Paths must be separated by forward slash ("/"). All files must be specified without a leading slash.

A file in "assets/bitmaps/" called "my.png" would be loaded with the handle "bitmaps/my.png".

To access a writable folder, prefix the file name with either user:, common:, or desktop: to get to either the user's personal data folder, the system's common data folder, or the desktop folder. You may NOT write to the assets folder during the game –it will ASSERT in debug build if you try.

There is a set of C stdio-compatible routines that you can use to port existing fopen-style interfaces:

- pf_open()
- pf_close()
- pf_seek()
- pf_tell()
- pf_getc()
- pf_gets()
- pf_read()
- pf_write() (only to user:, common: and desktop: folders)
- pf_eof()
- pf_error()
- pf_ungetc()

If you absolutely must use fscanf(), you can use pf_fgets to get a line of text, and then use sscanf() to parse the line. However, in general it is be much better to read and write the data as XML using TXmlNode.

Public Member Functions

- TFile ()
  Default constructor.
- ~TFile ()
  Destructor.
- bool Open (str path, eFileMode mode=kReadBinary)
  Open an existing file.
- bool IsValid () const
  Return true if the file was opened successfully.
- bool AtEOF ()
  Return true if the file is at the EOF.
- void Close ()
  Close a file.
- void Seek (long offset, eFileSeek seek)
  Seek to a specific file position.
- long Tell ()
  Get the current file position.
- long Size ()
  Get the file's size.
- long Read (void *buffer, unsigned long bytes)
  Read bytes into buffer.
- long Write (const void *buffer, unsigned long bytes)
  Write bytes to file from buffer.
- void Unget ()
  "Unget" one character.

Static Public Member Functions

- static bool Exists (str handle)
  Test to see if a file exists.
- static str Source (str handle)
  Return the source of the file.
- static str GetCWD ()
  Get the current working directory.

- static bool GetNextFile (str folder, str *file, bool subfolders=false)

  Iterate through the files in a folder.

- static void ScanForResources ()

  Internal function: Scan for updated file resources.

- static void ShutDownFileSystem ()

  Clear out the file system cache.

- static void ScanFolderForResources (str folder, str prefix="")

  Scan a folder (and subfolder) for files.

- static void AddMemoryFile (str memoryFileName, void *base, uint32_t size)

  Add a virtual file to the file system.

- static void AddFileMask (const char *mask, bool add=true)

  Add a set of files-to-be-ignored to TFile.

- static bool DeleteFile (str filename)

  Delete a file from user: or common:, or dereference a memory file.

15.21.2 Member Function Documentation

bool TFile::Open (str *path*, eFileMode *mode* = `kReadBinary`)

Open an existing file.

Closes any file this TFile previously had open, and attempts to open the given file.

Parameters:

path Path to file
mode Mode to open file

Returns:

true on success.

bool TFile::IsValid () const

Return true if the file was opened successfully.

Returns:

True on success; false on failure.

bool TFile::AtEOF ()

Return true if the file is at the EOF.

Returns:

True on EOF.

void TFile::Close ()

Close a file.

Optional; file is automatically closed on destruction of the TFile

void TFile::Seek (long *offset*, eFileSeek *seek*)

Seek to a specific file position.

Parameters:

offset Offset from reference position.
seek Which reference position to use.

long TFile::Tell ()

Get the current file position.

Returns:

Current offset into the file.

long TFile::Size ()

Get the file's size.

Returns:

File size in bytes.

long TFile::Read (void * *buffer*, unsigned long *bytes*)

Read bytes into buffer.

Parameters:

buffer Buffer to fill.
bytes Bytes to read.

Returns:

bytes read

long TFile::Write (const void * *buffer*, unsigned long *bytes*)

Write bytes to file from buffer.

Parameters:

buffer Buffer to read bytes from.
bytes Bytes to write.

Returns:

bytes written

void TFile::Unget ()

"Unget" one character.

Semantics are similar to ungetc, in that you can only ever "unget" one character.

Unlike ungetc, it will only unget exactly the previous character you just read–there is no option to select a character to unget.

Ungetting past the beginning of the file is not allowed.

static bool TFile::Exists (str *handle*) `[static]`

Test to see if a file exists.

Expects a standard resource handle.

Parameters:

handle Resource handle.

Returns:

True if file exists

static str TFile::Source (str *handle*) `[static]`

Return the source of the file.

Parameters:

handle Handle of the file to query.

Returns:

"filesystem" if it's a normal file, "memoryfile" if it's a memory file, an empty string if file isn't found, or the name of the source .pfp file.

static str TFile::GetCWD () `[static]`

Get the current working directory.

Returns:

Current working directory.

static bool TFile::GetNextFile (str *folder*, str * *file*, bool *subfolders* = `false`) `[static]`

Iterate through the files in a folder.

Parameters:

folder Folder to search. File globs are NOT supported currently.
file Pointer to a string to receive each file name.

Start iteration with an empty string, and pass the value you received previously to get the next value in the iteration.

Parameters:

subfolders True to return files in subfolders as well.

Returns:

True if successful; false to signal end of iteration.

static void TFile::ScanForResources () `[static]`

Internal function: Scan for updated file resources.

Called by Playground on application startup. If the file system changes this needs to be called again to update the internal cache.

static void TFile::ScanFolderForResources (str *folder*, str *prefix* = `""`) `[static]`

Scan a folder (and subfolder) for files.

Parameters:

folder Folder to scan.
prefix File system prefix to expect.

static void TFile::AddMemoryFile (str *memoryFileName*, void * *base*, uint32_t *size*) `[static]`

Add a virtual file to the file system.

After you call this function, opening a file with the given name for read will supply the given data. The data is not copied, so you are responsible for ensuring that the data buffer continues to exist for as long as the memory file is available. In other words, the client owns the memory block, and must ensure its lifetime exceeds that of the memory file entry.

Opening the file for write is not currently supported.

Parameters:

memoryFileName Name of the virtual file to add.
base Pointer to the start of the virtual file data. Pass NULL to delete the virtual file.
size Size of the virtual file.

static void TFile::AddFileMask (const char * *mask*, bool *add* = `true`) `[static]`

Add a set of files-to-be-ignored to TFile.

These files will not be "visible" to the TFile file system.

Paths should be given the same as a path would be given to TFile::Open().

Entries in this string are space (or line) separated.

Each entry matches either one complete file path, or one complete folder path. Folders are NOT matched in user: or common:–only complete file paths are matched.

Passing in an entry of -* will clear the entire file mask. If you put this at the start of your string, it will clear the file mask before adding the rest of the files in your string.

Parameters:

mask A space-delimited string of complete file or folder paths.
add True to add the files to the mask (ignore the files in the list); false to remove the files from the mask (to make them show up again).

static bool TFile::DeleteFile (str *filename*) `[static]`

Delete a file from user: or common:, or dereference a memory file.

For a memory file, just removes the entry in the file table: No memory is deallocated, as the client owns the memory.

Note that if you try to delete a file that you currently have open, the behavior is undefined.

Parameters:

filename Filename to delete. Must be prefixed by user: or common:.

Returns:

True on success.

15.22 TFlashHost Class Reference

`#include <pf/flashhost.h>`

15.22.1 Detailed Description

An embedded Flash-playback routine.

Can be used in-game for cut-scenes. Flash animation will play until it's "done", at which point control will return to the game.

From within Lua, you can call DisplaySplash() to display a Flash animation and an optional replacement bitmap.

To ensure that the Flash file plays correctly on the Mac, you should:

1. Have an 800x600 stage in your Flash content (assuming your display resolution is 800x600).
2. Have an empty first frame, and then place an 800x10 rectangle shape cast member at (0,-30). (Again, assuming that 800 is the display width.)

This will guarantee that the Flash stage size is calculated correctly by Playground. The Mac version of Playground uses WebKit to load an HTML file that loads an SWF file that loads the Flash content (it's complicated, but necessary). Flash does not have a method to determine the stage size of the loaded clip, however. You can get the dimensions of the movie clip, but these are computed as the bounding box of the objects onstage, and this is what we are using when we decide how much we need to scale the loaded content to fit 800x600.

This is why the rectangle shape is necessary: So that Playground can retrieve the 800 width and determine that it doesn't need to scale the resulting Flash file.

Public Member Functions

- TFlashHost (const char *filename)
  Constructor.
- ~TFlashHost ()
  Destructor.
- bool Start (bool bLoop=false, bool bTranslate=false, bool bAllowInput=false)
  Start Flash file.
- void Stop ()
  Stop playing file.
- long GetTotalFrames ()
  Get the total number of frames.
- long GetFrameNum ()
  Get the current frame number.
- bool IsPlaying ()
  Test to see whether the animation is playing.

15.22.2 Constructor & Destructor Documentation

TFlashHost::TFlashHost (const char * *filename*)

Constructor.

Parameters:

filename Flash file to play.

15.22.3 Member Function Documentation

bool TFlashHost::Start (bool *bLoop* = `false`, bool *bTranslate* = `false`, bool *bAllowInput* = `false`)

Start Flash file.

Flash movies need to be in Flash 4 format to provide the most compatibility across all systems. Translatable movies need to be Flash 6.

It is possible to translate a Flash movie by setting the bTranslate parameter to true. When translate mode is turned on, any string in the string table that begins with "FLASH_" will be sent to the Flash movie for translation. If all of the text in the Flash movie is set up as dynamic text, and each text box references a variable that is named after an entry in your strings.xml file (minus the "FLASH_"), then translation will succeed. You will also want to embed the font you want inside of your movie, so that if the user doesn't have the font installed on their system the font will still show up correctly.

For example, if you have a dynamic text variable in your movie named "text_box_1", and have an entry in your strings.xml table named "FLASH_text_box_1", then at run time, "text_box_1" in your flash movie will be set to whatever the value of "FLASH_text_box_1" is in your strings.xml file.

It is recommended that inside the Flash movie itself, incomplete text is used when doing layouts. That way it will be obvious if it has been forgotten to add any text to the translation pipeline. If the text inside the movie is correct, it is not clear whether or not the movie is using the original text or the translated text until someone actually goes in to translate the strings.xml file.

Parameters:

bLoop Loop file.
bTranslate Send translated strings to the movie.
bAllowInput If this is true, the system cursor will be shown and mouse clicks will be sent to the flash movie. If it is false, the cursor will be hidden and clicks will be sent to the game only.

Returns:

True on success.

long TFlashHost::GetTotalFrames ()

Get the total number of frames.

Returns:

Number of frames.

long TFlashHost::GetFrameNum ()

Get the current frame number.

Returns:

Frame number.

bool TFlashHost::IsPlaying ()

Test to see whether the animation is playing.

Returns:

True if playing.

15.23 TGameState Class Reference

`#include <pf/gamestate.h>`

15.23.1 Detailed Description

An object that allows the game to communicate state information to the system.

This allows the system to, for example, interact with MSN or Zylom web hooks. In Playground-published games this system is also used to collect game metrics.

See also:

Tracking Game Metrics

Public Member Functions

- void SetState (str key, int32_t value=0)

  Set a numeric key value.

- void SetState (str key, str value)

  Set a string key value.

- str GetState (str key)

  Get a string key value.

- bool QueryRestartMode ()

  Query this function frequently; if it ever returns true, immediately restart the current game mode at level 1.

- bool QueryJumpToMenu ()

  Query this function frequently; if it ever returns true, immediately exit the current game mode and bring up the main menu.

- bool QueryMuteGame ()

  Query this function to determine the mute state displayed in the game.

- bool QueryPauseGame ()

  TPlatform::GetEvent() will handle QueryPauseGame(); no user code is required to handle it correctly.

Static Public Member Functions

- static TGameState * GetInstance ()

  Get an instance to the global game state object.

- static void RegisterHandler (TGameStateHandler *handler)

  Register a game state handler.

Static Public Attributes

- static const char * kCmdLine
  Key to pass command line into attached game monitor.
- static const char * kLevel
  State key for setting current game level.
- static const char * kSubLevel
  State key for setting current game sub-level. (optional).
- static const char * kPause
  State key to let the system know your game has been paused or resumed.
- static const char * kMute
  State key to indicate the game is in MUTE state.
- static const char * kBreakNow
  Key to signify that the game is at a pause in the action and can be stopped.
- static const char * kScore
  State key for score.
- static const char * kGameMode
  State key for game mode.
- static const char * kGameOver
  State key for game over.
- static const char * kNoMoreContent
  State key for no more free content in a Web version.
- static const char * kDownloadButton
  Download button pressed: In a Web game, there will be a "download (and buy) the real game" button.
- static const char * kUpsellFrequency
  The frequency you should use to display upsell screens.
- static const char * kRestartButton
  Query this value to determine whether a restart button should be displayed instead of an "OK" button on an upsell screen.
- static const char * kSystemMode
  Key used to query the current system mode(s).
- static const char * kMaxLevel
  Key used to query what level the game should cap out at; used by some web providers to vary how many levels can be played.

15.23.2 Member Function Documentation

void TGameState::SetState (str *key*, int32_t *value* = 0)

Set a numeric key value.

To time states in a FirstPeek build, set a state to "1" when active and "0" when inactive. States always set in this manner will have their time written to the FirstPeek file.

Parameters:

key Key to set.
value Value to set it to.

void TGameState::SetState (str *key*, str *value*)

Set a string key value.

Parameters:

key Key to set.
value Value to set it to.

str TGameState::GetState (str *key*)

Get a string key value.

Parameters:

key Key to read.

Returns:

Current state of key.

bool TGameState::QueryRestartMode ()

Query this function frequently; if it ever returns true, immediately restart the current game mode at level 1.

Returns:

true to restart game mode.

bool TGameState::QueryJumpToMenu ()

Query this function frequently; if it ever returns true, immediately exit the current game mode and bring up the main menu.

Returns:

true to go to menu.

bool TGameState::QueryMuteGame ()

Query this function to determine the mute state displayed in the game.

As your user modifies the mute state within the game, you should call SetState() with kMute to notify the system of the change.

Returns:

true to mute game.

bool TGameState::QueryPauseGame ()

TPlatform::GetEvent() will handle QueryPauseGame(); no user code is required to handle it correctly.

Returns:

True to pause game; false otherwise.

static void TGameState::RegisterHandler (TGameStateHandler * *handler*) `[static]`

Register a game state handler.

Parameters:

handler Handler to register.

15.23.3 Member Data Documentation

const char* TGameState::kCmdLine `[static]`

Key to pass command line into attached game monitor.

This allows plug-in to interpret its own command line options.

This option is handled by Playground; there's no need for a game to use it.

const char* TGameState::kLevel `[static]`

State key for setting current game level.

If your game has no concept of levels, then set this to a constantly increasing value.

const char* TGameState::kPause `[static]`

State key to let the system know your game has been paused or resumed.

Value is non-zero for pause and zero for resume.

const char* TGameState::kMute `[static]`

State key to indicate the game is in MUTE state.

Use this to enable or disable muting from within the game. Monitor QueryMuteGame() to determine actual game mute settings.

const char* TGameState::kBreakNow [static]

Key to signify that the game is at a pause in the action and can be stopped.

This should ideally be triggered at the end of each level. Value is ignored.

const char* TGameState::kScore [static]

State key for score.

In games with no score, this can be safely ignored. Call this function every game loop.

const char* TGameState::kGameMode [static]

State key for game mode.

Should be set when user has selected a mode and is about to start playing. Value should be a one word description of the game mode, like "Story" or "Endless".

const char* TGameState::kGameOver [static]

State key for game over.

Value is ignored. Be sure to set the final score value BEFORE calling kGameOver.

const char* TGameState::kNoMoreContent [static]

State key for no more free content in a Web version.

Value is ignored. Set this after the player has beaten the last level available in a trial version of the game or has otherwise played past the end of the available content.

const char* TGameState::kDownloadButton [static]

Download button pressed: In a Web game, there will be a "download (and buy) the real game" button.

When it's pressed, call SetState(kDownloadButton).

const char* TGameState::kUpsellFrequency [static]

The frequency you should use to display upsell screens.

The game should show an upsell screen at the end of any level divisible by this number. Default is 0, which indicates that the game should use its default upsell frequency.

const char* TGameState::kRestartButton [static]

Query this value to determine whether a restart button should be displayed instead of an "OK" button on an upsell screen.

In the case that you've displayed a restart button, call SetState() with this value to indicate the restart button has been pressed.

const char* TGameState::kSystemMode [static]

Key used to query the current system mode(s).

Return result may have several modes separated by commas, including "web", "msn", "zylom", "metrics", etc.

const char* TGameState::kMaxLevel `[static]`

Key used to query what level the game should cap out at; used by some web providers to vary how many levels can be played.

Default is -1, which indicates no limit.

15.24 TGameStateHandler Class Reference

`#include <pf/gamestatehandler.h>`

15.24.1 Detailed Description

A handler for game state actions.
Typically implemented by Playground or PlayFirst.

Public Member Functions

- virtual ~TGameStateHandler ()
  Virtual Destructor.
- virtual bool SetState (const char *key, const char *value)=0
  Handle SetState().
- virtual const char * GetState (const char *key, const char *value)
  Get the value of a state.
- virtual bool QueryRestartMode ()
  Handle QueryRestartMode().
- virtual bool QueryJumpToMenu ()
  Handle QueryJumpToMenu().
- virtual bool QueryMuteGame ()
  Handle QueryMuteGame().
- virtual bool QueryPauseGame ()
  Allow the handler to force the pause of the game; this is handled internally by Playground.

15.24.2 Member Function Documentation

virtual bool TGameStateHandler::SetState (const char * *key*, const char * *value*) `[pure virtual]`

Handle SetState().
Handler can modify the key or value.

Parameters:

key State to set.
value New state value.

Returns:

True if the state has been handled, and you'd like to eat the state. If you return true, it will not perform any other operations on this state, though it will still be cached by TGameState.

virtual const char* TGameStateHandler::GetState (const char * *key*, const char * *value*) `[virtual]`

Get the value of a state.

Parameters:

key Value to retrieve.
value Cached value (the last value set at this key).

Returns:

Potentially modified state value.

virtual bool TGameStateHandler::QueryRestartMode () `[virtual]`

Handle QueryRestartMode().

Returns:

True to request a restart.

virtual bool TGameStateHandler::QueryJumpToMenu () `[virtual]`

Handle QueryJumpToMenu().

Returns:

True to jump to the main menu.

virtual bool TGameStateHandler::QueryMuteGame () `[virtual]`

Handle QueryMuteGame().

Returns:

True to mute the game.

virtual bool TGameStateHandler::QueryPauseGame () `[virtual]`

Allow the handler to force the pause of the game; this is handled internally by Playground.

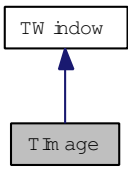
Returns:

True to pause the game

15.25 TImage Class Reference

`#include <pf/image.h>`

Inheritance diagram for TImage:



15.25.1 Detailed Description

The TImage class is a TWindow that contains and draws a TTexture.

It is not intended that the windowing system be used to render sprites in your game–that's what the sprite system is for. Among other things, there is a limited flexibility in rendering options.

A TImage will automatically set its kOpaque flag based on SetAlpha()–a SetAlpha() of less than 1 will reset the flag. In Lua initialization you can also use "alpha=true" to reset the flag.

Public Member Functions

- TImage (bool staticImage=true)
  Default Constructor.
- virtual ~TImage ()
  Destructor.
- void SetTexture (TTextureRef texture, TReal scale=1.0f)
  Set the image texture.
- TTextureRef GetTexture ()
  Get the current image texture.
- void SetAlpha (TReal alpha)
  Set the alpha for this image.
- TReal GetAlpha ()
  Get the current alpha of this image.
- TReal GetScale ()
  Get the current scale of this TImage.
- void SetRotate (bool rotate)
  Set this image to be rotated right (clockwise) by 90 degrees.
- void SetDrawFlags (uint32_t flags)

Set the draw flags for this image (including flip horizontal and flip vertical).

- virtual void Init (TWindowStyle &style)
  Initialize the Window.

Event Handlers

Functions to override to handle events in a window.

- virtual void Draw ()
  Draw the window.

15.25.2 Constructor & Destructor Documentation

TImage::TImage (bool *staticImage* = `true`)

Default Constructor.

Parameters:

staticImage Default behavior assumes that the image will change infrequently, and therefore sets kInfrequentChanges and tries to cache the image.

Create a TImage with staticImage set to false for an image that will change frequently.

15.25.3 Member Function Documentation

virtual void TImage::Draw () `[virtual]`

Draw the window.

TImage can draw only a portion of the window when only a part of the image needs to be redrawn.

Reimplemented from TWindow.

void TImage::SetTexture (TTextureRef *texture*, TReal *scale* = `1.0f`)

Set the image texture.

Parameters:

texture New texture.
scale Amount to scale texture by when drawing

TTextureRef TImage::GetTexture ()

Get the current image texture.

Returns:

A reference to the bound image.

void TImage::SetAlpha (TReal *alpha*)

Set the alpha for this image.

Parameters:

alpha Alpha value. 1.0==opaque.

TReal TImage::GetAlpha ()

Get the current alpha of this image.

Returns:

A value from 0 (transparent) to 1 (opaque).

TReal TImage::GetScale ()

Get the current scale of this TImage.

Returns:

Scale from 0 - 1

void TImage::SetRotate (bool *rotate*)

Set this image to be rotated right (clockwise) by 90 degrees.

Parameters:

rotate True to rotate.

void TImage::SetDrawFlags (uint32_t *flags*)

Set the draw flags for this image (including flip horizontal and flip vertical).

Parameters:

flags Draw flags to use: A combination of TDrawSpec::kFlipVertical and/or TDrawSpec::kFlipHorizontal.

virtual void TImage::Init (TWindowStyle & *style*) `[virtual]`

Initialize the Window.

Called by the system only in Lua initialization.

When you create your own custom window, this is where you put your own custom initialization that needs to happen before children are created. Fundamental window initialization is handled in every class by this function, so **when you override this function you almost always want to call your base class to handle base class initialization.**

Parameters:

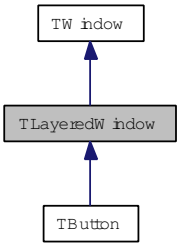
style The Lua style that was in effect when this window was created. This style contains all parameters specified explicitly for the window as well as parameters defined in the current style. Parameters set locally override ones in the style.

Reimplemented from TWindow.

15.26 TLayeredWindow Class Reference

`#include <pf/layeredwindow.h>`

Inheritance diagram for TLayeredWindow:



15.26.1 Detailed Description

A TLayeredWindow is a TWindow with multiple layers which can be switched between.
This can be used for animation, or for button presses, for instance.

Public Types

- enum { **kAll** = -1 }

Public Member Functions

- TLayeredWindow (uint32_t numberOfLayers)
  Default Constructor.
- virtual ~TLayeredWindow ()
  Destructor.
- void SetCurrentLayer (int32_t layer=kAll)
  Select the layer of this window to display or add children to.
- int32_t GetCurrentLayer ()
  Get the current active layer.
- uint32_t GetNumLayers ()
  Get the number of layers.
- virtual void OrphanChild (TWindow *child)
  Remove a child from this window, releasing ownership.
- virtual bool AdoptChild (TWindow *child, bool initWindow=true)
  Add a child to this window.

15.26.2 Constructor & Destructor Documentation

TLayeredWindow::TLayeredWindow (uint32_t *numberOfLayers*)

Default Constructor.

Parameters:

numberOfLayers Number of layers to create.

15.26.3 Member Function Documentation

void TLayeredWindow::SetCurrentLayer (int32_t *layer* = `kAll`)

Select the layer of this window to display or add children to.

Children added when one layer is selected will be added only to that layer, and therefore only displayed when that layer is selected.

Parameters:

layer The new active layer. Set to kAll to add children to all layers; layer 0 will be displayed in that state.

int32_t TLayeredWindow::GetCurrentLayer ()

Get the current active layer.

Returns:

An zero-based index to the current active layer.

uint32_t TLayeredWindow::GetNumLayers ()

Get the number of layers.

Returns:

The number of layers associated with this window.

virtual void TLayeredWindow::OrphanChild (TWindow * *child*) `[virtual]`

Remove a child from this window, releasing ownership.

Removes this child from *all* window layers.

Parameters:

child child to remove.

See also:

TWindow::OrphanChild

Reimplemented from TWindow.

virtual bool TLayeredWindow::AdoptChild (TWindow * *child*, bool *initWindow* = `true`) `[virtual]`

Add a child to this window.

Adds it to the current window layer only.

Parameters:

child Child to add.
initWindow Whether to run the init function OnNewParent() on this window.

Returns:

True if successful. False if OnNewParent() returned false, in which case child was NOT added.

See also:

TWindow::AdoptChild

Reimplemented from TWindow.

15.27 TLight Struct Reference

`#include <pf/light.h>`

15.27.1 Detailed Description

A 3d light.

Public Types

- enum ELightType { kDirectional, kPointSource, kSpotLight }
  Light types.

Public Member Functions

- TLight ()
  Default constructor, which resets the light parameters to a white directional light.

Public Attributes

- ELightType mType
  Type of light.
- TColor mDiff
  Diffuse value.
- TColor mSpec
  Specular value.
- TColor mAmb
  Ambient value.
- TVec3 mPos
  Position of light. No meaning for directional light sources.
- TVec3 mDir
  Direction of light. No meaning for point source light sources.
- TReal mRange
  Effective range of light. No meaning for directional light sources.
- TReal mAttenuation [3]
  Attenuation factors: Constant, Linear, and Quadratic.

- TReal mTheta

  Angle of inner cone of spotlight, in radians.

- TReal mPhi

  Angle of outer edge of spotlight dropoff, in radians.

15.27.2 Member Enumeration Documentation

enum TLight::ELightType

Light types.

Enumerator:

kDirectional A directional light source.
kPointSource A point source light.
kSpotLight A spotlight.

15.27.3 Member Data Documentation

TReal TLight::mAttenuation[3]

Attenuation factors: Constant, Linear, and Quadratic.

Attenuation is calculated based on the following equation:

$$A = \frac{1}{a_0 + a_1 D + a_2 D^2}$$

Where $a_0 - a_2$ are mAttenuation[0]-mAttenuation[2], and D is the distance of the light source to the vertex, normalized to 0,1 over the range of the light.

Referenced by TLight().

15.28 TLitVert Struct Reference

`#include <pf/vertexset.h>`

15.28.1 Detailed Description

3d untransformed, lit vertex.

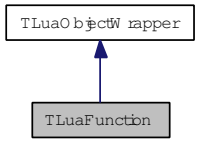
Public Attributes

- TVec3 pos
  Position in 3d space.
- uint32_t RESERVED
  UNUSED (must be zero).
- TColor32 color
  Vertex color.
- TColor32 specular
  Vertex specular component.
- TVec2 uv
  Vertex texture coordinate.

15.29 TLuaFunction Class Reference

`#include <pf/pflua.h>`

Inheritance diagram for TLuaFunction:



15.29.1 Detailed Description

A wrapper for a Lua function.

Public Member Functions

- TLuaFunction (lua_State *state)
  Constructor.
- void Call ()
  Call the function.
- bool IsFunction ()
  Test to verify that we're actually bound to a function.

15.29.2 Constructor & Destructor Documentation

TLuaFunction::TLuaFunction (lua_State * *state*)

Constructor.

Parameters:

state State to extract Lua function from. Pulls the top item off the stack and binds it to this TLuaFunction.

15.29.3 Member Function Documentation

bool TLuaFunction::IsFunction ()

Test to verify that we're actually bound to a function.

Returns:

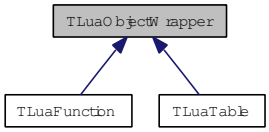
True if our bound object is a Lua function.

References TLuaObjectWrapper::Push().

15.30 TLuaObjectWrapper Class Reference

`#include <pf/pflua.h>`

Inheritance diagram for TLuaObjectWrapper:



15.30.1 Detailed Description

Wrap a Lua object for use within C++ code.

Keeps a reference to the Lua object so it won't be garbage collected.

Public Member Functions

- TLuaObjectWrapper (lua_State *state)
  Initialize our function with the object found on top of the stack.
- TLuaObjectWrapper (TLuaObjectWrapper &other)
  Copy constructor.
- virtual ~TLuaObjectWrapper ()
  Destructor.
- void Push ()
  Push our object onto the stack.
- bool IsString ()
  Is this object a string?
- bool IsNumber ()
  Is the object a number?
- bool IsTable ()
  Is the object a table?
- str AsString ()
  Convert the object to a string.
- lua_Number AsNumber ()
  Convert the object to a number.
- lua_State * GetState ()
  Get the current state associated with this object.

Protected Attributes

- lua_State * **mState**

15.30.2 Constructor & Destructor Documentation

TLuaObjectWrapper::TLuaObjectWrapper (lua_State * *state*)

Initialize our function with the object found on top of the stack.

Pops object from stack.

Parameters:

state Lua state.

TLuaObjectWrapper::TLuaObjectWrapper (TLuaObjectWrapper & *other*)

Copy constructor.

Parameters:

other State to copy.

References Push().

15.30.3 Member Function Documentation

void TLuaObjectWrapper::Push ()

Push our object onto the stack.

Note a Lua object can only be pushed onto the stack of the Lua interpreter that it was extracted from and originally created in.

Referenced by TLuaFunction::Call(), TLuaTable::GetSize(), TLuaFunction::IsFunction(), and TLuaObjectWrapper().

bool TLuaObjectWrapper::IsString ()

Is this object a string?

Returns:

True if it is a string.

bool TLuaObjectWrapper::IsNumber ()

Is the object a number?

Returns:

True if the object is a number.

bool TLuaObjectWrapper::IsTable ()

Is the object a table?

Returns:

True if a table

Referenced by TLuaTable::TLuaTable().

str TLuaObjectWrapper::AsString ()

Convert the object to a string.

Returns:

A string representation of the object, if one is available.

lua_Number TLuaObjectWrapper::AsNumber ()

Convert the object to a number.

Returns:

A numeric representation of the object, if one is available.

lua_State* TLuaObjectWrapper::GetState ()

Get the current state associated with this object.

Returns:

A Lua state.

15.31 TLuaParticleSystem Class Reference

`#include <pf/luaparticlesystem.h>`

15.31.1 Detailed Description

A particle system driven by Lua scripts.
Particle system documentation is at A Lua-Driven Particle System

Public Member Functions

- TLuaParticleSystem (TParticleRenderer *r=NULL)
  Default Constructor.
- virtual ~TLuaParticleSystem ()
  Destructor.
- void Draw (const TVec3 &at)
  Draw the system.
- virtual bool Init (str spec)
  An initializer with a client-defined specification.
- void NewScript ()
  Reset the script to a virgin state.
- virtual void Update (int ms)
  Update particle system.
- void RegisterFunction (str name, PFClassId processId)
  Register a function type with the particle system.
- void AdoptFunctionInstance (str name, TParticleFunction *function)
  Register a function type with the particle system.
- int32_t RegisterDataSource (str name, TParticleFunction *function)
  Register a data source.
- TParticleFunction * GetDataSource (int32_t source)
  Get a registered data source by index.
- ParticleMember Allocate (uint8_t size)
  Allocate a particle variable.
- TParticleFunctionRef GetParticleFunction (str type)
  Get a new instance of a particle function associated with a named type.

- TScript * GetScript ()
  Get a pointer to the attached particle system script.
- TScriptCodeRef GetScriptCode ()
  Get a reference to the asset-type TScriptCodeRef.
- bool IsDone ()
  Is the particle system done?
- void SetDone ()
  Flag the particle system as done.
- void ResetDone ()
  Not done any more!
- uint32_t GetParticleCount ()
  Get number of current active particles.
- void SetAlpha (float a)
  Set the alpha of this particle system.
- TParticleRenderer * GetParticleRenderer ()
  Get the embedded particle renderer.
- TParticleRenderer * GetRenderer ()
  Get a pointer to the currently attached TParticleRenderer.

Static Public Member Functions

- static TRandom * GetRandom ()
  Get the particle system random number generator.
- static void SetErrorHandler (TFailure *failureHandler)
  Bind an error handler to all TLuaParticleSystems.
- static TFailure * GetErrorHandler ()
  Get the current error handler.

Classes

- struct **PInstruction**

15.31.2 Constructor & Destructor Documentation

TLuaParticleSystem::TLuaParticleSystem (TParticleRenderer * *r* = `NULL`)

Default Constructor.

Parameters:

r Particle renderer to use. If NULL, will create a T2dParticleRenderer.

15.31.3 Member Function Documentation

void TLuaParticleSystem::Draw (const TVec3 & *at*)

Draw the system.

Parameters:

at Where to draw it. For 2d particles, the third component just sets the Z-depth to draw the particles.

virtual bool TLuaParticleSystem::Init (str *spec*) `[virtual]`

An initializer with a client-defined specification.

Note that this doesn't call NewScript() for you–it's acting more like a "Load()" call, and will likely have its name changed in 4.1 to better reflect its behavior.

Parameters:

spec Client spec.

virtual void TLuaParticleSystem::Update (int *ms*) `[virtual]`

Update particle system.

Parameters:

ms Number of milliseconds to advance system.

void TLuaParticleSystem::RegisterFunction (str *name*, PFClassId *processId*)

Register a function type with the particle system.

This interface allows you to create a new operator that can be used in the particle system.

Parameters:

name Name of process to use (or to replace).
processId The PFClassId of the class that provides the function. See Type Information and Casting for more information.

See also:

AdoptFunctionInstance

void TLuaParticleSystem::AdoptFunctionInstance (str *name*, TParticleFunction * *function*)

Register a function type with the particle system.

This interface allows you to create a new operator that can be used in the particle system.

Adopt semantics are used: You need to create a TParticleFunction-derived class instance and pass it in. Lifetime management is then handled by the TLuaParticleSystem.

For example:

```C++
AdoptFunctionInstance("fNewFunction", new MyParticleFunction );
```

Parameters:

name Name of process to use (or to replace).
function A new instance of the class that provides the function. The TLuaParticleSystem will delete this function on destruction.

See also:

RegisterFunction

int32_t TLuaParticleSystem::RegisterDataSource (str *name*, TParticleFunction * *function*)

Register a data source.

Parameters:

name Name of data source in Lua, e.g., "dSpriteVelocity"
function Class that contains the data accessor

Returns:

Data source index.

TParticleFunction* TLuaParticleSystem::GetDataSource (int32_t *source*)

Get a registered data source by index.

Parameters:

source Data source index to read (must be negative).

Returns:

The data source.

ParticleMember TLuaParticleSystem::Allocate (uint8_t *size*)

Allocate a particle variable.

This function is typically called from Lua to allocate custom particle variables, and by the renderer to allocate the default particle members.

Parameters:

size Size of particle variable (1-4) in TReal values.

Returns:

A ParticleMember struct that refers to the newly allocated member.

TParticleFunctionRef TLuaParticleSystem::GetParticleFunction (str *type*)

Get a new instance of a particle function associated with a named type.

Parameters:

type A registered particle function type.

Returns:

A reference-counted pointer to a newly created particle function.

TScript* TLuaParticleSystem::GetScript ()

Get a pointer to the attached particle system script.

Returns:

A pointer to the current script.

TScriptCodeRef TLuaParticleSystem::GetScriptCode ()

Get a reference to the asset-type TScriptCodeRef.
This allows Lua scripts to be pre-loaded once and maintained in the asset system.

Returns:

A reference to the TScriptCode object that contains the (compiled) Lua code.

bool TLuaParticleSystem::IsDone ()

Is the particle system done?

Returns:

True if done; false otherwise.

uint32_t TLuaParticleSystem::GetParticleCount ()

Get number of current active particles.

Returns:

The current number of particles active.

void TLuaParticleSystem::SetAlpha (float *a*)

Set the alpha of this particle system.

Parameters:

a Alpha (transparency) of this system. 0 is transparent, 1 opaque.

static TRandom* TLuaParticleSystem::GetRandom () [static]

Get the particle system random number generator.

Returns:

A pointer to the particle system random generator.

TParticleRenderer* TLuaParticleSystem::GetParticleRenderer ()

Get the embedded particle renderer.

Returns:

A pointer to the TParticleRenderer associated with this particle system.

static void TLuaParticleSystem::SetErrorHandler (TFailure * *failureHandler*) [static]

Bind an error handler to all TLuaParticleSystems.

Parameters:

failureHandler Handler to add.

static TFailure* TLuaParticleSystem::GetErrorHandler () [static]

Get the current error handler.

Returns:

A pointer to the current error handler, if any.

TParticleRenderer* TLuaParticleSystem::GetRenderer ()

Get a pointer to the currently attached TParticleRenderer.

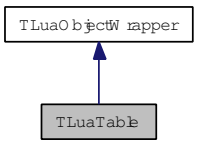
Returns:

A pointer to the particle renderer.

15.32 TLuaTable Class Reference

`#include <pf/luatable.h>`

Inheritance diagram for TLuaTable:



15.32.1 Detailed Description

A wrapper for Lua table access in C++.

Public Member Functions

- TLuaTable (lua_State *state)
  Constructor.
- TLuaTable (TLuaObjectWrapper &lobj)
  Construct from a TLuaObjectWrapper.
- uint32_t GetSize ()
  How many elements in the indexed portion of the table?
- str GetString (const char *key, str defaultValue="")
  Get a string from the table.
- str GetString (lua_Number key, str defaultValue="")
  Get a string from the table.
- lua_Number GetNumber (const char *key, lua_Number defaultValue=0)
  Get a lua_Number from a key in the table.
- lua_Number GetNumber (lua_Number key, lua_Number defaultValue=0)
  Get a lua_Number from a key in the table.
- void Assign (str key, str value)
  Assign a string to a string key.
- void Assign (lua_Number key, str value)
  Assign a string to a numeric key.
- void Assign (str key, lua_Number value)
  Assign a number to a string key.

- void Assign (lua_Number key, lua_Number value)

  Assign a number to a numeric key.

- void Assign (str key, bool value)

  Assign a boolean value to a string key.

- void Assign (lua_Number key, bool value)

  Assign a boolean value to a numeric key.

- void Assign (lua_Number key, TLuaObjectWrapper *luaObject)

  Assign a TLuaObjectWrapper-wrapped Lua object to a numeric key.

- void Assign (str key, TLuaObjectWrapper *luaObject)

  Assign a TLuaObjectWrapper-wrapped Lua object to a string key.

- void Erase (str key)

  Erase the key/value pair located at key in the table.

- void Erase (lua_Number key)

  Erase the key/value pair located at key in the table.

- bool GetBoolean (const char *key, bool defaultValue=false)

  Get a boolean value from a key in the table.

- TLuaFunction * GetFunction (const char *key)

  Get a TLuaFunction from a key in the table.

- TLuaTable * GetTable (const char *key)

  Acquire an embedded Lua table from within this table.

- TLuaTable * GetTable (lua_Number key)

  Acquire an embedded Lua table from within this table.

- TColor GetColor (str key, const TColor &defaultValue=TColor(0, 0, 0, 0))

  Get a TColor from a table of four values.

- TColor GetColor (lua_Number key, const TColor &defaultValue=TColor(0, 0, 0, 0))

  Get a TColor from a table of four values.

- TLuaObjectWrapper * GetNext (TLuaObjectWrapper **key)

  Takes a key and returns the key/value pair for the next key in the table.

- bool PushValue (const char *key)

  Push the value at a particular string key onto the Lua stack.

- bool PushValue (lua_Number key)

  Push the value at a particular numeric key onto the Lua stack.

Static Public Member Functions

- static TLuaTable * Create (lua_State *state)
  Create a new table, and wrap it in a TLuaTable.
- static void DeferDelete (TLuaTable *table)
  Internal function to defer the delete of a table until the next event loop.

15.32.2 Constructor & Destructor Documentation

TLuaTable::TLuaTable (lua_State * *state*)

Constructor.

Parameters:

state The lua_State pointer of the table. Use GetState() on a TScript to get the current state.

TLuaTable::TLuaTable (TLuaObjectWrapper & *lobj*)

Construct from a TLuaObjectWrapper.

Parameters:

lobj Object wrapper that presumably wraps a table.

References ASSERT, and TLuaObjectWrapper::IsTable().

15.32.3 Member Function Documentation

static TLuaTable* TLuaTable::Create (lua_State * *state*) `[static]`

Create a new table, and wrap it in a TLuaTable.

Parameters:

state State of Lua interpreter to use.

Returns:

A new (empty) TLuaTable. You're responsible for deleting it.

uint32_t TLuaTable::GetSize ()

How many elements in the indexed portion of the table?

Returns:

Number of elements in the array portion of the table.

References TLuaObjectWrapper::Push().

str TLuaTable::GetString (const char * *key*, str *defaultValue* = "")

Get a string from the table.

Parameters:

key The key to the string.
defaultValue Default value if key isn't found.

Returns:

A copy of the string, if one exists at that key. Otherwise an empty string.

str TLuaTable::GetString (lua_Number *key*, str *defaultValue* = "")

Get a string from the table.

Parameters:

key The key to the string.
defaultValue Default value if key isn't found.

Returns:

A copy of the string, if one exists at that key. Otherwise an empty string.

lua_Number TLuaTable::GetNumber (const char * *key*, lua_Number *defaultValue* = 0)

Get a lua_Number from a key in the table.

Parameters:

key Key to look up.
defaultValue Default value if key isn't found.

Returns:

A lua_Number, if one is found. Zero otherwise.

lua_Number TLuaTable::GetNumber (lua_Number *key*, lua_Number *defaultValue* = 0)

Get a lua_Number from a key in the table.

Parameters:

key Key to look up.
defaultValue Default value if key isn't found.

Returns:

A lua_Number, if one is found. Zero otherwise.

void TLuaTable::Assign (str *key*, str *value*)

Assign a string to a string key.

Parameters:

key Key to set.
value Value to set.

void TLuaTable::Assign (lua_Number *key*, str *value*)

Assign a string to a numeric key.

Parameters:

key Key to set.
value Value to set.

void TLuaTable::Assign (str *key*, lua_Number *value*)

Assign a number to a string key.

Parameters:

key Key to set.
value Value to set.

void TLuaTable::Assign (lua_Number *key*, lua_Number *value*)

Assign a number to a numeric key.

Parameters:

key Key to set.
value Value to set.

void TLuaTable::Assign (str *key*, bool *value*)

Assign a boolean value to a string key.

Parameters:

key Key to set.
value Value to set.

void TLuaTable::Assign (lua_Number *key*, bool *value*)

Assign a boolean value to a numeric key.

Parameters:

key Key to set.

value Value to set.

void TLuaTable::Assign (lua_Number *key*, TLuaObjectWrapper * *luaObject*)

Assign a TLuaObjectWrapper-wrapped Lua object to a numeric key.

Parameters:

key Key to assign object to in table.
luaObject Object to assign.

void TLuaTable::Assign (str *key*, TLuaObjectWrapper * *luaObject*)

Assign a TLuaObjectWrapper-wrapped Lua object to a string key.

Parameters:

key Key to assign object to in table.
luaObject Object to assign.

void TLuaTable::Erase (str *key*)

Erase the key/value pair located at key in the table.

Parameters:

key Key to erase.

void TLuaTable::Erase (lua_Number *key*)

Erase the key/value pair located at key in the table.

Parameters:

key Key to erase.

bool TLuaTable::GetBoolean (const char * *key*, bool *defaultValue* = `false`)

Get a boolean value from a key in the table.

Parameters:

key Key to look up.
defaultValue Default value if key isn't found.

Returns:

The value of the key as a boolean (using lua_toboolean()) if the key is found. False otherwise.

TLuaFunction* TLuaTable::GetFunction (const char * *key*)

Get a TLuaFunction from a key in the table.

Parameters:

key Key to look up.

Returns:

A TLuaFunction, if that slot in the table has a lua function. NULL otherwise. You should eventually delete this pointer.

TLuaTable* TLuaTable::GetTable (const char * *key*)

Acquire an embedded Lua table from within this table.

Parameters:

key Key where table is stored.

Returns:

A new TLuaTable pointer which you must eventually delete.

TLuaTable* TLuaTable::GetTable (lua_Number *key*)

Acquire an embedded Lua table from within this table.

Parameters:

key Key where table is stored.

Returns:

A new TLuaTable pointer which you must eventually delete.

TColor TLuaTable::GetColor (str *key*, const TColor & *defaultValue* = TColor(0, 0, 0, 0))

Get a TColor from a table of four values.

In Lua, if you define a color using the Color() function, you define it using values from 0-255. If you define using FColor(), the values are from 0-1.

Parameters:

key Key of table.
defaultValue Default value to return if no table found.

Returns:

A TColor.

TColor TLuaTable::GetColor (lua_Number *key*, const TColor & *defaultValue* = TColor(0, 0, 0, 0))

Get a TColor from a table of four values.

In Lua, if you define a color using the Color() function, you define it using values from 0-255. If you define using FColor(), the values are from 0-1.

Parameters:

key Key of table.
defaultValue Default value to return if no table found.

Returns:

A TColor.

TLuaObjectWrapper* TLuaTable::GetNext (TLuaObjectWrapper ** *key*)

Takes a key and returns the key/value pair for the next key in the table.

The key you pass in should start out NULL to start the iteration.

Warning:

If you stop an iteration in the middle, you're responsible for deleting the last key you've received in addition to the normal deletion of the last value.

Example Usage

C++
```
TLuaObjectWrapper *key=NULL;
TLuaObjectWrapper *value;
while ( (value=options->GetNext(&key))!=NULL)
{
     // Here we have a key/value pair.
     str keyString = key->GetString(); str valueString =
     value->GetString(); // this gets it as a string; you can also GetNumber()
     // if you need a table, you can TLuaTable table(value);

     delete value; // Delete the value once you've extracted it.
}
```

Parameters:

key Pointer to variable to receive next key. Initialize it to NULL to start the iteration, and leave the previous key in place to iterate.

Returns:

A pointer to a TLuaObjectWrapper. You must delete this pointer when you're done with it. Returns NULL after the last table item.

bool TLuaTable::PushValue (const char * *key*)

Push the value at a particular string key onto the Lua stack.

Parameters:

key Key of value to retrieve.

bool TLuaTable::PushValue (lua_Number *key*)

Push the value at a particular numeric key onto the Lua stack.

Parameters:

key Key of value to retrieve.

static void TLuaTable::DeferDelete (TLuaTable * *table*) `[static]`

Internal function to defer the delete of a table until the next event loop.

Parameters:

table Table to delete later.

15.33 TMat3 Class Reference

`#include <pf/mat.h>`

15.33.1 Detailed Description

2d Matrix with 2x2 rotation component and TVec2 offset component.

Scaling and rotation

- TMat3 & Scale (TReal x, TReal y)
  Scale this matrix in two dimensions.
- TMat3 & Scale (TReal s)
  Scale this matrix.
- TMat3 & Rotate (TReal radians)
  Rotate this matrix in place.
- static TMat3 GetRotation (TReal radians)
  Get a rotation matrix.

Public Member Functions

Construction and Initialization.

- TMat3 ()
  Default constructor.
- TMat3 (TReal m00, TReal m01, TReal m02, TReal m10, TReal m11, TReal m12, TReal m20, TReal m21, TReal m22)
  Construct from individual values.
- TMat3 (TVec3 v0, TVec3 v1, TVec3 v2)
  Construct from three vectors.
- TMat3 (const TMat3 &rhs)
  Copy construction.
- TMat3 & operator= (const TMat3 &rhs)
  Assignment.
- TMat3 & Identity ()
  Initialize to an identity.

Accessors

- TVec3 & operator[ ] (TIndex i)
  Array Accessor.
- const TVec3 & operator[ ] (TIndex i) const
  Array Accessor.
- TIndex Dim () const
  Dimensions of this array.

Assignment Operators

- TMat3 & operator+= (const TMat3 &rhs)
  Assignment operator.
- TMat3 & operator-= (const TMat3 &rhs)
  Assignment operator.
- TMat3 & operator∗= (const TMat3 &rhs)
  Assignment operator.
- TMat3 & operator∗= (TReal rhs)
  Assignment operator.
- TMat3 & operator/= (TReal rhs)
  Assignment operator.

Related Functions

(Note that these are not member functions.)

- bool operator== (const TMat3 &lhs, const TMat3 &rhs)
  Equality.
- bool operator!= (const TMat3 &lhs, const TMat3 &rhs)
  Inequality.
- TVec3 operator∗ (const TMat3 &lhs, const TVec3 &rhs)
  Matrix multiplication with a vector.
- TVec2 operator∗ (const TMat3 &lhs, const TVec2 &rhs)
  Matrix multiplication with a vector.
- TVec3 operator∗ (const TVec3 &lhs, const TMat3 &rhs)
  Matrix multiplication with a vector.
- TVec2 operator∗ (const TVec2 &lhs, const TMat3 &rhs)
  Matrix multiplication with a vector.
- TMat3 operator∗ (const TMat3 &lhs, TReal rhs)
  Member-wise scaling of the vector.

- TVec2 Multiply2x2 (const TMat3 &lhs, const TVec2 &rhs)

  A restricted 2x2 matrix vector multiply.

- TVec2 Multiply2x2 (const TVec2 &lhs, const TMat3 &rhs)

  A restricted 2x2 matrix vector multiply.

- TMat3 Multiply2x2 (const TMat3 &lhs, const TMat3 &rhs)

  A restricted 2x2 matrix-matrix multiply.

- TVec2 operator% (const TVec2 &lhs, const TMat3 &rhs)

  Restricted-multiply operator.

- TVec2 operator% (const TMat3 &lhs, const TVec2 &rhs)

  Restricted-multiply operator.

- TMat3 operator% (const TMat3 &lhs, const TMat3 &rhs)

  Restricted-multiply operator.

- TVec3 operator∗ (const TVec3 &lhs, const TMat4 &rhs)

  Matrix multiplication with a vector.

15.33.2 Constructor & Destructor Documentation

TMat3::TMat3 ()

Default constructor.
Initializes to identity.

15.33.3 Member Function Documentation

TVec3& TMat3::operator[ ] (TIndex *i*)

Array Accessor.

Parameters:

i The row number, from 0-2.

Returns:

a reference to the TVec3 that represents the row.

const TVec3& TMat3::operator[ ] (TIndex *i*) const

Array Accessor.

Parameters:

i The row number, from 0-2.

Returns:

a reference to the TVec3 that represents the row.

TMat3& TMat3::operator+= (const TMat3 & *rhs*)

Assignment operator.

Parameters:

rhs The right-hand-side of the assignment.

Returns:

a reference to this.

TMat3& TMat3::operator-= (const TMat3 & *rhs*)

Assignment operator.

Parameters:

rhs The right-hand-side of the assignment.

Returns:

a reference to this.

TMat3& TMat3::operator∗= (const TMat3 & *rhs*)

Assignment operator.

Parameters:

rhs The right-hand-side of the assignment.

Returns:

a reference to this.

TMat3& TMat3::operator∗= (TReal *rhs*)

Assignment operator.

Parameters:

rhs The right-hand-side of the assignment.

Returns:

a reference to this.

TMat3& TMat3::operator/= (TReal *rhs*)

Assignment operator.

Parameters:

rhs The right-hand-side of the assignment.

Returns:

a reference to this.

TMat3& TMat3::Scale (TReal *x*, TReal *y*)

Scale this matrix in two dimensions.

Assumes that this matrix is used as a 2d matrix.

Parameters:

x Scale in X dimension.
y Scale in Y dimension.

Returns:

this matrix.

Referenced by TDrawSpec::TDrawSpec().

TMat3& TMat3::Scale (TReal *s*)

Scale this matrix.

Assumes this matrix is used as a 2d matrix.

Parameters:

s Amount to scale X and Y axes.

Returns:

this matrix.

static TMat3 TMat3::GetRotation (TReal *radians*) `[static]`

Get a rotation matrix.

Parameters:

radians Rotation in radians.

Returns:

A new matrix that represents the given rotation frame.

TMat3& TMat3::Rotate (TReal *radians*)

Rotate this matrix in place.

Parameters:

radians Radians to rotate the matrix by. Only changes the orientation portion of the matrix; position is unchanged.

Returns:

A reference to this matrix.

15.33.4 Friends And Related Function Documentation

bool operator== (const TMat3 & *lhs*, const TMat3 & *rhs*) `[related]`

Equality.

Returns:

True on equal.

bool operator!= (const TMat3 & *lhs*, const TMat3 & *rhs*) `[related]`

Inequality.

Returns:

True on not equal.

TVec3 operator∗ (const TMat3 & *lhs*, const TVec3 & *rhs*) `[related]`

Matrix multiplication with a vector.

Returns:

$M * v$

TVec2 operator∗ (const TMat3 & *lhs*, const TVec2 & *rhs*) `[related]`

Matrix multiplication with a vector.

Returns:

$M * v$

TVec3 operator∗ (const TVec3 & *lhs*, const TMat3 & *rhs*) `[related]`

Matrix multiplication with a vector.

Returns:

$v^T * M$

TVec2 operator∗ (const TVec2 & *lhs*, const TMat3 & *rhs*) `[related]`

Matrix multiplication with a vector.

Returns:

$v^T * M$

TMat3 operator∗ (const TMat3 & *lhs*, TReal *rhs*) `[related]`

Member-wise scaling of the vector.

Note this won't necessarily do what you want if you're trying to produce a scale vector. See TMat3::Scale().

Returns:

$$\begin{vmatrix} a_{00} * s & a_{10} * s & a_{20} * s \\ a_{01} * s & a_{11} * s & a_{21} * s \\ a_{02} * s & a_{12} * s & a_{22} * s \end{vmatrix}$$

TVec2 Multiply2x2 (const TMat3 & *lhs*, const TVec2 & *rhs*) `[related]`

A restricted 2x2 matrix vector multiply.

Does the rotation/scaling but no translation of the vector.

Parameters:

lhs Matrix left-hand-side of the multiply
rhs Vector right-hand-side.

Returns:

Matrix[2x2]∗Vector

Referenced by operator%().

TVec2 Multiply2x2 (const TVec2 & *lhs*, const TMat3 & *rhs*) `[related]`

A restricted 2x2 matrix vector multiply.

Does the rotation/scaling but no translation of the vector.

Parameters:

lhs Vector left-hand-side of the multiply
rhs Matrix right-hand-side.

Returns:

Vector∗Matrix[2x2]

TMat3 Multiply2x2 (const TMat3 & *lhs*, const TMat3 & *rhs*) [related]

A restricted 2x2 matrix-matrix multiply.

Parameters:

lhs Matrix left-hand-side.
rhs Matrix right-hand-side.

Returns:

Matrix[2x2]*Matrix[2x2], with lhs[2] as the translation component.

TVec2 operator% (const TVec2 & *lhs*, const TMat3 & *rhs*) [related]

Restricted-multiply operator.

See also:

Multiply2x2()

Returns:

Multiply2x2(lhs,rhs)

References Multiply2x2().

TVec2 operator% (const TMat3 & *lhs*, const TVec2 & *rhs*) [related]

Restricted-multiply operator.

See also:

Multiply2x2()

Returns:

Multiply2x2(lhs,rhs)

References Multiply2x2().

TMat3 operator% (const TMat3 & *lhs*, const TMat3 & *rhs*) [related]

Restricted-multiply operator.

See also:

Multiply2x2()

Returns:

Multiply2x2(lhs,rhs)

References Multiply2x2().

TVec3 operator* (const TVec3 & *lhs*, const TMat4 & *rhs*) [related]

Matrix multiplication with a vector.

Returns:

$v^T * M$

15.34 TMat4 Class Reference

`#include <pf/mat.h>`

15.34.1 Detailed Description

3d Matrix with 3x3 rotation component and TVec3 offset component.

Public Types

- enum { **kDIM** = 4 }

Public Member Functions

Construction and Initialization.

- TMat4 ()
  Default constructor.
- TMat4 (TReal m00, TReal m01, TReal m02, TReal m03, TReal m10, TReal m11, TReal m12, TReal m13, TReal m20, TReal m21, TReal m22, TReal m23, TReal m30, TReal m31, TReal m32, TReal m33)
  Construct from individual values.
- TMat4 (TVec4 v0, TVec4 v1, TVec4 v2, TVec4 v3)
  Construct from vectors.
- TMat4 (const TMat4 &rhs)
  Copy construction.
- TMat4 & operator= (const TMat4 &rhs)
  Assignment operator.
- ~TMat4 ()
  Destruction.

Accessors

- TVec4 & operator[ ] (TIndex i)
  Array Accessor.
- const TVec4 & operator[ ] (TIndex i) const
  Array Accessor.

Assignment operators.

- TMat4 & operator+= (const TMat4 &rhs)
  Assignment.

- TMat4 & operator-= (const TMat4 &rhs)
  Assignment.
- TMat4 & operator*= (const TMat4 &rhs)
  Assignment.
- TMat4 & operator*= (TReal rhs)
  Assignment.
- TMat4 & operator/= (TReal rhs)
  Assignment.

Initializers.

- TMat4 & Identity ()
  Make this matrix the identity.
- TMat4 & LookAt (const TVec3 &pos, const TVec3 &at, const TVec3 &up)
  Change this matrix to be a view matrix that's oriented to "look at" a point.
- TMat4 & Perspective (TReal nearPlane, TReal farPlane, TReal fov, TReal aspect)
  Make this matrix a perspective matrix.
- TMat4 & OffsetPerspective (TReal nearPlane, TReal farPlane, TReal fov, TReal aspect, const TVec2 &offsets)
  Make this matrix an offset perspective matrix.
- TMat4 & Orthogonal (TReal left, TReal right, TReal bottom, TReal top, TReal zNear, TReal zFar)
  Make this matrix an orthogonal projection matrix that transforms objects within the given box into view coordinates.

Manipulators

- TMat4 & RotateAxis (const TVec3 &axis, TReal a)
  Rotate this matrix around an axis.
- TMat4 & RotateX (TReal a)
  Rotate this matrix around the X axis.
- TMat4 & RotateY (TReal a)
  Rotate this matrix around the Y axis.
- TMat4 & RotateYPR (TReal yaw, TReal pitch, TReal roll)
  Rotate this matrix around all three axes, given as Yaw, Pitch, and Roll.
- TMat4 & RotateZ (TReal a)
  Rotate this matrix around the Z axis.
- TMat4 & Scale (TReal x, TReal y, TReal z)
  Scale this matrix.
- TMat4 & Translate (const TVec3 &v)
  Translate this matrix by a vector.

Utility Mmbers

- TMat4 Transpose () const
  Return a transposed matrix.
- TMat4 Adjoint () const
  Matrix adjoint function.
- TMat4 Inverse () const
  Return an inverse of the matrix.
- TReal Determinant () const
  Calculate the determinant of the matrix.

Related Functions

(Note that these are not member functions.)

- TMat3 operator∗ (const TMat3 &lhs, const TMat3 &rhs)
  Matrix multiplication.
- bool operator== (const TMat4 &lhs, const TMat4 &rhs)
  Equality.
- bool operator!= (const TMat4 &lhs, const TMat4 &rhs)
  Inequality.
- TMat4 operator∗ (const TMat4 &lhs, const TMat4 &rhs)
  Matrix multiplication.
- TVec4 operator∗ (const TMat4 &lhs, const TVec4 &rhs)
  Matrix multiplication with a vector.
- TVec3 operator∗ (const TMat4 &lhs, const TVec3 &rhs)
  Matrix multiplication with a vector.
- TVec4 operator∗ (const TVec4 &lhs, const TMat4 &rhs)
  Matrix multiplication with a vector.
- TMat4 operator∗ (const TMat4 &lhs, TReal s)
  Matrix scaling.
- TMat4 operator/ (const TMat4 &lhs, TReal s)
  Matrix scaling.

15.34.2 Constructor & Destructor Documentation

TMat4::TMat4 ()

Default constructor.

Initializes to identity.

TMat4::TMat4 (const TMat4 & *rhs*)

Copy construction.

Parameters:

rhs Copy source.

15.34.3 Member Function Documentation

TMat4& TMat4::operator= (const TMat4 & *rhs*)

Assignment operator.

Parameters:

rhs Right hand side of the equation.

TVec4& TMat4::operator[ ] (TIndex *i*)

Array Accessor.

Parameters:

i The row number, from 0-2.

Returns:

a reference to the TVec4 that represents the row.

const TVec4& TMat4::operator[ ] (TIndex *i*) const

Array Accessor.

Parameters:

i The row number, from 0-2.

Returns:

a reference to the TVec4 that represents the row.

TMat4& TMat4::Identity ()

Make this matrix the identity.

Returns:

A reference to this.

TMat4& TMat4::LookAt (const TVec3 & *pos*, const TVec3 & *at*, const TVec3 & *up*)

Change this matrix to be a view matrix that's oriented to "look at" a point.

Parameters:

pos Viewer position.
at Point we're looking at.
up Local "up" vector.

Returns:

A reference to this.

TMat4& TMat4::Perspective (TReal *nearPlane*, TReal *farPlane*, TReal *fov*, TReal *aspect*)

Make this matrix a perspective matrix.

Parameters:

nearPlane Distance from viewer to the near plane.
farPlane Distance from viewer to the far plane. The smaller your ratio of far to near, the higher Z-buffer resolution you get.
fov The field of view in radians.
aspect The aspect ratio of the resulting view.

Returns:

A reference to this.

TMat4& TMat4::OffsetPerspective (TReal *nearPlane*, TReal *farPlane*, TReal *fov*, TReal *aspect*, const TVec2 & *offsets*)

Make this matrix an offset perspective matrix.

In order to display a partially clipped 3d window (say you wanted to allow a window to scroll off the screen), you need to have a perspective matrix that has been skewed to display part of a larger projection.

Parameters:

nearPlane Distance from viewer to the near plane.
farPlane Distance from viewer to the far plane. The smaller your ratio of far to near, the higher Z-buffer resolution you get.
fov The field of view in radians.
aspect The aspect ratio of the resulting view.
offsets The amounts to skew the x and y portions of the matrix. If you want to clip 20% of the left edge, you'd set offsets.x to 0.2. Similarly if you want to clip 20% of the bottom edge, you'd set y to -0.2.

Returns:

A reference to this.

TMat4& TMat4::Orthogonal (TReal *left*, TReal *right*, TReal *bottom*, TReal *top*, TReal *zNear*, TReal *zFar*)

Make this matrix an orthogonal projection matrix that transforms objects within the given box into view coordinates.

Parameters:

left Left side of box.
right Right side of box.
bottom Bottom of box.
top Top of box.
zNear Near side of box.
zFar Far side of box.

Returns:

A reference to this.

TMat4& TMat4::RotateAxis (const TVec3 & *axis*, TReal *a*)

Rotate this matrix around an axis.

Parameters:

axis A unit vector that defines an axis to rotate around.
a Number of radians to rotate matrix.

Returns:

A reference to this.

TMat4& TMat4::RotateX (TReal *a*)

Rotate this matrix around the X axis.

Parameters:

a Number of radians to rotate matrix.

Returns:

A reference to this.

TMat4& TMat4::RotateY (TReal *a*)

Rotate this matrix around the Y axis.

Parameters:

a Number of radians to rotate matrix.

Returns:

A reference to this.

TMat4& TMat4::RotateYPR (TReal *yaw*, TReal *pitch*, TReal *roll*)

Rotate this matrix around all three axes, given as Yaw, Pitch, and Roll.

Parameters:

yaw Number of radians to rotate matrix around Y axis.
pitch Number of radians to rotate matrix around X axis.
roll Number of radians to rotate matrix around Z axis.

Returns:

A reference to this.

TMat4& TMat4::RotateZ (TReal *a*)

Rotate this matrix around the Z axis.

Parameters:

a Number of radians to rotate matrix.

Returns:

A reference to this.

TMat4& TMat4::Scale (TReal *x*, TReal *y*, TReal *z*)

Scale this matrix.

Parameters:

x Scale to apply to X axis.
y Scale to apply to Y axis.
z Scale to apply to Z axis.

Returns:

A reference to this.

TMat4& TMat4::Translate (const TVec3 & *v*)

Translate this matrix by a vector.

Parameters:

v Amount to translate.

Returns:

A reference to this.

TMat4 TMat4::Transpose () const

Return a transposed matrix.

Returns:

A new transposed matrix.

TMat4 TMat4::Inverse () const

Return an inverse of the matrix.

Returns:

An inverse copy of the matrix.

TReal TMat4::Determinant () const

Calculate the determinant of the matrix.

Returns:

The determinant.

15.34.4 Friends And Related Function Documentation

TMat3 operator∗ (const TMat3 & *lhs*, const TMat3 & *rhs*) `[related]`

Matrix multiplication.

Returns:

$M_1 * M_2$

bool operator== (const TMat4 & *lhs*, const TMat4 & *rhs*) `[related]`

Equality.

Returns:

True on equal.

bool operator!= (const TMat4 & *lhs*, const TMat4 & *rhs*) `[related]`

Inequality.

Returns:

True on not equal.

TMat4 operator∗ (const TMat4 & *lhs*, const TMat4 & *rhs*) `[related]`

Matrix multiplication.

Returns:

$M_1 * M_2$

TVec4 operator∗ (const TMat4 & *lhs*, const TVec4 & *rhs*) `[related]`

Matrix multiplication with a vector.

Returns:

$M * v$

TVec3 operator∗ (const TMat4 & *lhs*, const TVec3 & *rhs*) `[related]`

Matrix multiplication with a vector.

Returns:

$M * v$

TVec4 operator∗ (const TVec4 & *lhs*, const TMat4 & *rhs*) `[related]`

Matrix multiplication with a vector.

Returns:

$M * v$

TMat4 operator∗ (const TMat4 & *lhs*, TReal *s*) `[related]`

Matrix scaling.

Returns:

$$\begin{vmatrix} a_{00} * s & a_{10} * s & a_{20} * s & a_{30} * s \\ a_{01} * s & a_{11} * s & a_{21} * s & a_{31} * s \\ a_{02} * s & a_{12} * s & a_{22} * s & a_{32} * s \\ a_{03} * s & a_{13} * s & a_{23} * s & a_{33} * s \end{vmatrix}$$

TMat4 operator/ (const TMat4 & *lhs*, TReal *s*) `[related]`

Matrix scaling.

Returns:

$(1/s) * M$

15.35 TMaterial Struct Reference

`#include <pf/pftypes.h>`

15.35.1 Detailed Description

A rendering material.

Public Attributes

- TColor mcDiff
  Diffuse value.
- TColor mcAmb
  Ambient value.
- TColor mcSpec
  Specular value.
- TColor mcEmit
  Emit (glow) value.
- TReal mfPower
  Specular reflectance. Zero to disable specular.

15.36 TMessage Class Reference

`#include <pf/message.h>`

15.36.1 Detailed Description

Application message base class.

Actual messages being passed around by the application will either use TMessage directly or derive from TMessage, depending on whether they need additional payload.

Warning:

Never use C++ or C style casting to coerce a TMessage to another type; always use GetCast<>(). When a message is sent from Lua, it is wrapped in a TLuaMessageWrapper, which will report the mType of the contained TMessage-derived message–and it has a GetCast operator that will give you the actual contained message. But casting it to your target object will result in undefined (and certainly incorrect) behavior, so it's best to always use GetCast.

Public Types

- enum EMessageID {
  kGeneric = 0, kCloseWindow = 1, kDefaultAction, kButtonPress,
  kPressAnyKey, kQuitNow, kModalClosed, kTextEditChanged,
  kCommandOnly, kSliderValChanged, kSliderMouseUp, kSliderPageUp,
  kSliderPageDown, kUserMessageBase = 1000 }
  System-level predefined message IDs.

Public Member Functions

- TMessage (int32_t type=kGeneric, str name="", TWindow *destination=NULL)
  Constructor.
- virtual ~TMessage ()
  Destructor.

Type Information and Casting

- PFClassId ClassId ()
  Get the ClassId.
- virtual bool IsKindOf (PFClassId type)
  Determine whether this message is derived from type.
- template<class TO>
  TO * GetCast ()
  Safely cast this message to another type.

Public Attributes

- int32_t mType

  The EMessageID of this message, or a user defined type (starting at kUserMessageBase).

- str mName

  Name of this message.

- TWindow * mDestination

  Optional TWindow destination.

15.36.2 Member Enumeration Documentation

enum TMessage::EMessageID

System-level predefined message IDs.

Start user message IDs at kUserMessageBase

Enumerator:

kGeneric A generic message–the derived class determines the type.
kCloseWindow Close the current window if you get this message ID.
kDefaultAction Trigger the default action (done by pressing "enter" equivalent).
kButtonPress Button pressed.
kPressAnyKey Unhandled user input.
kQuitNow Time to exit the application.
kModalClosed A modal window closed.
kTextEditChanged New information has been typed into/removed from a text edit field.
kCommandOnly This message is empty; it's being sent to run the accompanying command.
kSliderValChanged A slider value changed.
kSliderMouseUp The mouse has been released on a slider.
kSliderPageUp Someone clicked above the slider handle to create a virtual page-up.
kSliderPageDown Someone clicked below the slider handle to create a virtual page-down.
kUserMessageBase First ID available for client applications.

15.36.3 Constructor & Destructor Documentation

TMessage::TMessage (int32_t *type* = `kGeneric`, str *name* = `""`, TWindow * *destination* = `NULL`)

Constructor.

Parameters:

type Message type.
name Message name.
destination Optional destination window.

15.36.4 Member Function Documentation

PFClassId TMessage::ClassId ()

Get the ClassId.

Returns:

A ClassId that can be passed to IsKindOf.

See also:

Type Information and Casting

bool TMessage::IsKindOf (PFClassId *type*) `[virtual]`

Determine whether this message is derived from type.

Parameters:

type ClassId() of type to test.

See also:

Type Information and Casting

template<class TO> template< class TO > TO * TMessage::GetCast ()

Safely cast this message to another type.

Returns:

A cast pointer, or NULL.

See also:

Type Information and Casting

15.36.5 Member Data Documentation

str TMessage::mName

Name of this message.

For a button message, this is the name of the button that is sending the message.

15.37 TMessageListener Class Reference

`#include <pf/messagelistener.h>`

15.37.1 Detailed Description

A message listener–a class that you override and register with the TWindowManager if you want to listen for broadcast messages.

Public Member Functions

- virtual ~TMessageListener ()
  Destructor.
- virtual bool OnMessage (TMessage *message)=0
  This function will be called for each broadcast message that's delivered.

15.37.2 Member Function Documentation

virtual bool TMessageListener::OnMessage (TMessage * *message*) `[pure virtual]`

This function will be called for each broadcast message that's delivered.

Parameters:

message The message being delievered.

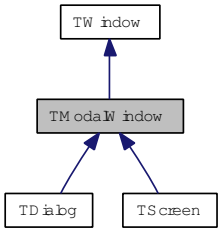
Returns:

True if the message was processed. No more searching for handlers will occur, and the message will be deleted by the caller if necessary.

15.38 TModalWindow Class Reference

`#include <pf/modalwindow.h>`

Inheritance diagram for TModalWindow:



15.38.1 Detailed Description

Base class for any window that can be a modal window.
Adds a number of functions to the default TWindow:

- Handles tasks that are associated with a modal window.
- Manages a TClock that automatically gets paused when the modal is covered by another modal.
- Does some internal bookeeping relevant to the window stack and opaque full-screen windows.
- Handles dispatching key messages to a default focus window.

Focus and Key Handling

- void SetDefaultFocus (TWindow *focus)

  Set the window that will receive an "implicit" focus when no other window has the focus.

- TWindow * GetDefaultFocus ()

  Get the current default focus.

- virtual void AddHotkey (str key, str buttonToPress)

  Add a hot key to the current modal window.

- virtual bool OnKeyDown (char key, uint32_t flags)

  Raw key hit on keyboard: TModalWindow will check to see if it's a "hot" key, and if so, then trigger the appropriate child button.

- static str NormalizeKeyString (str key)

  Take a string with a key description and normalize the description.

Public Member Functions

- TClock * GetClock ()
  Get a reference to the clock associated with this modal window.
- virtual void PostChildrenInit (TWindowStyle &style)
  Do post-children-added initialization when being created from Lua.

Construction/destruction

- TModalWindow ()
  Default constructor.
- virtual ~TModalWindow ()
  Destructor.

Modal Window Task Handling

- void AdoptTask (TTask *task)
  Add a task to the modal window's task list.
- bool OrphanTask (TTask *task)
  Remove a task from the task list.
- void DestroyTasks ()
  Destroy all tasks this window owns.

Message handlers

TModalWindow handles these messages for you.

- virtual void OnSetFocus (TWindow *previous)
  This window is receiving the keyboard focus.
- virtual bool OnNewParent ()
  Handle any initialization or setup that is required when this window is assigned to a new parent.
- virtual bool OnMessage (TMessage *message)
  Handle a message.
- virtual bool OnChar (char key)
  Translated character handler.
- void DoModalProcess (TTask::ETaskContext context=TTask::eNormal)
  Handles any modal processing that needs to happen.

15.38.2 Constructor & Destructor Documentation

TModalWindow::TModalWindow ()

Default constructor.

Sets window type to include kModal.

15.38.3 Member Function Documentation

void TModalWindow::AdoptTask (TTask * *task*)

Add a task to the modal window's task list.

Task list takes ownership of the task and will delete it when the task notifies that it is complete.

Parameters:

task Task to add.

bool TModalWindow::OrphanTask (TTask * *task*)

Remove a task from the task list.

Does not delete the task, but rather releases ownership; calling function now owns task.

Parameters:

task Task to remove.

Returns:

true if task was removed, false if task was not found

void TModalWindow::DestroyTasks ()

Destroy all tasks this window owns.

Used when the window is about to be destroyed.

Destruction is "safe"–tasks will be added to a destroy list if the list is being iterated.

void TModalWindow::SetDefaultFocus (TWindow * *focus*)

Set the window that will receive an "implicit" focus when no other window has the focus.

Parameters:

focus Default focus target. NULL to remove the default focus.

virtual void TModalWindow::AddHotkey (str *key*, str *buttonToPress*) `[virtual]`

Add a hot key to the current modal window.

When someone presses the given key, the given button name will be pressed.

To create more arbitrary commands that don't press visible buttons, create invisible buttons that can be "pressed" by name.

Parameters:

key Key to bind this button to. Can include:

- Ctrl+ for control, e.g., "Ctrl+C" would be control-c.
- Alt+ for the alt key, e.g., "Alt+C" would be alt-C.
- Shift+ for the shift key, e.g., "Shift+Alt+C" would be shift-alt-C.

If the key starts with '@', then ANY modifiers will be accepted, e.g., "@A" would work for an (uncaptured) "A", "Ctrl-A", "Alt-A", etc.

Parameters:

buttonToPress The name of the button to press. An empty string will delete the key association.

virtual bool TModalWindow::OnKeyDown (char *key*, uint32_t *flags*) `[virtual]`

Raw key hit on keyboard: TModalWindow will check to see if it's a "hot" key, and if so, then trigger the appropriate child button.

Parameters:

key Key pressed on keyboard.
flags TEvent::EKeyFlags mask representing the state of other keys on the keyboard when this key was hit.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented from TWindow.

static str TModalWindow::NormalizeKeyString (str *key*) `[static, protected]`

Take a string with a key description and normalize the description.

For example:

```
* alt-SHIFT-Ctrl-f
*
```

would return

```
* shift-ctrl-alt-F
*
```

The normalized string always sorts the operators in the order shift, ctrl, alt, and presents them in lower case. The key name itself is capitalized.

This function is used internally to normalize the key string in AddHotkey() and TModalWindow::OnKeyDown()– there's no need for you to call it yourself unless you're extending TModalWindow.

Parameters:

key Key to normalize.

Returns:

A normalized string.

virtual void TModalWindow::OnSetFocus (TWindow * *previous*) `[virtual]`

This window is receiving the keyboard focus.

Parameters:

previous The window that was previously focused. Can be NULL.

Reimplemented from TWindow.

virtual bool TModalWindow::OnNewParent () `[virtual]`

Handle any initialization or setup that is required when this window is assigned to a new parent.
No initialization of the window has happened prior to this call.

Returns:

True on success; false on failure.

See also:

Init
PostChildrenInit

Reimplemented from TWindow.
Reimplemented in TDialog.

virtual bool TModalWindow::OnMessage (TMessage * *message*) `[virtual]`

Handle a message.

Parameters:

message Payload of message.

Returns:

True if message handled; false otherwise.

Reimplemented from TWindow.
Reimplemented in TDialog.

virtual bool TModalWindow::OnChar (char *key*) `[virtual]`

Translated character handler.

Parameters:

key Key hit on keyboard, along with shift translations.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented from TWindow.

void TModalWindow::DoModalProcess (TTask::ETaskContext *context* = `TTask::eNormal`)

Handles any modal processing that needs to happen.

Processes tasks in the task list. This is called internally by the system once per frame in TTask::eNormal state and once in TTask::eOnDraw state.

Parameters:

context The context the processes should be called in.

TClock* TModalWindow::GetClock ()

Get a reference to the clock associated with this modal window.

This clock will be automatically paused and unpaused when other modal windows hide and reveal this modal window.

Returns:

A pointer to the clock.

virtual void TModalWindow::PostChildrenInit (TWindowStyle & *style*) `[virtual]`

Do post-children-added initialization when being created from Lua.

Any initialization that needs to happen after a window's children have been added can be placed in a derived version of this function.

Warning:

Remember to always call the base class if you're overriding this function.

Parameters:

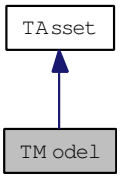
style Current style environment that this window was created in.

Reimplemented from TWindow.

15.39 TModel Class Reference

`#include <pf/model.h>`

Inheritance diagram for TModel:



15.39.1 Detailed Description

A 3d model.

Public Member Functions

- void Draw ()

  Draw the model using the current texture, material, and transformation matrix.

- bool Pick (uint32_t iWndX, uint32_t iWndY, TReal *pfPickDist, TVec3 *pvHit)

  Cast a ray at the model and determine whether it's been hit.

- str GetName ()

  Get the name of the asset as it was created.

- long GetPolyCount ()

  Get the number of polygons (triangles) in the model.

- uint32_t GetTriangleCount ()

  Get the number of triangles in the triangle array.

- const uint16_t * GetTriangles ()

  Get a pointer to an array of uint16_t values triples that indicate vertex indices that define triangles.

- const TVert * GetVertices ()

  Get an array of the model's vertices.

- uint32_t GetVertexCount ()

  Get the number of vertices in the model.

- TModelRef GetRef ()

  Get a reference to this asset. Do not call in a constructor!

Static Public Member Functions

- static TModelRef Get (str assetName)

  Factory.

Public Attributes

- TModelData * mData

  Implemenation details.

Protected Member Functions

- virtual void Restore ()

  Restore an asset.

- virtual void Release ()

  Release an asset.

15.39.2 Member Function Documentation

void TModel::Draw ()

Draw the model using the current texture, material, and transformation matrix.

Any texture used with TModel::Draw must be square and have dimensions that are powers of two.

bool TModel::Pick (uint32_t *iWndX*, uint32_t *iWndY*, TReal * *pfPickDist*, TVec3 * *pvHit*)

Cast a ray at the model and determine whether it's been hit.

Uses current transformation matrix to position model.

Parameters:

iWndX X in TScreen coordinates.
iWndY Y in TScreen coordinates.
pfPickDist Pointer to a float that starts out initialized to the maximum distance the cast ray should collide with polygons. On return contains the actual distance from the screen to the model.
pvHit The 3d point where the cast ray intersects the model.

Returns:

True if model hit; false otherwise.

str TModel::GetName ()

Get the name of the asset as it was created.

Returns:

The name of the asset.

long TModel::GetPolyCount ()

Get the number of polygons (triangles) in the model.

Deprecated

This function will be replaced by GetTriangleCount().

Returns:

Number of triangles.

uint32_t TModel::GetTriangleCount ()

Get the number of triangles in the triangle array.

Returns:

Number of triangles.

const uint16_t* TModel::GetTriangles ()

Get a pointer to an array of uint16_t values triples that indicate vertex indices that define triangles.

Returns:

A pointer to triangle indices.

const TVert* TModel::GetVertices ()

Get an array of the model's vertices.

Returns:

A pointer to an array of TVert vertices.

See also:

GetVertexCount

uint32_t TModel::GetVertexCount ()

Get the number of vertices in the model.

Returns:

A count of vertices in the vertex array.

See also:

GetVertices

static TModelRef TModel::Get (str *assetName*) `[static]`

Factory.

Parameters:

assetName Asset id of the model

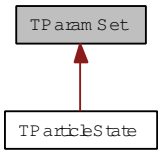
Returns:

A TModelRef of the model

15.40 TParamSet Class Reference

`#include <pf/luaparticlesystem.h>`

Inheritance diagram for TParamSet:



15.40.1 Detailed Description

A set of parameters or return values, depending on context.

Public Member Functions

- void ResetPartial (uint32_t base, uint8_t *s, uint32_t count)
  Reload part of a ParamSet with return parameters.
- void Reset (uint8_t *s, uint32_t count)
  Reload a ParamSet with return parameters.
- TParamSet (TReal *p, uint8_t *s, uint32_t count)
  Create a TParamSet.
- uint8_t GetCount ()
  Get the current parameter count.
- uint32_t GetOffset (uint8_t param)
  Get the offset of the nth entry in the set.
- uint8_t GetSize (uint8_t entry)
  Get the size of a particular entry.
- void Redirect (TReal *p)
  Point this at a new parameter set (typically a new particle).
- template<typename Type>
  Type & Param (uint8_t param)
  Parameter extractor.
- TReal & GetReal (uint32_t index)
  Get a raw real value.
- TReal GetReal (uint32_t index) const
  Get a raw real value.

Static Public Attributes

- static const uint32_t kMaxParams = 255

  The maximum number of parameters that can be passed.

15.40.2 Constructor & Destructor Documentation

TParamSet::TParamSet (TReal * *p*, uint8_t * *s*, uint32_t *count*)

Create a TParamSet.

Parameters:

p Set of parameter values (must be count*4 in TReals)
s Sizes of parameters (must be count int8_t values)
count Number of parameters.

References Reset().

15.40.3 Member Function Documentation

void TParamSet::ResetPartial (uint32_t *base*, uint8_t * *s*, uint32_t *count*)

Reload part of a ParamSet with return parameters.

Parameters:

base New first parameter in set
s A pointer to a local list of parameter sizes (number of floats in each parameter).
count Number of parameters. Must be less than or equal to original parameter count.

References ASSERT, and kMaxParams.

Referenced by TParticleMachineState::InitReturnValues(), TParticleMachineState::Push(), and Reset().

void TParamSet::Reset (uint8_t * *s*, uint32_t *count*)

Reload a ParamSet with return parameters.

Parameters:

s A pointer to a local list of parameter sizes (number of floats in each parameter).
count Number of parameters. Must be less than or equal to original parameter count.

References ResetPartial().

Referenced by TParamSet().

uint8_t TParamSet::GetCount ()

Get the current parameter count.

Returns:

Number of parameters in the set.

Reimplemented in TParticleState.

Referenced by TParticleState::GetCount(), and TParticleState::GetReal().

uint32_t TParamSet::GetOffset (uint8_t *param*)

Get the offset of the nth entry in the set.

Parameters:

param Parameter entry to query.

Returns:

Offset (in TReals) into the set.

Reimplemented in TParticleState.

Referenced by TParticleState::GetOffset(), TParticleMachineState::GetOffset(), TParticleMachineState::GetReal(), and TParticleMachineState::Push().

uint8_t TParamSet::GetSize (uint8_t *entry*)

Get the size of a particular entry.

Parameters:

entry Entry to query.

Returns:

Size (in TReals) of entry.

Referenced by TParticleMachineState::GetSize().

void TParamSet::Redirect (TReal * *p*)

Point this at a new parameter set (typically a new particle).

Parameters:

p

Reimplemented in TParticleState.

Referenced by TParticleState::Redirect().

template<typename Type> Type& TParamSet::Param (uint8_t *param*)

Parameter extractor.

Parameters:

Type Type of parameter to extract.
param Parameter base (which parameter number this is)

Returns:

A reference of type Type to the parameter that can be read or written to.

Reimplemented in TParticleState.

Referenced by TParticleMachineState::Param().

TReal& TParamSet::GetReal (uint32_t *index*)

Get a raw real value.

Parameters:

index Index of real to extract.

Returns:

A value in the set.

Reimplemented in TParticleState.

Referenced by TParticleState::GetReal(), TParticleMachineState::GetReal(), and TParticleMachineState::Push().

TReal TParamSet::GetReal (uint32_t *index*) const

Get a raw real value.

Parameters:

index Index of real to extract.

Returns:

A value in the set.

15.41 TParticleFunction Class Reference

`#include <pf/luaparticlesystem.h>`

15.41.1 Detailed Description

A user data source.

Public Member Functions

- virtual void InitFrame ()
  Optional initializer that's called once per particle render frame.
- virtual uint8_t GetReturnSize (int paramCount, uint8_t *sizes)=0
  Initialize parameters.
- virtual bool Process (TParticleState &particle, TParticleMachineState &params)=0
  Process function or fetch data.
- PFClassId ClassId ()
  The class id of this class.
- virtual bool IsKindOf (int type)
  Query whether this class IsKindOf another class.
- template<class TO>
  TO * GetCast ()
  Safely cast this class to another class.

Static Public Member Functions

- static TParticleFunction * CreateFromId (PFClassId id)
  Dynamic creation.

Public Attributes

- str mName
  The name of the function; will be set internally when it's registered.

15.41.2 Member Function Documentation

virtual uint8_t TParticleFunction::GetReturnSize (int *paramCount*, uint8_t * *sizes*) [pure virtual]

Initialize parameters.

For DataSource functions, incoming parameters will always be

Parameters:

paramCount Number of parameters in array.
sizes Pointer to size array (NULL for registered data sources).

Returns:

Size of parameter in TReals.

virtual bool TParticleFunction::Process (TParticleState & *particle*, TParticleMachineState & *params*) [pure virtual]

Process function or fetch data.

Parameters:

particle Particle being processed.
params [in/out] parameters

Returns:

True on success; false if parameters are incorrect.

static TParticleFunction* TParticleFunction::CreateFromId (PFClassId *id*) [static]

Dynamic creation.

Parameters:

id The PFClassId of the class to create.

Returns:

A new TParticleFunction-derived class instance.

virtual bool TParticleFunction::IsKindOf (int *type*) [virtual]

Query whether this class IsKindOf another class.

Parameters:

type The ClassId of the class to compare to.

Returns:

True if class is, or is derived from, target class.

template<class TO> TO* TParticleFunction::GetCast ()

Safely cast this class to another class.

Parameters:

TO Class to convert to.

Returns:

A cast pointer, or NULL if this class is unrelated to the target.

15.42 TParticleMachineState Class Reference

`#include <pf/luaparticlesystem.h>`

15.42.1 Detailed Description

The internal state of a TLuaParticleSystem.

Public Member Functions

- TParticleMachineState (TParamSet &paramSet, TLuaParticleSystem *lps)
  A set of function parameters.
- template<typename Type>
  Type & Param (int8_t param)
  Parameter extractor.
- TReal & GetReal (uint32_t index)
  Get a TReal value directly from the stack.
- uint8_t GetSize (int8_t param)
  Get the size of one of the parameters.
- void Push (TReal *r, uint8_t size)
  Push a value onto the stack.
- template<typename Type>
  void Push (Type value)
  Push a value onto the stack.
- void Leave ()
  Exit a function (removes any remaining parameters; a NOP if the function pushed any return values).
- uint8_t GetCount ()
  Get the local parameter count.
- uint32_t GetOffset (uint8_t param)
  Get the offset of a parameter on the stack.
- void InitReturnValues (uint8_t *s, uint32_t count)
  Reload with return parameters.
- void SetNumParams (uint8_t num)
  Set the number of parameters being passed to the current function.
- uint8_t GetNumParams ()
  Get the number of parameters passed.

- TParticleFunction * GetDataSource (int32_t source)

  Get a registered data source by index.

15.42.2 Constructor & Destructor Documentation

TParticleMachineState::TParticleMachineState (TParamSet & *paramSet*, TLuaParticleSystem * *lps*)

A set of function parameters.

Parameters:

paramSet The stack to initialize function parameters from.
lps A pointer to the associated TLuaParticleSystem.

15.42.3 Member Function Documentation

template<typename Type> Type& TParticleMachineState::Param (int8_t *param*)

Parameter extractor.

Parameters:

Type Type of parameter to extract.
param Parameter base (which parameter number this is)

Returns:

A Type reference that can be read or written to.

References ASSERT, and TParamSet::Param().

TReal& TParticleMachineState::GetReal (uint32_t *index*)

Get a TReal value directly from the stack.

Parameters:

index Index of value.

Returns:

The value from the stack.

References TParamSet::GetOffset(), and TParamSet::GetReal().

uint8_t TParticleMachineState::GetSize (int8_t *param*)

Get the size of one of the parameters.

Parameters:

param Parameter to query.

Returns:

Size of parameter.

References ASSERT, and TParamSet::GetSize().

void TParticleMachineState::Push (TReal * *r*, uint8_t *size*)

Push a value onto the stack.

Parameters:

r Pointer to the value array.
size Size of the value array.

References TParamSet::GetOffset(), TParamSet::GetReal(), and TParamSet::ResetPartial().

Referenced by Push().

template<typename Type> void TParticleMachineState::Push (Type *value*)

Push a value onto the stack.

Invalidates incoming parameter list.

Parameters:

Type
value

References Push().

uint8_t TParticleMachineState::GetCount ()

Get the local parameter count.

Returns:

Number of parameters in the set.

uint32_t TParticleMachineState::GetOffset (uint8_t *param*)

Get the offset of a parameter on the stack.

Parameters:

param The parameter number to retrieve. The stack is an array of TReal values, but they logically group into parameters, so if the first parameter is a Vec3(), the second parameter offset will be 3.

Returns:

The offset to the requested parameter.

References TParamSet::GetOffset().

void TParticleMachineState::InitReturnValues (uint8_t * *s*, uint32_t *count*)

Reload with return parameters.

Parameters:

s A pointer to a local list of parameter sizes (number of floats in each parameter).

count Number of return values.

References TParamSet::ResetPartial().

void TParticleMachineState::SetNumParams (uint8_t *num*)

Set the number of parameters being passed to the current function.

Parameters:

num Number of parameters

uint8_t TParticleMachineState::GetNumParams ()

Get the number of parameters passed.

Returns:

The number of incoming parameters.

TParticleFunction* TParticleMachineState::GetDataSource (int32_t *source*)

Get a registered data source by index.

Parameters:

source Data source index to read (must be negative).

Returns:

The data source.

15.43 ParticleMember Struct Reference

`#include <pf/luaparticlesystem.h>`

15.43.1 Detailed Description

A particle member value.

When additional values are added to a particle (beyond what exists in the base particle values), each is allocated as a ParticleMember.

Public Member Functions

- ParticleMember (int16_t base=0, uint16_t size=0)
  Constructor.

Public Attributes

- int16_t mBase
  Index in TReals into the particle.
- uint16_t mSize
  Size of this member.

15.43.2 Constructor & Destructor Documentation

ParticleMember::ParticleMember (int16_t *base* = 0, uint16_t *size* = 0)

Constructor.

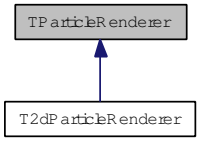
Parameters:

base Base index (in TReals) of this member.
size Size of this member in TReals.

15.44 TParticleRenderer Class Reference

`#include <pf/particlerenderer.h>`

Inheritance diagram for TParticleRenderer:



15.44.1 Detailed Description

The abstract particle renderer class: This class is used by TLuaParticleSystem to wrap an actual particle renderer. This way you can use TLuaParticleSystem to drive the default 2d particle system or other more complex systems.

Public Member Functions

- virtual void Draw (const TVec3 &at, TReal alpha, const ParticleList &particles, int maxParticles)=0
  Render the particles.
- virtual void SetTexture (TTextureRef texture)=0
  Set the texture for the particle.
- virtual void SetRendererOption (str option, const TReal(&value)[4])=0
  Set a renderer-specific option.
- virtual TReal * GetPrototypeParticle ()=0
  Get an initialized particle that will be copied over each particle after creation but before running initializers.
- virtual uint32_t GetPrototypeParticleSize ()=0
  Size of the array of TReals returned by GetPrototypeParticle.
- virtual str GetLuaInitString ()=0
  Get a TLuaParticleSystem initialization string.

15.44.2 Member Function Documentation

virtual void TParticleRenderer::Draw (const TVec3 & *at*, TReal *alpha*, const ParticleList & *particles*, int *maxParticles*) `[pure virtual]`

Render the particles.

Parameters:

at Location to render particles.
alpha Alpha to render particles with.
particles The list of particles to render.
maxParticles The maximum number of particles this particle system is expecting to render. MUST be greater than the number of particles or Bad Things will happen.

Implemented in T2dParticleRenderer.

virtual void TParticleRenderer::SetTexture (TTextureRef *texture*) [pure virtual]

Set the texture for the particle.

Parameters:

texture Texture to use.

Implemented in T2dParticleRenderer.

virtual void TParticleRenderer::SetRendererOption (str *option*, const TReal(&) *value*[4]) [pure virtual]

Set a renderer-specific option.

Parameters:

option Option to set.
value Value to set option to, in the form of an array of TReals. Not all values in array are relevant for all options.

Implemented in T2dParticleRenderer.

virtual TReal* TParticleRenderer::GetPrototypeParticle () [pure virtual]

Get an initialized particle that will be copied over each particle after creation but before running initializers.

Returns:

A pointer to an array of TReals.

Implemented in T2dParticleRenderer.

virtual uint32_t TParticleRenderer::GetPrototypeParticleSize () [pure virtual]

Size of the array of TReals returned by GetPrototypeParticle.

Returns:

Number of reals.

Implemented in T2dParticleRenderer.

virtual str TParticleRenderer::GetLuaInitString () [pure virtual]

Get a TLuaParticleSystem initialization string.

Returns:

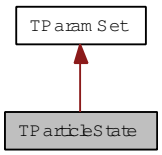
A valid Lua string that will be executed by a TLuaParticleSystem that uses this renderer.

Implemented in T2dParticleRenderer.

15.45 TParticleState Class Reference

`#include <pf/luaparticlesystem.h>`

Inheritance diagram for TParticleState:



15.45.1 Detailed Description

A particle state.

A particle state is made up of some number of floating point values and an indication of elapsed milliseconds.

Note the private inheritance: We don't want to expose TParamSet::Reset on a TParticleState, but we want the TParamSet implementation.

Public Member Functions

- TParticleState (TReal *p, uint8_t *s, uint32_t count, uint32_t ms)
  Constructor.
- template<typename Type>
  Type & Param (uint8_t param)
  Parameter extractor.
- TReal & GetReal (uint32_t index)
  Get a raw real value.
- uint32_t GetOffset (uint8_t param)
  Get the offset of the nth entry in the set.
- void Redirect (TReal *p)
  Point this at a new parameter set (typically a new particle).
- uint32_t GetCount ()
  Get the current parameter count.
- uint32_t GetMS ()
  Get the number of milliseconds being processed.
- bool GetAlive ()
  Query whether this particle is still alive.
- void KillParticle ()

Kill this particle (mark for destruction).

15.45.2 Constructor & Destructor Documentation

TParticleState::TParticleState (TReal * *p*, uint8_t * *s*, uint32_t *count*, uint32_t *ms*)

Constructor.

Parameters:

p A pointer to an array of TReals that will be used as the dynamic particle state. Must be count*4 TReals.
s A pointer to an array of bytes that indicate parameter sizes.
count The number of members in the particle.
ms The number of milliseconds that have passed in this time step.

15.45.3 Member Function Documentation

template<typename Type> Type& TParticleState::Param (uint8_t *param*)

Parameter extractor.

Parameters:

Type Type of parameter to extract.
param Parameter base (which parameter number this is)

Returns:

A Type reference that can be read or written to.

Reimplemented from TParamSet.

TReal& TParticleState::GetReal (uint32_t *index*)

Get a raw real value.

Parameters:

index Index of real to extract.

Returns:

A value in the set.

Reimplemented from TParamSet.

References ASSERT, TParamSet::GetCount(), GetOffset(), and TParamSet::GetReal().

uint32_t TParticleState::GetOffset (uint8_t *param*)

Get the offset of the nth entry in the set.

Parameters:

param Parameter entry to query.

Returns:

Offset (in TReals) into the set.

Reimplemented from TParamSet.

References TParamSet::GetOffset().

Referenced by GetReal().

void TParticleState::Redirect (TReal * *p*)

Point this at a new parameter set (typically a new particle).

Parameters:

p

Reimplemented from TParamSet.

References TParamSet::Redirect().

uint32_t TParticleState::GetCount ()

Get the current parameter count.

Returns:

Number of parameters in the set.

Reimplemented from TParamSet.

References TParamSet::GetCount().

uint32_t TParticleState::GetMS ()

Get the number of milliseconds being processed.

Returns:

Time elapsed in milliseconds.

bool TParticleState::GetAlive ()

Query whether this particle is still alive.

Returns:

True if alive.

15.46 TPfHiscores Class Reference

`#include <pf/pfhiscores.h>`

15.46.1 Detailed Description

TPfHiScores - class that manages local and global hiscore saving and viewing.

This is the backend class for managing hiscores. It has the ability to save and load local hiscores as well as submit and retrieve hiscores from the server.

For information about how to test your hiscore implementation against a debug server, please see the PlayFirst Global High Scores information.

Localization - the hiscore system can be localized by including a file called "hiscore.xml" in the same data directory as the .dll or the .exe if you are using a static lib. The contents of this xml file should be:

XML

```
<hiscore>
   <language>en</language>
   <defaulterror>Unable to connect to server. Please try again later.</defaulterror>
</hiscore>
```

The language parameter will override any setting used with eLanguage property in SetProperty(). Language should be ISO-639 (e.g. "en", "jp", "fr", "en_CA"). The default error string is what is displayed in the case of not being able to connect to the server.

Public Types

- enum EStatus { **eSuccess** = 0, **ePending**, **eError** }

  After sending a request to the hiscore server, this is the status of the request.

- enum EProperty { **eLanguage** = 0, **ePlayerName**, **eGameMode** }

  The various properties that can be set for the hiscore module.

- enum EMedalType { eMedalType_Game = 0, eMedalType_Mode }

  The various medal types for use in KeepMedal().

- enum EUserScore { **eLocalEligible** = 0, **eGlobalBest**, **eLocalBest** }

  The various modes for calling GetUserBestScore().

- enum ESubmitMode { kSubmitScore = 0x00000001, kSubmitMedal = 0x00000002, kSubmitAll = 0xFFFFFFFF }

  Submit modes.

Public Member Functions

- TPfHiscores (bool saveData=true)

  Default constructor.

- ~TPfHiscores ()

Default destructor.

- void SetProperty (EProperty property, str value)
  Sets a given property.
- void KeepScore (int32_t score, bool replaceExisting, const char *gameData)
  Log a score for the current player.
- void KeepMedal (const char *medalName, EMedalType type, str gameData)
  Log a medal for the current player.
- bool GetRememberedUserInfo (str *userName, str *password)
  Retreives a username and password if one has been saved.
- void SetRememberedUserInfo (str userName, str password)
  Saves a user name and password.
- void RequestCategoryInformation ()
  Submit a request to the server to retreive category information.
- int32_t GetCategoryCount ()
  Return the number of categories available for hiscores.
- bool GetCategoryName (int32_t n, char *name, uint32_t bufSize)
  Fill in a table name.
- void RequestScores (int32_t categoryIndex)
  Submit a request to the server to retreive hiscores.
- EStatus GetServerRequestStatus (char *msg, uint32_t bufLen, bool *pQualified)
  Return the status of the last server request.
- int32_t GetScoreCount (bool local)
  Retrieve the number of scores currently downloaded from the server.
- bool GetScore (bool local, int32_t n, int32_t *pRank, char *name, uint32_t bufSize, bool *pAnonymous, int32_t *pScore, char *gameData, uint32_t gameDataBufferSize)
  Fill in all the various score information for a given score.
- bool GetUserBestScore (EUserScore userScore, int32_t *pScore, int32_t *pRank, char *gameData, uint32_t gameDataBufferSize)
  Fill in all the various score information for a given user.
- int GetNumMedalsToSubmit ()
  Returns the number of medals that a user has earned but has not submitted to the server with SubmitData().
- int GetNumMedalsEarned ()
  Returns the number of medals that a user has earned, regardless of whether or not they have been submitted.
- bool GetEarnedMedalInfo (int index, str *medalName, EMedalType *type, str *gameMode, str *gameData)

Returns stored medal info for the current user.

- bool SubmitData (str username, str password, bool bRemember, ESubmitMode submitMode)

  Submit the current user's best score and medals to the server.

- void ClearScores ()

  Clear the local scores for the current game mode.

- void ClearPlayerData (str playername)

  Clear the local scores and medal for a specified player name.

- uint32_t EncryptData (const void *toEncrypt, uint32_t len, char *buf, uint32_t bufLen)

  Encrypt a byte stream.

- uint32_t DecryptData (const char *toDecrypt, void *buf, uint32_t bufLen)

  Decrypt a string.

- void LogScore (int32_t score, bool replaceExisting, const char *gameData, const char *serverData=NULL)
- bool SubmitScore (str username, str password, bool bRemember)
- bool SubmitMedals (const char *medalsData, str username, str password, bool bRemember)

15.46.2 Member Enumeration Documentation

enum TPfHiscores::EMedalType

The various medal types for use in KeepMedal().

Enumerator:

eMedalType_Game Medals that are awarded for the entire game, regardless of what mode the user is in.
eMedalType_Mode Medals that are awarded per game mode (the user can earn this medal once in each game mode).

enum TPfHiscores::ESubmitMode

Submit modes.

Controls what type of data is submitted

Enumerator:

kSubmitScore Submit the user's best score to the server.
kSubmitMedal Submit the user's medals to the server.
kSubmitAll Submit all available data to the server.
It is recommended that this mode be used whenever the UI permits - that way all of the user's data is submitted with one call.

15.46.3 Constructor & Destructor Documentation

TPfHiscores::TPfHiscores (bool *saveData* = `true`)

Default constructor.

This initalizes the hiscores system. Among other things, it loads in a preference file off disk which contains all the stored local scores.

The game name and encryption key must be set in the global configuration prior to constructing. See TPlatform::SetConfig() for details. For standalone builds, see the standalone hiscore documentation.

Parameters:

saveData Whether or not data should be saved between sessions, default is true. For example, in a web game you might not want scores to persist between sessions, in which case saveData should be false.

15.46.4 Member Function Documentation

void TPfHiscores::SetProperty (EProperty *property*, str *value*)

Sets a given property.

Parameters:

property Which property to set. eLanguage - which language to set (ISO-639 (e.g. "en", "jp", "fr", "en_CA")). The default language is "en". Note that this property will have no effect if you are using the "hiscore.xml" localization file described above. ePlayerName - This is the local player name. This will be used for logging and returning local scores. eGameMode - Once the game mode is set, it will be used for all score submission and retreivals.

value Value to set the property to (for example, if property is eLanguage, value might be "en").

void TPfHiscores::KeepScore (int32_t *score*, bool *replaceExisting*, const char * *gameData*)

Log a score for the current player.

This stores a score for the current player into the local score table. A score must be logged before it can be submitted. The score is logged for the player name set with SetPlayerName()

Parameters:

score the score the user is submitting

replaceExisting should this score submission replace an existing user score, or create a new one? Typical usage is that story/career mode games replace an existing score, whereas arcade modes allow a new score with each submission.

gameData game specific scoring information (up to 60 chars supported by server) - it is important that this be data that is additional scoring information (i.e. last level reached) and not scoring interpretation information (i.e. awarded the "Best Ever" trophy) so that the server can adjust awards later on. Also, this game data should not include any text that would need to be localized. Therefore, it should just be numbers if at all possible.

void TPfHiscores::KeepMedal (const char * *medalName*, EMedalType *type*, str *gameData*)

Log a medal for the current player.

This stores a medal for the current player. A medal must be logged before it can be submitted. The medal is logged for the player name set with SetPlayerName()

If a player has already earned this particular medal, then this call does nothing. It is possible to tell if this call changes the number of medals the player has by calling GetNumMedalsToSubmit() before and after this call.

Parameters:

medalName The name of the medal the user earned. The same names used to fill in the TPlatform::kPFGameMedalName config settings should be used here.
type The type of medal earned
gameData This string can be used to keep game specific information about how this medal was earned. This data will not be transmitted to the server, and is just kept for retrieval with GetEarnedMedalInfo(). An example use might be a time stamp so that when GetEarnedMedalInfo() is called to display a UI, a date can be displayed along with the medal.

bool TPfHiscores::GetRememberedUserInfo (str * *userName*, str * *password*)

Retreives a username and password if one has been saved.

If the user submits their score with the bRemember flag set to true, then the username and password are saved, so that next time the user does not have to type them again.

Parameters:

userName str to hold user name
password str to hold password

Returns:

- if the user/pass have been saved, this returns true and fills in the user/pass. if it has not been saved, it returns false and does nothing to username and password.
If the user/pass have been saved, this returns tru and fills in the user/pass. If it has not been saved, it returns false and does nothing to the username and password.

void TPfHiscores::SetRememberedUserInfo (str *userName*, str *password*)

Saves a user name and password.

Saves a user name and password without logging any score information.

Parameters:

userName If NULL or "" will delete any previously saved information.
password If NULL or "" will delete any previously saved information.

void TPfHiscores::RequestCategoryInformation ()

Submit a request to the server to retreive category information.

After calling this function, GetServerRequestStatus() must be polled until a result is ready.

int32_t TPfHiscores::GetCategoryCount ()

Return the number of categories available for hiscores.

You must have retrieved the category information with RequestCategoryInformation() first.

Returns:

Number of categories available.

bool TPfHiscores::GetCategoryName (int32_t *n*, char * *name*, uint32_t *bufSize*)

Fill in a table name.

You must have retrieved the category information with RequestCategoryInformation() first.

Parameters:

n Which category to fetch.
→ ***name*** Fills in category's name
bufSize Size of name buffer.

Returns:

Returns false if category information has not been properly initialized.

void TPfHiscores::RequestScores (int32_t *categoryIndex*)

Submit a request to the server to retreive hiscores.

After calling this function, GetServerRequestStatus() must be polled until a result is ready.

Parameters:

categoryIndex Which category is requested (use GetCategoryCount() to see how many categories are available)

EStatus TPfHiscores::GetServerRequestStatus (char * *msg*, uint32_t *bufLen*, bool * *pQualified*)

Return the status of the last server request.

After calling any functiont that contacts the server, this function must be polled until a result is ready.

Parameters:

msg If EStatus is eError, an error message will be placed inside msg
bufLen Buffer length of msg.
pQualified For SubmitData(), if EStatus is eSuccess, this fills in whether or not the score qulaified for a global record.

Returns:

The current status of the request. If it is eError, the request failed and the message inside msg should be displayed to the user.

int32_t TPfHiscores::GetScoreCount (bool *local*)

Retrieve the number of scores currently downloaded from the server.

To retrieve a server score, you must have retrieved the score information with RequestScores() first.

Parameters:

local True to retrieve for local scores.

Returns:

Number of scores.

bool TPfHiscores::GetScore (bool *local*, int32_t *n*, int32_t * *pRank*, char * *name*, uint32_t *bufSize*, bool * *pAnonymous*, int32_t * *pScore*, char * *gameData*, uint32_t *gameDataBufferSize*)

Fill in all the various score information for a given score.

To retrieve a server score, you must have retrieved the score information with RequestScores() first.

Parameters:

local True for local high scores, false for global.
n Index into the score table.
→ ***pRank*** Fills in score's rank in current table.
→ ***name*** Fills in player's name.
bufSize Size of name buffer.
→ ***pAnonymous*** Fills in whether score is anonymous or not.
→ ***pScore*** Fills in score
→ ***gameData*** Fills in game specifc data
gameDataBufferSize Size of gameData buffer that can be filled in.

Returns:

Returns false if scores are not properly intialized.

bool TPfHiscores::GetUserBestScore (EUserScore *userScore*, int32_t * *pScore*, int32_t * *pRank*, char * *gameData*, uint32_t *gameDataBufferSize*)

Fill in all the various score information for a given user.

Depending on the userScore type passed in, different score information will be filled in.

Parameters:

userScore Type of score to fetch: eLocalEligible - the user's best local score eligible for submission (one that has not already been submitted). eGlobalBest - the user's best score on the current global score table (obtained from RequestScores()).
→ ***pScore*** Fills in the user's score.
→ ***pRank*** Fills in score's rank in current table.
→ ***gameData*** Fills in game specifc data.
gameDataBufferSize Size of gameData buffer that can be filled in.

Returns:

Returns false if the user has no eligible score, or if the game cannot find a user score in the current table.

int TPfHiscores::GetNumMedalsToSubmit ()

Returns the number of medals that a user has earned but has not submitted to the server with SubmitData().

This call does not return the total number of medals a user has earned, only the number of medals that have not yet been submitted.

This result of this call is not dependent on the current game mode - it returns a total number of medals across all game modes.

int TPfHiscores::GetNumMedalsEarned ()

Returns the number of medals that a user has earned, regardless of whether or not they have been submitted.

This only reflects the number of medals that the user has earned according to locally saved data. It does not fetch anything from the server.

This result of this call is not dependent on the current game mode - it returns a total number of medals across all game modes.

The result from this call can be used to index into GetEarnedMedalInfo().

bool TPfHiscores::GetEarnedMedalInfo (int *index*, str * *medalName*, EMedalType * *type*, str * *gameMode*, str * *gameData*)

Returns stored medal info for the current user.

Calling GetNumMedalsEarned() will let you know how many medals can be retrieved.

Parameters:

index Index of medal to be retrieved.
→ *medalName* Fills in the name of the medal (if medalName is not NULL).
→ *type* Fills in the type of the medal (if type is not NULL)
→ *gameMode* Fills in the gameMode that the medal was earned in (if gameMode is not NULL).
→ *gameData* Fills in the gameData associated with the medal (if gameData is not NULL).

Returns:

Returns false if the passed in index is out of range. Otherwise returns true

bool TPfHiscores::SubmitData (str *username*, str *password*, bool *bRemember*, ESubmitMode *submitMode*)

Submit the current user's best score and medals to the server.

After calling this, you should poll GetServerRequestStatus() to check for errors or successful submission.

Only one call to SubmitData() can be made at a time. You can not issue a second call until GetServerRequestStatus() has returned a result. Issuing a second call before GetServerRequestStatus() returns a result may result in undefined behavior.

Medals will only be submitted if a password is used. If a password is not used, then medals will be saved until the next time a password-enabled submission is made.

All of a user's available medals are submitted - the types of medals are not filtered by the current game mode.

Parameters:

username Name to submit the score under.
password The user's playfirst password. If this is NULL or "", then an anonymous submission is issued.
bRemember If this is true, then the module saves the username and password for future use. If it is false, it deletes any previously saved username and password.
submitMode What data to submit. When possible kSubmitAll should be used so that all user data is submitted. However, in certain UI presentations, it may only be appropriate to submit medals but not scores, or vice versa.

Returns:

If score submission is not possible, this returns false (i.e. the player has already submitted their best score, or

if they do not have any scores or medals for this game mode, etc.).

void TPfHiscores::ClearScores ()

Clear the local scores for the current game mode.

This will delete the current local scores for the current game mode. It cannot be undone.

void TPfHiscores::ClearPlayerData (str *playername*)

Clear the local scores and medal for a specified player name.

This will remove any scores associated with the player from the local score table, and will also remove their local medals. This operation cannot be undone.

Parameters:

playername - name to erase.

uint32_t TPfHiscores::EncryptData (const void * *toEncrypt*, uint32_t *len*, char * *buf*, uint32_t *bufLen*)

Encrypt a byte stream.

This will encrypt the passed in byte stream, and returns a string encoded in base64 (so it can be passed around like a string).

Parameters:

toEncrypt The bytestream that is to be encrypted.
len The length of data to encrypt - note that if you are encrypting a text string you will want to encrypt the null terminator too, so you should pass in strlen(toEncrypt) + 1.
→ ***buf*** The buffer to fill in with the encrypted string.
bufLen The size of the buffer. If this is too small, the function will return the size needed to encrypt the string, and will not fill in buf at all.

Returns:

0 on success, or else returns the length of the buffer needed to encrypt this string.

uint32_t TPfHiscores::DecryptData (const char * *toDecrypt*, void * *buf*, uint32_t *bufLen*)

Decrypt a string.

This will decrypt a null terminated Base64 string into a byte stream.

Parameters:

toDecrypt A null terminated Base64 string to decrypt.
→ ***buf*** The buffer to fill in with the decrypted string.
bufLen The size of the buffer. If this is too small, the function will return the size needed to decrypt the string, and will not fill in buf at all.

Returns:

0 on success, or else returns the length of the buffer needed to encrypt this string.

void TPfHiscores::LogScore (int32_t *score*, bool *replaceExisting*, const char * *gameData*, const char * *serverData* = NULL)

Deprecated

This function has been replaced by KeepScore(). Please use that instead.

Log a score for the current player.

This stores a score fore the current player into the local score table. A score must be logged before it can be submitted. The score is logged for the player name set with SetPlayerName()

Parameters:

score the score the user is submitting
replaceExisting should this score submission replace an existing user score, or create a new one? Typical usage is that story/career mode games replace an existing score, whereas arcade modes allow a new score with each submission.
gameData game specific scoring information (up to 60 chars supported by server) - it is important that this be data that is additional scoring information (i.e. last level reached) and not scoring interpretation information (i.e. awarded the "Best Ever" trophy) so that the server can adjust awards later on. Also, this game data should not include any text that would need to be localized. Therefore, it should just be numbers if at all possible.
serverData an optional XML formatted string that will enable certain hiscore-related features on the hiscore server. The XML must be well-formed.

Currently available features are:

Medals: Medals are awards given to users that complete certain tasks in the game. A medal has two parameters. The "name" parameter is the identification name of the medal. The "per" parameter can either be "type" or "game" - "type" means that this medal is specific to the current game mode, whereas "game" means that this medal is a global medal awarded across all game modes.

Examples: To submit 2 medals for this score:

XML

```
<medal name="medal1" per="game"/>
<medal name="medal2" per="type"/>
```

bool TPfHiscores::SubmitScore (str *username*, str *password*, bool *bRemember*)

Deprecated

This function has been replaced by SubmitData(). Please use that instead.

Submit a new score to the server.

After calling this, you should poll GetServerRequestStatus() to check for errors or successful submission.

Parameters:

username Name to submit the score under.
password The user's playfirst password. If this is NULL or "", then an anonymous submission is issued.
bRemember If this is true, then the module saves the username and password for future use. If it is false, it deletes any previously saved username and password.

Returns:

If score submission is not possible, this returns false (i.e. the player has already submitted their best score, or if they do not have any scores for this game mode, etc.).

bool TPfHiscores::SubmitMedals (const char * *medalsData*, str *username*, str *password*, bool *bRemember*)

Deprecated

This function has been replaced by SubmitData(). Please use that instead.

Submit medal information for the current player to the server.

After calling this, you should poll GetServerRequestStatus() to check for errors or successful submission

This stores the medals information for the current player into the local score table. The medals information must be logged before it can be submitted. The medals are logged for the player name set with SetPlayerName()

Parameters:

medalsData A well-formed XML string that specifies medal information to be stored on the server.
username Name to submit the score under.
password The user's playfirst password.
bRemember If this is true, then the module saves the username and password for future use. if it is false, it deletes any previously saved username and password.

Returns:

If the medals submission is not possible, this returns false.

Medals are awards given to users that complete certain tasks in the game. A medal has two parameters. The "name" parameter is the identification name of the medal. The "per" parameter can either be "type" or "game" - "type" means that this medal is specific to the current game mode, whereas "game" means that this medal is a global medal awarded across all game modes.

Examples:

XML

```
<medal name="dinerdash7_frag50" per="type" type="dinerdash7_counterstrike"/>
<medal name="dinerdash7_frag50" per="type" type="dinerdash7_fragfest"/>
<medal name="dinerdash7_frag100" per="type" type="dinerdash7_fragfest"/>
<medal name="dinerdash7_killedRonaldMcDonald" per="game"/>
<medal name="dinerdash7_killedBurgerKing" per="game"/>
```

15.47 TPlatform Class Reference

`#include <pf/platform.h>`

15.47.1 Detailed Description

The platform-specific functionality encapsulation class.

This class is created within the library and exists in one global instance that can be acquired anywhere in the application using the static function TPlatform::GetInstance().

Display Related Functions

- enum ECursorMode { **kCursorModeAbsolute**, **kCursorModeDelta** }
  These mouse button constants are here to allow you to respond to the use of other mouse buttons.
- void SetDisplay (uint32_t width, uint32_t height, bool fullscreen)
  Initialize the current display mode.
- void GetDisplay (uint32_t *pWidth, uint32_t *pHeight, bool *pbFullscreen)
  Get the current display parameters.
- void SetCursor (TTextureRef texture, TPoint hotSpot, bool hardware=false)
  Set a mouse cursor to an image.
- void ShowCursor (bool show)
  Show or hide the cursor.
- void SetCursorPos (const TPoint &at)
  Set the cursor position.
- void SetCursorMode (ECursorMode mode)
  Set the cursor mode.
- ECursorMode GetCursorMode () const
  Get the cursor mode.
- bool IsForeground ()
  Return true if the application is currently the foreground window.
- void SetForeground ()
  Set the game window to the foreground.
- bool SetFullscreen (bool bFullscreen)
  Convenience function for toggling fullscreen.
- bool IsFullscreen ()

True if the window is full screen.

- void AdoptTextureRefreshListener (TTask *rn)
  Add a Texture Refresh Listener: On some platforms (DirectX) there are situations that cause all textures to be destroyed.

- bool OrphanTextureRefreshListener (TTask *rn)
  Remove a Texture Refresh Listener.

Platform Environment and Information

- static TPlatform * GetInstance ()
  Get the singleton TPlatform.

- static str GetConfig (str setting, str defaultSetting="")
  Query for a configuration setting.

- static bool IsEnabled (str setting)
  Query as to whether a setting is enabled.

- static void SetConfig (str setting, str value)
  Set a client configuration value.

- class TSoundManager * GetSoundManager ()
  Get the sound manager.

- class TWindowManager * GetWindowManager ()
  Get the application window manager.

- TTaskList * GetTaskList ()
  Get the application task list.

- class TStringTable * GetStringTable ()
  Get the string table.

Public Types

- enum ExtendedMouseEvents {
  kMouseRightUp, kMouseRightDown, kMouseMiddleUp, kMouseMiddleDown,
  kMouseScrollLeft, kMouseScrollRight, kMouseScrollUp, kMouseScrollDown }
  These mouse button constants are here to allow you to respond to the use of other mouse buttons.

Public Member Functions

System Commands

Commands that interact with the operating system.

- void SetWindowTitle (str title)

Set the window application title.

- void OpenBrowser (const char *url)
  Open a URL in a Web browser on the target system.
- void GetEvent (TEvent *pEvent)
  Get an event from the application event queue.
- bool StringToClipboard (str copyString)
  Send a string to the system clipboard.
- str StringFromClipboard ()
  Retrieve a string, if any, from the current clipboard.
- void Exit (int32_t exitValue=0)
  Exit the program.

Timer and user event functions

Functions that can be used to set up and cancel timers, query elapsed time, and pause the application.

- uint32_t Timer ()
  A count in milliseconds since the program has initialized,.
- void Sleep (uint32_t ms)
  Sleep the program for a number of milliseconds.
- bool OrphanTask (TTask *task)
  Release a task from the global task list.
- void AdoptTask (TTask *task)
  Add a task to the global task list.

Randomness

*/** Return a random integer.*

Clients are encouraged to use this as opposed to the stdlib version for maximum compatibility, and this random number generator is randomly seeded at application startup.

Returns:

A random integer from 0 to 0xFFFFFFFF.

- uint32_t **Rand** ()

Static Public Attributes

Configuration values

Use these values in GetConfig() to get the related setting, or with SetConfig() to change the related setting (for writable values).

See also:

TPlatform::GetConfig
TPlatform::SetConfig

- static const char * kComputerId

Unique Computer Identifier.

- static const char * kCheatMode
  Cheat mode.
- static const char * kDownloadURL
  In a Web game, this is the URL that the browser should be set to when you click the "Download" button in the game.
- static const char * kInstallKey
  A key unique to the install of this application.
- static const char * kEncryptionKey
  The application's encryption key.
- static const char * kHiscoreLocalOnly
  Query as to whether hiscore mode is local-only.
- static const char * kHiscoreAnonymous
  Query as to whether hiscore mode is anonymous only.
- static const char * kBuildTag
  Query how this particular build has been tagged.
- static const char * kFirstPeek
  Query whether this build is a "first peek" build: A limited-functionality public beta version.
- static const char * kGameName
  What is the name of this game? Defined to be the string "gamename".
- static const char * kPublisherName
  What is the name of the publisher of this game? Defaults to "PlayFirst".
- static const char * kTextGraphicSpriteRender
  Call TPlatform::SetConfig(TPlatform::kTextGraphicSpriteRender, 1) to default all new TTextGraphic instances to "sprite rendering" mode.
- static const char * kGameVersion
  What version/build is this EXE? Defined to be the string "version".
- static const char * kPFGameHandle
  This value is the game handle that the game uses to communicate with any PlayFirst services, such as the hiscore system.
- static const char * kPFGameModeName
  This value is the prefix to the game mode names used to communicate with the PlayFirst Hiscore system.
- static const char * kPFGameMedalName
  This value is the prefix to the medal names used to communicate with the PlayFirst Hiscore system.
- static const char * kVsyncWindowedMode
  Instruct Playground to wait for the vertical blanking period to start prior to drawing to the screen in windowed mode.
- static const char * kUTF8Mode
  Enable UTF-8 support mode.
- static const char * kAutoMergeMask

Automatically merge any .mask.png files when loading a .jpg file.

- static const char * kOverlayWindowMouseEvents
  Enable mouse event routing to overlay windows.
- static const char * kVerboseDebug
  Enable verbose debug info from Playground.
- static const char * kMonitorFrequency
  Query the current monitor frequency.
- static const char * kFlashMajorVersion
  Raise the minimum major Flash version that your game requires.
- static const char * kSpriteAlwaysTestChildren
  Always do HitTest() on sprite children, even if the parent TSprite doesn't have a texture.
- static const char * kAdjustAnimatedParticles
  When using animated textures for particle systems, automatically adjust the position of each particle so that the animation frames play in the correct positions.
- static const char * kFixButtonMaskScaling
  Fix button mask scaling to properly scale a mask to the size of the button.
- static const char * kFPUPrecision
  Enable this in PlaygroundInit() to enable high-precision floating point.
- static const char * kKeepOGGUncompressed
  Enabling this will decompress any non-streamed .ogg sounds into memory, rather than keeping them compressed.
- static const char * kSoftwareTextureCulling
  If this is enabled, Playground will software cull any TTexture draws that occur outside of TRenderer::GetClippingRectangle.
- static const char * kLargeVRAMUsage
  Certain video cards do not handle VRAM paging well.

15.47.2 Member Enumeration Documentation

enum TPlatform::ExtendedMouseEvents

These mouse button constants are here to allow you to respond to the use of other mouse buttons.

PlayFirst game design constraints forbid the use of any other than the left mouse button as a "necessary" part of the user interface. In other words, any other buttons on the mouse or mouse wheel needs to be a supplemental interface for convenience of power users.

Enumerator:

kMouseRightUp Mouse right-button-up event.
kMouseRightDown Mouse right-button-down event.
kMouseMiddleUp Mouse middle-button-up event.
kMouseMiddleDown Mouse middle-button-down event.

kMouseScrollLeft Mouse scroll-left-button event.
kMouseScrollRight Mouse scroll-right-button event.
kMouseScrollUp Mouse scroll-up-button event.
kMouseScrollDown Mouse scroll-down-button event.

enum TPlatform::ECursorMode

These mouse button constants are here to allow you to respond to the use of other mouse buttons.

PlayFirst game design constraints forbid the use of any other than the left mouse button as a "necessary" part of the user interface. In other words, any other buttons on the mouse or mouse wheel needs to be a supplemental interface for convenience of power users.

15.47.3 Member Function Documentation

static TPlatform* TPlatform::GetInstance () `[static]`

Get the singleton TPlatform.

Returns:

The one-and-only TPlatform.

During normal game operation, it can be assumed that TPlatform always exists.

static str TPlatform::GetConfig (str *setting*, str *defaultSetting* = "") `[static]`

Query for a configuration setting.

Settings are collected from settings.xml, and can be overridden by command line options.

Options are set on the command line using one command line parameter, **options**. Options are concatenated using HTTP-GET semantics: Multiple options are appended using &.

For example, to set option "first" to 1 and option "second" to 2, you would use the command line:

options=first=1&second=2

This somewhat odd syntax allows us to pass in a stream of arbitrary options from HTML through the ActiveX wrapper code.

Parameters:

setting Setting to query.
defaultSetting The value to return if the setting is not already configured.

Returns:

Value of that setting, or the value of defaultSetting if the setting is not found.

See also:

kCheatMode
kComputerId
kInstallKey
kBuildTag
kGameName
kGameVersion

kEncryptionKey
kHiscoreLocalOnly
kHiscoreAnonymous
kPublisherName

static bool TPlatform::IsEnabled (str *setting*) `[static]`

Query as to whether a setting is enabled.

Uses the same settings as GetConfig().

Parameters:

setting Setting to query.

Returns:

True if setting is enabled.

See also:

GetConfig()

static void TPlatform::SetConfig (str *setting*, str *value*) `[static]`

Set a client configuration value.

Use this to set the encryption key, or to store a value to later be retrieved by GetConfig().

Parameters:

setting Setting to modify
value New value for setting.

See also:

GetConfig()

class TSoundManager* TPlatform::GetSoundManager ()

Get the sound manager.

Returns:

The application sound manager.

class TWindowManager* TPlatform::GetWindowManager ()

Get the application window manager.

Returns:

The TWindowManager created by TPlatform.

TTaskList* TPlatform::GetTaskList ()

Get the application task list.

Returns:

The TTaskList that holds the system's tasks.

class TStringTable* TPlatform::GetStringTable ()

Get the string table.

Returns:

The TStringTable created by TPlatform

void TPlatform::SetWindowTitle (str *title*)

Set the window application title.

Parameters:

title Title of application.

void TPlatform::OpenBrowser (const char * *url*)

Open a URL in a Web browser on the target system.

Parameters:

url URL to open.

void TPlatform::GetEvent (TEvent * *pEvent*)

Get an event from the application event queue.

See also:

TWindowManager::HandleEvent()

Parameters:

pEvent Event to process.

bool TPlatform::StringToClipboard (str *copyString*)

Send a string to the system clipboard.

Parameters:

copyString String to copy.

str TPlatform::StringFromClipboard ()

Retrieve a string, if any, from the current clipboard.

Returns:

A string representation of the current copy buffer.

void TPlatform::Exit (int32_t *exitValue* = 0)

Exit the program.

This function does return, but the program will exit on its next pass through the main event loop.

Parameters:

exitValue Exit code.

void TPlatform::SetDisplay (uint32_t *width*, uint32_t *height*, bool *fullscreen*)

Initialize the current display mode.

Parameters:

width Width of target display
height Height of target display
fullscreen True for fullscreen.

void TPlatform::GetDisplay (uint32_t * *pWidth*, uint32_t * *pHeight*, bool * *pbFullscreen*)

Get the current display parameters.

Parameters:

pWidth Width of display.
pHeight Height of display.
pbFullscreen True for fullscreen.

void TPlatform::SetCursor (TTextureRef *texture*, TPoint *hotSpot*, bool *hardware* = `false`)

Set a mouse cursor to an image.

Use 100% magenta (RGB=255,0,255) for transparent color in a software cursor. For hardware cursors, a normal image with alpha is appropriate.

Playground retains both the hardware and software cursor, and uses the one appropriate to the current hardware and windowing state. Because of this, you should always provide both a software and hardware cursor to Playground when you want to be sure the custom cursor will be visible.

Parameters:

texture Texture to use. Set to TTextureRef() (i.e., NULL) to disable software cursor. A software cursor texture needs to be a power-of-two in size and square. A hardware texture must be exactly 32x32 pixels, must *not* be simple, and uses normal alpha transparency (magenta will be magenta!).

You should always supply both a software cursor and a hardware cursor if you want a custom cursor to be available.

Parameters:

hotSpot Point within texture that the hot spot should be (i.e., the point where the clicking happens).
hardware True to set the hardware cursor, false to set the software cursor.

void TPlatform::ShowCursor (bool *show*)

Show or hide the cursor.

Parameters:

show True to show the cursor.

void TPlatform::SetCursorPos (const TPoint & *at*)

Set the cursor position.

Deprecated

This API doesn't quite work as expected on the Mac, and so will be removed from a future version of Playground. To achieve relative cursor functionality, please use SetCursorMode(TPlatform::kCursorModeDelta) instead.

Parameters:

at Position to set mouse cursor, in application window coordinates.

void TPlatform::SetCursorMode (ECursorMode *mode*)

Set the cursor mode.

Parameters:

mode Set the data mode for kMouseMove events.

bool TPlatform::IsForeground ()

Return true if the application is currently the foreground window.

Returns:

True if application is in the foreground.

void TPlatform::SetForeground ()

Set the game window to the foreground.

On some systems, this is more of a request than an action, but it will get the user's attention.

bool TPlatform::SetFullscreen (bool *bFullscreen*)

Convenience function for toggling fullscreen.

Parameters:

bFullscreen True for full-screen.

Returns:

True on success.

bool TPlatform::IsFullscreen ()

True if the window is full screen.

Returns:

True on full screen.

void TPlatform::AdoptTextureRefreshListener (TTask * *rn*)

Add a Texture Refresh Listener: On some platforms (DirectX) there are situations that cause all textures to be destroyed.

If you have a texture that was created by loading it from a file or resource, it will be automatically rebuilt by the library. If your texture is created programmatically, however, you will need to recreate it manually when a texture-loss event occurs.

The library internally maintains a list of Texture Refresh Listeners that all get called when system textures need to be rebuilt. Add a TTask listener to the list using this function.

Note that the TTask::DoTask() return value is respected, so be sure to return true unless you want your refresh listener to self-destruct.

Parameters:

rn A Texture Refresh listener to add to the internal list of functions to call when all textures (surfaces) are lost.

bool TPlatform::OrphanTextureRefreshListener (TTask * *rn*)

Remove a Texture Refresh Listener.

Parameters:

rn The Listener to remove.

Returns:

true if listener was removed, false if listener was not found

See also:

AddTextureRefreshListener()

uint32_t TPlatform::Timer ()

A count in milliseconds since the program has initialized,.

Returns:

Time value in milliseconds.

void TPlatform::Sleep (uint32_t *ms*)

Sleep the program for a number of milliseconds.

Parameters:

ms Number of milliseconds to sleep.

bool TPlatform::OrphanTask (TTask * *task*)

Release a task from the global task list.

Releases ownership of the TTask pointer to the calling function.

Parameters:

task Task to release.

Returns:

true if task was removed, false if task was not found

void TPlatform::AdoptTask (TTask * *task*)

Add a task to the global task list.

Transfers ownership to the global task list, which will expect to destroy the task when it is complete.

Parameters:

task Task to adopt.

15.47.4 Member Data Documentation

const char* TPlatform::kComputerId `[static]`

Unique Computer Identifier.

See also:

GetConfig

const char* TPlatform::kCheatMode [static]

Cheat mode.

Be sure to set your kEncryptionKey before querying this value.

See also:

GetConfig
kEncryptionKey

const char* TPlatform::kInstallKey [static]

A key unique to the install of this application.

Used to detect which portal the game was installed from.

const char* TPlatform::kEncryptionKey [static]

The application's encryption key.

The application MUST set this key to a key assigned by PlayFirst to function correctly with preferences, high scores, and the cheat enabler.

See also:

kCheatMode

const char* TPlatform::kBuildTag [static]

Query how this particular build has been tagged.

Build tagging may be used to enable/disable certain feature sets.

const char* TPlatform::kGameName [static]

What is the name of this game? Defined to be the string "gamename".

Set this in your "Main" routine to set the name your game should report to the high score server. If you want to change the name your game gets in the data folder, you must call this function from PlaygroundInit().

See also:

PlaygroundInit

const char* TPlatform::kPublisherName [static]

What is the name of the publisher of this game? Defaults to "PlayFirst".

This value is used to determine part of the path for the the user: and common: folders.

In order for this setting to have an effect on the user folders for the application, it must be set in the special initialization routine PlaygroundInit().

See also:

PlaygroundInit

const char* TPlatform::kTextGraphicSpriteRender [static]

Call TPlatform::SetConfig(TPlatform::kTextGraphicSpriteRender, 1) to default all new TTextGraphic instances to "sprite rendering" mode.

See also:

TTextGraphic::SetSpriteRender

const char* TPlatform::kGameVersion [static]

What version/build is this EXE? Defined to be the string "version".

Call TPlatform::GetConfig(TPlatform::kGameVersion) to query the version number of your game. Do not read version.h directly.

const char* TPlatform::kPFGameHandle [static]

This value is the game handle that the game uses to communicate with any PlayFirst services, such as the hiscore system.

This value should be set with the PFGAMEHANDLE value in key.h on application startup. See the Playground Skeleton sample for code that does this.

const char* TPlatform::kPFGameModeName [static]

This value is the prefix to the game mode names used to communicate with the PlayFirst Hiscore system.

To get the game mode name, you need to add on the number for the game mode you want, starting with index 1. So for example, to get the first game mode, you could ask for str(kPFGameModeName) + "1". This value should be set with the PFGAMEMODENAMES value in key.h on application startup. See the Playground Skeleton sample for code that does this.

const char* TPlatform::kPFGameMedalName [static]

This value is the prefix to the medal names used to communicate with the PlayFirst Hiscore system.

To get the medal name, you need to add on the number for the game mode you want, starting with index 1. So for example, to get the first medal, you could ask for str(kPFGameMedalName) + "1". This value should be set with the PFGAMEMEDALNAMES value in key.h on application startup. See the Playground Skeleton sample for code that does this.

const char* TPlatform::kVsyncWindowedMode [static]

Instruct Playground to wait for the vertical blanking period to start prior to drawing to the screen in windowed mode.

Has no effect on full-screen mode, which always flips its buffer at the vertical refresh.

Can minimize "tearing" when enabled. Does cause the game to pause to wait for the screen vertical refresh to happen, which can seriously slow down the frame rate of your game by causing it to skip frames when the time to compute core logic and render a frame takes slightly longer than one video frame to calculate.

For instance, if the game takes 1/60 of a second (16.66ms) to perform all calculations and finish rendering, but the monitor is set to a 70Hz update, setting this flag will cause the screen to update at only 35FPS (half of 70FPS), since by the time one frame is complete, it will have already missed the next video synchronization. As a result,

the game will need to wait for the following frame vertical refresh to draw. If you can only draw once every two frames at 70Hz, you'll get a 35Hz (35FPS) update.

Set the value to "1" to enable, or "0" to disable this feature. Defaults to being disabled.

const char* TPlatform::kUTF8Mode [static]

Enable UTF-8 support mode.

- Enables TWindow::OnUTF8Char() dispatch.
- When enabled, OnChar() will only be called for low-ASCII characters, and only if no OnUTF8Char() has already handled the character.

Warning:

This mode has not yet been approved for use in PlayFirst-developed games.
Must be enabled in PlaygroundInit() section.

const char* TPlatform::kAutoMergeMask [static]

Automatically merge any .mask.png files when loading a .jpg file.

When you call:

TTexture::Get("foo.jpg");

...then if kAutoMergeMask is enabled, it will look for foo.mask.png and, if found, automatically perform a TTexture::GetMerged("foo.jpg", "foo.mask.png").

const char* TPlatform::kMonitorFrequency [static]

Query the current monitor frequency.

Returns "0" if the monitor frequency is not available. Can change when the computer switches between full screen and windowed mode.

const char* TPlatform::kFlashMajorVersion [static]

Raise the minimum major Flash version that your game requires.

The value should be a version number in quotes. For instance, to require the Flash 8 control to be installed on a system, set the value to "8".

const char* TPlatform::kKeepOGGUncompressed [static]

Enabling this will decompress any non-streamed .ogg sounds into memory, rather than keeping them compressed.

This will reduce the time it takes to decompress the sound while it is playing, but comes at a cost of increased memory usage.

const char* TPlatform::kLargeVRAMUsage [static]

Certain video cards do not handle VRAM paging well.

If your game uses more than 128MB VRAM, you might consider setting this flag, which will force the game to run in 16-bit mode on certain cards and certain OS's that have exhibited VRAM paging problems.

15.48 TPoint Class Reference

`#include <pf/point.h>`

15.48.1 Detailed Description

The TPoint class is a 2d integer point representation.

Public Member Functions

- TPoint ()
  Default constructor. Zeros all members.
- TPoint (int32_t x, int32_t y)
  Initializing constructor.
- bool operator== (const TPoint &point) const
  Equality.
- bool operator!= (const TPoint &point) const
  Inequality.
- TPoint & operator+= (const TPoint &point)
  Add a point to this point.
- TPoint operator+ (const TPoint &point) const
  Memberwise addition.
- TPoint & operator-= (const TPoint &point)
  Subtract a point from this point.
- TPoint operator- (const TPoint &point) const
  Memberwise subtraction.
- TPoint operator- ()
  Negation.

Public Attributes

- int32_t x
  Horizontal coordinate.
- int32_t y
  Vertical coordinate.

15.49 TPrefs Class Reference

`#include <pf/prefs.h>`

15.49.1 Detailed Description

The TPrefs class is designed to help with the saving of preferences for a game.
Preferences can be stored at either a global level or a user level.
See note about unique installs in the TPrefs() constructor comments.

Public Member Functions

- TPrefs (bool saveData=true)
  Constructor.
- ~TPrefs ()
  Destructor.
- uint32_t GetNumUsers ()
  Get the number of users.
- void DeleteUser (uint32_t userNum)
  Deletes the current user at the specified slot, and slides all users above that slot down.
- int32_t GetInt (str prefName, int32_t defaultValue, int32_t userIndex=kGlobalIndex)
  Gets an int from the preferences.
- void SetInt (str prefName, int32_t value, int32_t userIndex=kGlobalIndex, bool save=true)
  Sets an int in the preferences.
- str GetStr (str prefName, str defaultValue, int32_t userIndex=kGlobalIndex)
  Gets a str from the preferences.
- void SetStr (str prefName, str value, int32_t userIndex=kGlobalIndex, bool save=true)
  Sets a str in the preferences.
- int32_t GetBinary (str prefName, void *buffer, uint32_t bufferLen, int32_t userIndex=kGlobalIndex)
  Gets a binary block from the preferences.
- void SetBinary (str prefName, const void *data, uint32_t dataLength, int32_t userIndex=kGlobalIndex, bool save=true)
  Sets a block of binary data in the preferences.
- void SavePrefs ()
  Commit preferences to permanent storage.

- str GetUserStr (int32_t userIndex)

  Gets all the user data as a str.

- void SetUserStr (int32_t userIndex, str data)

  Sets a user data from a passed in str.

Static Public Attributes

- static const int32_t kGlobalIndex = -1

  An index to pass in as a userIndex parameter to any get/set function to get/set a global preference.

15.49.2 Constructor & Destructor Documentation

TPrefs::TPrefs (bool *saveData* = `true`)

Constructor.

Prior to calling the constructor, you need to set the application encryption key. See TPlatform::SetConfig().

Warning:

Be sure to save the game periodically during play. Any time the user has, e.g., gained a new level, or any other significant event the game should remember, you should tell it to write a save file. The reason is that, when your game is wrapped in DRM, when the DRM expires it will often directly send a kQuit message, which terminates the (default) game loop.

If there are performance reasons you don't want to write the file frequently, you can use the option in SetInt() and SetStr() to NOT write the file immediately; as long as the setting has been given to TPrefs, it will be written on game exit.

Selecting a Specific Configuration

In order to prevent games from being downloaded multiple times and being able to use the same saved games, a TPrefs object attempts to distinguish between unique installs of the game. It does this by looking at the directory the game is running from.

Therefore, for debugging purposes, if you want your game to always be treated as the same install, you can hand create an install.txt file in the assets folder. This file can do two things:

1) If the file is empty, TPrefs will load/save preferences as if it was running from the most recently created preferences location (or create a new location if one doesn't exist).

2) If the file is not empty, it will treat the contents of this file as the install path, and read/save games as if it was that install. (example: putting C:/game/mygame in the install.txt file will make the TPrefs object pretend the game is running from c:/game/mygame

Parameters:

saveData Whether or not data should be saved between sessions, default is true. For example, in a web game you might not want scores to persist between sessions, in which case saveData should be false.

15.49.3 Member Function Documentation

uint32_t TPrefs::GetNumUsers ()

Get the number of users.

Users must be numbered sequentially, and there may not be more than 1023 users.

Returns:

The number of currently created users in the preferences.

void TPrefs::DeleteUser (uint32_t *userNum*)

Deletes the current user at the specified slot, and slides all users above that slot down.

If user 2 is deleted, then user 3 becomes user 2, 4 becomes 3, etc.

Parameters:

userNum Which user to delete

int32_t TPrefs::GetInt (str *prefName*, int32_t *defaultValue*, int32_t *userIndex* = kGlobalIndex)

Gets an int from the preferences.

Parameters:

prefName Name of preference to get.
defaultValue If this preference has not been set, what value should be returned.
userIndex Which user to get from, or if this is kGlobalIndex then get from the global preferences. Users must be numbered sequentially, and there may not be more than 1023 users.

Returns:

Returns the stored int if it exists, or else defaultValue if it does not exist.

void TPrefs::SetInt (str *prefName*, int32_t *value*, int32_t *userIndex* = kGlobalIndex, bool *save* = `true`)

Sets an int in the preferences.

Parameters:

prefName Name of preference to set.
value Value to set preference to.
userIndex Which user to set, or if this is kGlobalIndex then set a global preference. Users must be numbered sequentially, and there may not be more than 1023 users.
save If this is true, commits preferences to disk. if false, preferences are changed in memory only, until a different preference is set with save set to true, or until SavePrefs() is called.

str TPrefs::GetStr (str *prefName*, str *defaultValue*, int32_t *userIndex* = kGlobalIndex)

Gets a str from the preferences.

Parameters:

prefName Name of preference to Get.
defaultValue If this preference has not been set, what value should be returned.
userIndex Which user to get from, or if this is kGlobalIndex then get from the global preferences. Users must be numbered sequentially, and there may not be more than 1023 users.

Returns:

Returns the stored str if it exists, or else defaultValue if it does not exist.

void TPrefs::SetStr (str *prefName*, str *value*, int32_t *userIndex* = kGlobalIndex, bool *save* = `true`)

Sets a str in the preferences.

String length is limited to 4Mb-16 bytes.

Parameters:

prefName Name of preference to set.
value Value to set preference to.
userIndex Which user to set, or if this is kGlobalIndex then set a global preference. Users must be numbered sequentially, and there may not be more than 1023 users.
save If this is true, commits preferences to disk. if false, preferences are changed in memory only, until a different preference is set with save set to true, or until SavePrefs() is called.

int32_t TPrefs::GetBinary (str *prefName*, void * *buffer*, uint32_t *bufferLen*, int32_t *userIndex* = kGlobalIndex)

Gets a binary block from the preferences.

Parameters:

prefName Name of preference to get.
buffer Address of location to store data in.
bufferLen Size of buffer in bytes.
userIndex Which user to get from, or if this is kGlobalIndex then get from the global preferences. Users must be numbered sequentially, and there may not be more than 1023 users.

Returns:

If the preference does not exist, -1 is returned. If buffer is successfully filled in with the preference, 0 is returned. If the preference exists and buffer is NULL or bufferLen is too small to store the data, then the size that the buffer needs to be is returned.

void TPrefs::SetBinary (str *prefName*, const void * *data*, uint32_t *dataLength*, int32_t *userIndex* = kGlobalIndex, bool *save* = `true`)

Sets a block of binary data in the preferences.

Binary data size is limited to 4Mb-16 bytes.

Parameters:

prefName Name of preference to set.
data Address of data to set in preferences.
dataLength Size of data in bytes.
userIndex Which user to set, or if this is kGlobalIndex then set a global preference. Users must be numbered sequentially, and there may not be more than 1023 users.
save If this is true, commits preferences to disk. If false, preferences are changed in memory only, until a different preference is set with save set to true, or until SavePrefs() is called.

str TPrefs::GetUserStr (int32_t *userIndex*)

Gets all the user data as a str.

Parameters:

userIndex Which user to get data from.

Returns:

A str containing all the user's data.

void TPrefs::SetUserStr (int32_t *userIndex*, str *data*)

Sets a user data from a passed in str.

In order to maintain proper functionality this should only be called with a str return from GetUserStr.

Parameters:

userIndex Which user to set data.
data String to set.

15.50 TPrefsDB Class Reference

`#include <pf/prefsdb.h>`

15.50.1 Detailed Description

The TPrefsDB class is designed to help with the saving of preferences for a game.

Preferences can be stored at either a global level or a user level.

See note about unique installs in the TPrefsDB() constructor comments.

TPrefsDB differs from TPrefs in the method of saving the data; it uses a database format that can save part of the file instead of rewriting the entire file each time, so if your game has a large data save requirement, it will likely be much faster to save if you switch from TPrefs to TPrefsDB.

Public Member Functions

- TPrefsDB (bool saveData=true)
  Constructor.
- ~TPrefsDB ()
  Destructor.
- uint32_t GetNumUsers ()
  Get the number of users.
- void DeleteUser (uint32_t userNum)
  Deletes the current user at the specified slot, and slides all users above that slot down.
- int32_t GetInt (str prefName, int32_t defaultValue, int32_t userIndex=kGlobalIndex)
  Gets an int from the preferences.
- void SetInt (str prefName, int32_t value, int32_t userIndex=kGlobalIndex, bool save=true)
  Sets an int in the preferences.
- str GetStr (str prefName, str defaultValue, int32_t userIndex=kGlobalIndex)
  Gets a str from the preferences.
- void SetStr (str prefName, str value, int32_t userIndex=kGlobalIndex, bool save=true)
  Sets a str in the preferences.
- int32_t GetBinary (str prefName, void *buffer, uint32_t bufferLen, int32_t userIndex=kGlobalIndex)
  Gets a binary block from the preferences.
- void SetBinary (str prefName, const void *data, uint32_t dataLength, int32_t userIndex=kGlobalIndex, bool save=true)
  Sets a block of binary data in the preferences.

- void SavePrefs ()

  Commit preferences to permanent storage.

Static Public Attributes

- static const int32_t kGlobalIndex = -1

  An index to pass in as a userIndex parameter to any get/set function to get/set a global preference.

15.50.2 Constructor & Destructor Documentation

TPrefsDB::TPrefsDB (bool *saveData* = `true`)

Constructor.

Prior to calling the constructor, you need to set the application encryption key. See TPlatform::SetConfig().

Parameters:

saveData Whether or not data should be saved between sessions, default is true. For example, in a web game you might not want scores to persist between sessions, in which case saveData should be false.

15.50.3 Member Function Documentation

uint32_t TPrefsDB::GetNumUsers ()

Get the number of users.

Users must be numbered sequentially, and there may not be more than 1023 users.

Returns:

The number of currently created users in the preferences.

void TPrefsDB::DeleteUser (uint32_t *userNum*)

Deletes the current user at the specified slot, and slides all users above that slot down.

If user 2 is deleted, then user 3 becomes user 2, 4 becomes 3, etc.

Parameters:

userNum Which user to delete

int32_t TPrefsDB::GetInt (str *prefName*, int32_t *defaultValue*, int32_t *userIndex* = kGlobalIndex)

Gets an int from the preferences.

Parameters:

prefName Name of preference to get.
defaultValue If this preference has not been set, what value should be returned.
userIndex Which user to get from, or if this is kGlobalIndex then get from the global preferences. Users must be numbered sequentially, and there may not be more than 1023 users.

Returns:

Returns the stored int if it exists, or else defaultValue if it does not exist.

void TPrefsDB::SetInt (str *prefName*, int32_t *value*, int32_t *userIndex* = kGlobalIndex, bool *save* = `true`)

Sets an int in the preferences.

Parameters:

prefName Name of preference to set.
value Value to set preference to.
userIndex Which user to set, or if this is kGlobalIndex then set a global preference. Users must be numbered sequentially, and there may not be more than 1023 users.
save If this is true, commits preferences to disk. if false, preferences are changed in memory only, until a different preference is set with save set to true, or until SavePrefsdb() is called.

str TPrefsDB::GetStr (str *prefName*, str *defaultValue*, int32_t *userIndex* = kGlobalIndex)

Gets a str from the preferences.

Parameters:

prefName Name of preference to Get.
defaultValue If this preference has not been set, what value should be returned.
userIndex Which user to get from, or if this is kGlobalIndex then get from the global preferences. Users must be numbered sequentially, and there may not be more than 1023 users.

Returns:

Returns the stored str if it exists, or else defaultValue if it does not exist.

void TPrefsDB::SetStr (str *prefName*, str *value*, int32_t *userIndex* = kGlobalIndex, bool *save* = `true`)

Sets a str in the preferences.

String length is limited to 4Mb-16 bytes.

Parameters:

prefName Name of preference to set.
value Value to set preference to.
userIndex Which user to set, or if this is kGlobalIndex then set a global preference. Users must be numbered sequentially, and there may not be more than 1023 users.
save If this is true, commits preferences to disk. if false, preferences are changed in memory only, until a different preference is set with save set to true, or until SavePrefsdb() is called.

int32_t TPrefsDB::GetBinary (str *prefName*, void * *buffer*, uint32_t *bufferLen*, int32_t *userIndex* = kGlobalIndex)

Gets a binary block from the preferences.

Parameters:

prefName Name of preference to get.
buffer Address of location to store data in.
bufferLen Size of buffer in bytes.
userIndex Which user to get from, or if this is kGlobalIndex then get from the global preferences. Users must be numbered sequentially, and there may not be more than 1023 users.

Returns:

If the preference does not exist, -1 is returned. If buffer is successfully filled in with the preference, 0 is returned. If the preference exists and buffer is NULL or bufferLen is too small to store the data, then the size that the buffer needs to be is returned.

void TPrefsDB::SetBinary (str *prefName*, const void * *data*, uint32_t *dataLength*, int32_t *userIndex* = kGlobalIndex, bool *save* = `true`)

Sets a block of binary data in the preferences.

Binary data size is limited to 4Mb-16 bytes.

Parameters:

prefName Name of preference to set.
data Address of data to set in preferences.
dataLength Size of data in bytes.
userIndex Which user to set, or if this is kGlobalIndex then set a global preference. Users must be numbered sequentially, and there may not be more than 1023 users.
save If this is true, commits preferences to disk. If false, preferences are changed in memory only, until a different preference is set with save set to true, or until SavePrefsdb() is called.

15.51 TRandom Class Reference

`#include <pf/random.h>`

15.51.1 Detailed Description

A deterministic random number generator.

Based on the Mersenne Twister algorithm, it has a period of $219937 - 1$ iterations, it's as fast as or faster than rand(), and it's equally distributed in 623 dimensions.

Public Member Functions

- TRandom ()

  Default Constructor.

- void Seed (uint32_t s)

  Seed with a 32-bit number.

- void SeedArray (unsigned long init_key[], uint32_t key_length)

  Seed with array.

- uint32_t SaveState (void *buffer, uint32_t bufferLength)

  Get the current random number generator state, which you can then use to restore to a known state using RestoreState.

- void RestoreState (void *buffer)

  Restore random number generator state from SaveState.

- unsigned long Rand32 ()

  Generates a random unsigned long.

- TReal RandFloat ()

  Generates a random number on [0,1)-real-interval.

- double RandDouble ()

  Generates a random number on [0,1)-real-interval, that is random out to double-precision granularity.

- int32_t RandRange (int32_t bottom, int32_t top)

  Generates a random integer between the two parameters, inclusive.

15.51.2 Member Function Documentation

void TRandom::Seed (uint32_t *s*)

Seed with a 32-bit number.

Parameters:

s A random 32-bit seed. Once seeded with a particular number, the sequence will always be the same.

void TRandom::SeedArray (unsigned long *init_key*[ ], uint32_t *key_length*)

Seed with array.

Use this if 4 billion possible sequences are not enough.

Parameters:

init_key Array of long integers.
key_length Number of long integers in the array.

uint32_t TRandom::SaveState (void * *buffer*, uint32_t *bufferLength*)

Get the current random number generator state, which you can then use to restore to a known state using RestoreState.

Parameters:

buffer - buffer to store state in
bufferLength - number of bytes in buffer

Returns:

0 if successful, otherwise returns the length that buffer needs to be for success;

void TRandom::RestoreState (void * *buffer*)

Restore random number generator state from SaveState.

Parameters:

buffer Buffer to read state from.

unsigned long TRandom::Rand32 ()

Generates a random unsigned long.

Returns:

A random 32-bit integer that is equally random across all 32 bits.

TReal TRandom::RandFloat ()

Generates a random number on [0,1)-real-interval.

Returns:

A number between zero and one, never equalling 1.0

double TRandom::RandDouble ()

Generates a random number on [0,1)-real-interval, that is random out to double-precision granularity.

Since a double has (on the Wintel reference platform) more bits of precision than an int/long, it takes two calls from Rand32 and combines them into a double precision random number.

Returns:

A number between zero and one, never equalling 1.0

int32_t TRandom::RandRange (int32_t *bottom*, int32_t *top*)

Generates a random integer between the two parameters, inclusive.

Will generate all options equally, including bottom and top.

Parameters:

bottom Lowest number that will be returned.
top Highest number that will be returned.

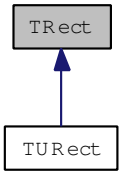
Returns:

A random integer.

15.52 TRect Class Reference

`#include <pf/rect.h>`

Inheritance diagram for TRect:



15.52.1 Detailed Description

A rectangle.

Like the Windows RECT class, the right and bottom (X2/Y2) coordinates are one past the edge of the rectangle.

Public Member Functions

- TRect ()
  Default constructor. Zeros all members.
- TRect (int32_t X1, int32_t Y1, int32_t X2, int32_t Y2)
  Initializing constructor.
- TRect (const TPoint &topLeft, const TPoint &bottomRight)
  Construct a TRect from two points.
- int32_t GetWidth () const
  Get the width of the rectangle.
- int32_t GetHeight () const
  Get the height of the rectangle.
- bool operator== (const TRect &r) const
  Equality.
- bool operator!= (const TRect &r) const
  Inequality.
- TRect & operator+= (const TPoint &p)
  Move a rect by a point.
- TRect & operator-= (const TPoint &p)
  Move a rect by a point.

- bool Contains (const TRect &r) const
  Test to see whether a rectangle is inside this rectangle.
- bool Contains (const TPoint &p) const
  Test to see whether a point is inside this rectangle.
- bool IsInside (const TPoint &p) const
  Test to see whether a point is inside this rectangle.
- bool Overlaps (const TRect &rect) const
  Test to see if another rectangle overlaps this rectangle.
- void Union (const TRect &rect1, const TRect &rect2)
  Make this rectangle the union of the two parameter rectangles.
- void Intersect (const TRect &rect1, const TRect &rect2)
  Make this rectangle the intersection of the two parameter rectangles.
- TPoint & GetTopLeft ()
  Get a TPoint that represents the upper left corner of the rectangle.
- const TPoint & GetTopLeft () const
  Get a TPoint that represents the upper left corner of the rectangle.
- TPoint & GetBottomRight ()
  Get a TPoint that represents the lower right corner of the rectangle.
- const TPoint & GetBottomRight () const
  Get a TPoint that represents the lower right corner of the rectangle.

Static Public Member Functions

- static TRect FromXYWH (int32_t x, int32_t y, int32_t w, int32_t h)
  Build a TRect from the upper left corner and the desired width and height.

Public Attributes

- int32_t x1
  Upper left corner x coordinate.
- int32_t y1
  Upper left corner y coordinate.
- int32_t x2
  Lower right corner x coordinate.
- int32_t y2
  Lower right corner y coordinate.

15.52.2 Constructor & Destructor Documentation

TRect::TRect (const TPoint & *topLeft*, const TPoint & *bottomRight*)

Construct a TRect from two points.

Parameters:

topLeft Upper left corner of the TRect.
bottomRight Lower right corner of the TRect.

15.52.3 Member Function Documentation

static TRect TRect::FromXYWH (int32_t *x*, int32_t *y*, int32_t *w*, int32_t *h*) `[static]`

Build a TRect from the upper left corner and the desired width and height.

Parameters:

x X coordinate of the upper left corner.
y Y coordinate of the upper left corner.
w Width
h Height

Returns:

A new TRect calculated from X,Y,W,H

int32_t TRect::GetWidth () const

Get the width of the rectangle.

Returns:

Exclusive width of the rectangle. (x2-x1)

int32_t TRect::GetHeight () const

Get the height of the rectangle.

Returns:

Exclusive height of the rectangle. (y2-y1)

Referenced by TTextGraphic::Draw().

TRect& TRect::operator+= (const TPoint & *p*)

Move a rect by a point.

Parameters:

p Point to offset rect by.

Returns:

A reference to this.

TRect& TRect::operator-= (const TPoint & *p*)

Move a rect by a point.

Parameters:

p Point to offset rect by.

Returns:

A reference to this.

bool TRect::Contains (const TRect & *r*) const

Test to see whether a rectangle is inside this rectangle.

Parameters:

r TRect to test.

Returns:

True if r is inside this.

References x1, x2, y1, and y2.

bool TRect::Contains (const TPoint & *p*) const

Test to see whether a point is inside this rectangle.

Parameters:

p Point to test.

Returns:

True if inside. If p.x==x2 or p.y==y2, the test fails.

References TPoint::x, and TPoint::y.

bool TRect::IsInside (const TPoint & *p*) const

Test to see whether a point is inside this rectangle.

Deprecated

This function is going to be deleted in favor of the more clearly named TRect::Contains().

Parameters:

p Point to test.

Returns:

True if inside. If p.x==x2 or p.y==y2, the test fails.

bool TRect::Overlaps (const TRect & *rect*) const

Test to see if another rectangle overlaps this rectangle.

Parameters:

rect Rectangle to test.

Returns:

True on overlap.

References x1, x2, y1, and y2.

void TRect::Union (const TRect & *rect1*, const TRect & *rect2*)

Make this rectangle the union of the two parameter rectangles.

Can pass this rectangle as either parameter.

Parameters:

rect1 First
rect2 Second

void TRect::Intersect (const TRect & *rect1*, const TRect & *rect2*)

Make this rectangle the intersection of the two parameter rectangles.

Can pass this rectangle as either parameter.

Warning:

Assumes the two rectangles overlap. Resulting value is undefined if the original rectangles do not overlap.

Parameters:

rect1 First
rect2 Second

TPoint& TRect::GetTopLeft ()

Get a TPoint that represents the upper left corner of the rectangle.

Returns:

Corner point

Referenced by TWindow::ClientToParent(), TWindow::ClientToScreen(), TWindow::ParentToClient(), and TWindow::ScreenToClient().

const TPoint& TRect::GetTopLeft () const

Get a TPoint that represents the upper left corner of the rectangle.

Returns:

Corner point

TPoint& TRect::GetBottomRight ()

Get a TPoint that represents the lower right corner of the rectangle.

Returns:

Corner point

Referenced by TWindow::ClientToParent(), TWindow::ClientToScreen(), TWindow::ParentToClient(), and TWindow::ScreenToClient().

const TPoint& TRect::GetBottomRight () const

Get a TPoint that represents the lower right corner of the rectangle.

Returns:

Corner point

15.53 TRenderer Class Reference

`#include <pf/renderer.h>`

15.53.1 Detailed Description

The interface to the rendering subsystem.

Available as a singleton while the game is running using TRenderer::GetInstance().

Raw Primitive Drawing.

Routines for drawing groups of lines or triangles, given either 2d or 3d vertex data.

Must be called in a window Draw() function, or between BeginRenderTarget/EndRenderTarget.

Respects the currently active 2d or 3d rendering environment. Functions are indifferent to Begin2d/Begin3d/End2d/End3d.

- enum EDrawType {
  kDrawPoints = 1, kDrawLines, kDrawLineStrip, kDrawTriangles,
  kDrawTriStrip, kDrawTriFan }
  Type for DrawVertices.
- static const int kMaxIndices = 65535
  The maximum number of indices that can be passed into DrawIndexedVertices().
- void DrawVertices (EDrawType type, const TVertexSet &vertices)
  Draw a set of vertices using the current environment.
- void DrawIndexedVertices (EDrawType type, const TVertexSet &vertices, uint16_t *indices, uint32_t indexCount)
  Draw a set of vertices using the current environment.

2d/3d Agnostic Code

These functions allow you to modify the 2d or 3d rendering environment.

Some of these states are cleared/reset by calling Begin2d or Begin3d, so don't expect any state to be preserved across those calls.

- enum EShadeMode { kShadeFlat = 1, kShadeGouraud }
  Shading modes.
- enum EBlendMode {
  kBlendNormal, kBlendOpaque, kBlendAdditiveAlpha, kBlendSubtractive,
  kBlendMultiplicative, kBlendINVALID = -1 }
  Blending modes for SetBlendMode().

- enum EFilteringMode { kFilterPoint, kFilterLinear }

  Filtering modes.

- enum ETextureMapMode { kMapClamp, kMapWrap, kMapMirror }

  The various texture map modes.

- void SetShadeMode (EShadeMode shadeMode)

  Set the shade mode.

- bool RenderTargetIsScreen ()

  Query as to whether the current render target is the screen.

- void SetTexture (TTextureRef pTexture=TTextureRef())

  Set the current rendering texture.

- TTextureRef GetTexture ()

  Get the currently assigned texture.

- void SetBlendMode (EBlendMode blendMode)

  Set the current blend mode.

- void SetFilteringMode (EFilteringMode filteringMode)

  Set the current filtering mode to use when scaling images.

- void SetTextureMapMode (ETextureMapMode umap, ETextureMapMode vmap)

  Set the texture map mode for use when texture coordinates extend beyond the edge of the internal texture.

- void SetZBufferWrite (bool writeToZbuffer)

  Enable or disable writing to zbuffer.

- void SetZBufferTest (bool testZbuffer)

  Enable zbuffer test.

- void SetColorWrite (bool colorWrite)

  Enable writing of RGB color data.

- bool GetColorWrite ()

  Get the current "color write" state.

- void ClearZBuffer ()

  Clear the ZBuffer.

- bool GetZBufferWrite ()

  Query ZBuffer-write state.

- bool GetZBufferTest ()

  Query ZBuffer-test state.

Scene Management Functions

Enable drawing to a render target or the backbuffer, or set up a 2d or 3d rendering context.

Drawing to the backbuffer is usually handled by the system; by the time a TWindow::Draw() function is called, you are already in a rendering state.

- enum ERenderTargetMode {
  kFullRenderRGB1, kFullRenderRGBX, kMergeRenderRGB1, kMergeRenderRGBX,
  kMergeRenderXXXA }
  Modes for TRenderer::BeginRenderTarget.
- bool BeginRenderTarget (TTextureRef texture, ERenderTargetMode mode)
  Start rendering to a texture.
- void EndRenderTarget ()
  Complete the rendering to a texture.
- bool BeginDraw (bool needRefresh)
  Open an internal draw context.
- void EndDraw (bool flip=true)
  Close and complete an internal draw context.
- bool Begin2d ()
  Begin 2d rendering.
- void End2d ()
  Finish a 2d rendering set.
- bool Begin3d ()
  Begin 3d rendering.
- void End3d ()
  Finish a 3d rendering set.
- bool InDraw () const
  Query as to whether we're currently in a drawing mode.

3d-Related functions

- enum ECullMode { kCullNone = 1, kCullCW, kCullCCW }
  Possible cull modes.
- void SetWorldMatrix (TMat4 *pMatrix)
  Set the world matrix.
- void SetViewMatrix (TMat4 *pMatrix)
  Set the view matrix.

- void SetProjectionMatrix (TMat4 *pMatrix)

  Set the projection matrix.

- void SetView (const TVec3 &eye, const TVec3 &at, const TVec3 &up)

  Set the view matrix based on the viewer location, a target location, and an up vector.

- void SetPerspectiveProjection (TReal nearPlane, TReal farPlane, TReal fov=PI/4.0f, TReal aspect=0)

  Set a perspective projection matrix.

- void SetOrthogonalProjection (TReal nearPlane, TReal farPlane)

  Set an orthogonal projection matrix.

- void GetWorldMatrix (TMat4 *m)

  Get the current world matrix.

- void GetViewMatrix (TMat4 *m)

  Get the current view matrix.

- void GetProjectionMatrix (TMat4 *m)

  Get the current projection matrix.

- void SetAmbientColor (const TColor &color)

  Set the scene's ambient color.

- void SetCullMode (ECullMode cullMode)

  Set the current cull mode of the 3d render.

- void SetMaterial (TMaterial *mat)

  Set the current rendering material.

- void SetLight (uint32_t index, TLight *light)

  Set up one of the scene lights.

- bool ToggleHUD ()

  Toggle the HUD.

- bool In2d () const

  Currently in Begin2d mode.

- bool In3d () const

  Currently in Begin3d mode.

- void SetOption (str option, str value)

  Set a renderer option.

- str GetOption (str option)

  Get a renderer option.

Public Member Functions

- ~TRenderer ()
  Destructor.
- void SetViewport (const TRect &viewport)
  Set the current rendering viewport.
- void GetViewport (TRect *viewport)
  Get the current rendering viewport.
- void SetClippingRectangle (const TURect &clip)
  Set the current screen clipping rectangle.
- TRect GetClippingRectangle ()
  Get the current screen clipping rectangle.
- void PushClippingRectangle (const TURect &clip)
  Push a clipping rectangle onto the internal stack.
- void PopClippingRectangle ()
  Pop a clipping rectangle from the internal stack.
- void PushViewport (const TRect &viewport)
  Push a viewport on the internal viewport stack.
- void PopViewport ()
  Restore a viewport from the viewport stack.

Information

- str GetSystemData ()
  Get data about the current system.
- bool GetTextureSquareFlag ()
  Query whether textures on this computer will be created as square internally.

2d-Related Functions

- void FillRect (const TURect &rect, const TColor &color, TTextureRef dst=TTextureRef())
  Fill a rectangle, optionally specifying a destination texture.
- bool IsCapturePending ()
  Query as to whether the previously set capture texture is still waiting to be filled with the screen render.
- void SetCaptureTexture (TTextureRef texture)
  Set a capture texture.
- TTextureRef GetCaptureTexture ()
  Get the current capture texture from the renderer.

Static Public Member Functions

- static TRenderer * GetInstance ()

  Accessor.

15.53.2 Member Enumeration Documentation

enum TRenderer::EDrawType

Type for DrawVertices.

Enumerator:

kDrawPoints DrawVertices renders the vertices as a collection of isolated points.
kDrawLines DrawVertices renders the vertices as a set of isolated line segments.
kDrawLineStrip DrawVertices renders the vertices as a line strip.
kDrawTriangles DrawVertices renders each group of 3 vertices as a triangle.
kDrawTriStrip DrawVertices renders the vertices as a triangle strip.
kDrawTriFan DrawVertices renders the vertices as a triangle fan.

enum TRenderer::EShadeMode

Shading modes.

See also:

TRenderer::SetShadeMode

Enumerator:

kShadeFlat Flat shading with no shading across a polygon.
kShadeGouraud Gouraud shading with smooth shading across a polygon.

enum TRenderer::EBlendMode

Blending modes for SetBlendMode().

Enumerator:

kBlendNormal Normal alpha drawing.
kBlendOpaque Draw with no alpha.
This mode requires your texture to be square and each side be a power of two in order to work consistently.
kBlendAdditiveAlpha Additive drawing, respecting source alpha. Useful for "glowing" effects.
kBlendSubtractive Subtractive drawing; useful for shadows or special effects.
kBlendMultiplicative Multiplicative drawing; can also be used for shadows or special effects.
kBlendINVALID Invalid blend mode.

enum TRenderer::EFilteringMode

Filtering modes.

See also:

SetFilteringMode

Enumerator:

kFilterPoint Point-filtering: Select the nearest pixel.
kFilterLinear Bilinear (or trilinear for MIPMAPs) filtering: Blend the nearest pixels.

enum TRenderer::ETextureMapMode

The various texture map modes.

See also:

TRenderer::SetTextureMapMode()

Enumerator:

kMapClamp Clamp the texture to 0,1.
kMapWrap Repeat texture coordinates.
Uses the fractional part of the texture coordinates only. Do not expect wrap to work with arbitrarily large numbers, as some video cards limit wrapping to as little as ± 4.
kMapMirror Mirror coordinate: 0->1->0->1.

enum TRenderer::ERenderTargetMode

Modes for TRenderer::BeginRenderTarget.

Enumerator:

kFullRenderRGB1 Render RGB with an opaque alpha.
Does not preserve the current texture contents.
kFullRenderRGBX Render RGB with an undefined alpha.
Does not preserve the current texture contents. Alpha contents are **undefined**.
kMergeRenderRGB1 Render RGB with an opaque alpha, preserving the current texture contents; entire final surface (not just areas that are overdrawn) will get an opaque alpha.
Note that this call is actually slower than kFullRenderRGB1, since the existing texture data needs to be copied to the render surface.
kMergeRenderRGBX Render RGB with an opaque alpha, preserving the current texture contents; final surface alpha is unchanged.
Note that this call is actually slower than kFullRenderRGBX, since the existing texture data needs to be copied to the render surface.
kMergeRenderXXXA Render Alpha without modifying RGB pixels.
RGB of original image will be preserved.

enum TRenderer::ECullMode

Possible cull modes.

Enumerator:

kCullNone No culling.
kCullCW Cull faces with clockwise vertices.
kCullCCW Cull faces with counterclockwise vertices.

15.53.3 Member Function Documentation

void TRenderer::SetViewport (const TRect & *viewport*)

Set the current rendering viewport.

Only affects the 3d rendering functions, as the viewport is overridden internally by the TTexture::Draw functions.

Parameters:

viewport Viewport

void TRenderer::GetViewport (TRect * *viewport*)

Get the current rendering viewport.

Parameters:

viewport Viewport

void TRenderer::SetClippingRectangle (const TURect & *clip*)

Set the current screen clipping rectangle.

Parameters:

clip Rectangle to clip to in screen coordinates.

void TRenderer::PushClippingRectangle (const TURect & *clip*)

Push a clipping rectangle onto the internal stack.

Parameters:

clip Clipping rectangle to push.

void TRenderer::PushViewport (const TRect & *viewport*)

Push a viewport on the internal viewport stack.

Parameters:

viewport Viewport

void TRenderer::DrawVertices (EDrawType *type*, const TVertexSet & *vertices*)

Draw a set of vertices using the current environment.

There is a hard limit on the number of vertices of TVertexSet::kMaxVertices.

Parameters:

type Type of data to draw.
vertices A set of vertices.

void TRenderer::DrawIndexedVertices (EDrawType *type*, const TVertexSet & *vertices*, uint16_t * *indices*, uint32_t *indexCount*)

Draw a set of vertices using the current environment.

There is a hard limit on the number of vertices of TVertexSet::kMaxVertices, and on the number of indices of kMaxIndices.

Parameters:

type Type of data to draw.
vertices A set of vertices.
indices An array of indices into the set of vertices.
indexCount The number of indices.

void TRenderer::SetTexture (TTextureRef *pTexture* = TTextureRef ())

Set the current rendering texture.

Parameters:

pTexture Texture to assign. Default parameter is NULL texture.

TTextureRef TRenderer::GetTexture ()

Get the currently assigned texture.

Returns:

The current texture.

void TRenderer::SetBlendMode (EBlendMode *blendMode*)

Set the current blend mode.

The blend mode is reset to "normal" when calling either Begin2d or Begin3d.

Parameters:

blendMode The new blend mode.

void TRenderer::SetFilteringMode (EFilteringMode *filteringMode*)

Set the current filtering mode to use when scaling images.

Parameters:

filteringMode Mode to use.

void TRenderer::SetTextureMapMode (ETextureMapMode *umap*, ETextureMapMode *vmap*)

Set the texture map mode for use when texture coordinates extend beyond the edge of the internal texture.

NOTE that the internal texture is almost always square and a power of two, so most textures should be limited to those constraints when using wrapping modes.

Parameters:

umap Texture map mode for U coordinates.
vmap Texture map mode for V coordinates.

void TRenderer::SetZBufferWrite (bool *writeToZbuffer*)

Enable or disable writing to zbuffer.

Parameters:

writeToZbuffer True to write to zbuffer.

See also:

SetZBufferTest

void TRenderer::SetZBufferTest (bool *testZbuffer*)

Enable zbuffer test.

Parameters:

testZbuffer True to test zbuffer when drawing.

See also:

SetZBufferWrite

void TRenderer::SetColorWrite (bool *colorWrite*)

Enable writing of RGB color data.

Default is on.

If you disable writing of RGB color data, no visible drawing will happen, though if zbuffer writing is still active drawing will still occur to that plane.

Parameters:

colorWrite True to enable writing.

bool TRenderer::GetColorWrite ()

Get the current "color write" state.

Returns:

True if color writing is enabled.

See also:

SetColorWrite

void TRenderer::ClearZBuffer ()

Clear the ZBuffer.

Done automatically by BeginDraw() and BeginRenderTarget().

bool TRenderer::GetZBufferWrite ()

Query ZBuffer-write state.

Returns:

True if ZBuffer writing is enabled; false otherwise.

See also:

SetZBufferWrite

bool TRenderer::GetZBufferTest ()

Query ZBuffer-test state.

Returns:

True if ZBuffer testing is enabled; false otherwise.

See also:

SetZBufferTest

str TRenderer::GetSystemData ()

Get data about the current system.

Returns:

A system data string.

bool TRenderer::GetTextureSquareFlag ()

Query whether textures on this computer will be created as square internally.

You can always create textures of any shape, but internally a texture may be extended to the next power-of-two, square size that will fit your requested texture size.

Note that texture dimensions will be rounded up to the next power-of-two on cards that require it; always query TTexture::GetInternalSize() if the exact texture dimensions are relevant.

Returns:

True if textures will always be square, internally.

bool TRenderer::BeginRenderTarget (TTextureRef *texture*, ERenderTargetMode *mode*)

Start rendering to a texture.

Must NOT be called during a Draw() update. After being called, all rendering commands without a texture target will be directed instead to the texture passed in to BeginRenderTarget().

If you are rendering to a texture that you will be completely covering with content, use one of the "full" render modes (kFullRenderRGB1 or kFullRenderRGBX). Note that in a full mode that if you don't cover the surface you may get garbage left over in the rendering buffer in any untouched areas of the image.

If your image already has content that you want to render on top of, use one of the merge modes.

If you want your resulting image to have an alpha component, you'll need to render it in two passes. This is because a lot of video cards don't have the ability to render alpha to a buffer–but they do have the ability to redirect alpha to the RGB channels. So that's what kMergeRenderXXXA does: It sets up the rendering pipeline to render just the alpha to the RGB channels, and then copies the resulting data back into your texture alpha channel. The usage pattern looks like this:

C++
```
TTextureRef myTexture = .... ; // Create your texture
TRenderer * r = TRenderer::GetInstance();
if (r->BeginRenderTarget(myTexture,TRenderer::kFullRenderRGBX))
{
      DoRendering();       // Paint the full texture
      r->EndRenderTarget();
}

if (r->BeginRenderTarget(myTexture,TRenderer::kMergeRenderXXXA))
{
      DoRendering();       // Paint the full texture again, this time
      r->EndRenderTarget(); // to get alpha information
}
```

Warning:

Will not render to areas of a surface beyond the extents of the screen surface (800x600 by default). This allows us to use the most compatible render-to-surface modes.

Additionally, by using the most compatible modes, this is not a fast operation. It is best used to pre-render an image that will be used multiple times rather than attempting to execute it every frame.

Parameters:

texture Texture to render to.
mode Render target mode.

Returns:

True on success.

void TRenderer::EndRenderTarget ()

Complete the rendering to a texture.

See also:

BeginRenderTarget

bool TRenderer::BeginDraw (bool *needRefresh*)

Open an internal draw context.

Called automatically by the system.

Typically internal use only. This function is called by the system prior to your TWindow::Draw() and TWindow::PostDraw() calls.

Parameters:

needRefresh True if we should refresh the screen (for support of dirty- rectangle drawing). Usually false, meaning you're redrawing the entire screen.

Returns:

True on success. If BeginDraw() returns false, do not call EndDraw().

void TRenderer::EndDraw (bool *flip* = `true`)

Close and complete an internal draw context.

Also presents the completed rendered screen.

Parameters:

flip Flip or blit the back buffer to the screen.

bool TRenderer::Begin2d ()

Begin 2d rendering.

Any calls to TTexture::Draw∗() functions should happen between a Begin2d() and End2d() call.

It's a good idea to minimize renderer state changes, as they can be expensive on some cards. Try to group most of your 2d rendering together in as few Begin2d() groups as possible.

Clears all 3d matrices when called. You can set the view and world matrix between Begin2d() and End2d(), but they will be cleared again on End2d().

You can use the helper class TBegin2d to automatically close the block on exiting the scope.

Returns:

True on success. False on failure, which can mean that you are already in a Begin2d() state, you are in a Begin3d() state, or some other aspect of the engine has gotten into a bad state. Check log file messages for details.

See also:

TBegin2d

Referenced by TBegin2d::TBegin2d().

void TRenderer::End2d ()

Finish a 2d rendering set.

See also:

Begin2d

Referenced by TBegin2d::Done().

bool TRenderer::Begin3d ()

Begin 3d rendering.

Any calls to TModel::Draw∗() functions should happen between a Begin3d() and End3d() call. Also, calls to DrawVertices() with vertex types TLitVert or TVert should be within a Begin3d() block.

It's a good idea to minimize renderer state changes, as they can be expensive on some cards. Try to group most of your 3d rendering together in as few Begin3d() groups as possible.

You can use the helper class TBegin3d to automatically close the block on exiting the scope.

Returns:

True on success. False on failure, which can mean that you are already in a Begin3d() state, you are in a Begin2d() state, or some other aspect of the engine has gotten into a bad state. Check log file messages for details.

See also:

TBegin3d

Referenced by TBegin3d::TBegin3d().

void TRenderer::End3d ()

Finish a 3d rendering set.

See also:

Begin3d

Referenced by TBegin3d::Done().

bool TRenderer::InDraw () const

Query as to whether we're currently in a drawing mode.

Returns:

True if we're between BeginDraw() and EndDraw(), or BeginRenderTarget() and EndRenderTarget().

void TRenderer::FillRect (const TURect & *rect*, const TColor & *color*, TTextureRef *dst* = TTextureRef ())

Fill a rectangle, optionally specifying a destination texture.

Unlike TTexture::Draw∗() calls, FillRect can be used to draw a rectangle in any texture type, and does not depend on the Begin2d/End2d state. **Will not blend** using the alpha value; instead it writes the alpha value as part of the color into the destination, when the destination texture supports alpha.

In the case of screen fills, the alpha value **is never used,** since to draw an alpha value to the screen is meaningless (the screen surface is never used as a source texture).

Parameters:

rect Rectangle to fill; this rectangle is either relative to the upper left corner of the current viewport or window (when dst is NULL), or is relative to the upper left corner of the texture.

color Color to draw–this color (RGBA) will be written verbatim into the surface across the rectangle with no blending, such that all pixels in the rectangle will exactly equal the RGBA color specified.

dst Optional destination texture. Current viewport and window coordinates are ignored when dst is non-NULL.

bool TRenderer::IsCapturePending ()

Query as to whether the previously set capture texture is still waiting to be filled with the screen render.

Returns:

True if the capture texture is still pending.

void TRenderer::SetCaptureTexture (TTextureRef *texture*)

Set a capture texture.

This texture will be filled with the results of the next screen update. Technically, the next EndDraw() will trigger a copy of the back-buffer surface to the given texture; EndDraw() typically is called at the end of a screen update by TWindowManager.

The capture texture MUST be exactly the size of the screen, or SetCaptureTexture() will silently fail (or ASSERT in debug builds).

Parameters:

texture Texture to fill.

TTextureRef TRenderer::GetCaptureTexture ()

Get the current capture texture from the renderer.

After a texture is filled, this function will return an empty texture.

Returns:

The texture that's waiting to be filled with the results of a screen render. An empty TTextureRef if SetCaptureTexture() hasn't been called, or after the texture has been filled.

void TRenderer::SetWorldMatrix (TMat4 * *pMatrix*)

Set the world matrix.

Playground uses a World/View/Projection transformation model. This method allows you to set the current world matrix.

Parameters:

pMatrix New world matrix.

void TRenderer::SetViewMatrix (TMat4 ∗ *pMatrix*)

Set the view matrix.

Playground uses a World/View/Projection transformation model. This method allows you to set the current view matrix.

Helper function SetView() can be used to set the view matrix based on a location, a look-at target, and an up vector.

Parameters:

pMatrix New view matrix.

void TRenderer::SetProjectionMatrix (TMat4 ∗ *pMatrix*)

Set the projection matrix.

Playground uses a World/View/Projection transformation model. This method allows you to set the current projection matrix.

Projection can be set up automatically using SetPerspectiveProjection.

See also:

SetPerspectiveProjection

Parameters:

pMatrix New projection matrix.

void TRenderer::SetView (const TVec3 & *eye*, const TVec3 & *at*, const TVec3 & *up*)

Set the view matrix based on the viewer location, a target location, and an up vector.

Parameters:

eye The viewer's location.
at The target location.
up Up vector.

void TRenderer::SetPerspectiveProjection (TReal *nearPlane*, TReal *farPlane*, TReal *fov* = `PI/4.0f`, TReal *aspect* = `0`)

Set a perspective projection matrix.

Parameters:

nearPlane Distance to near plane.
farPlane Distance to far plane.
fov Field of view in radians.
aspect Aspect ratio of view (0 to calculate the aspect ration from the viewport size).

void TRenderer::SetOrthogonalProjection (TReal *nearPlane*, TReal *farPlane*)

Set an orthogonal projection matrix.

Parameters:

nearPlane Distance to the near plane.
farPlane Distance to the far plane.

void TRenderer::GetWorldMatrix (TMat4 * *m*)

Get the current world matrix.

Parameters:

m A matrix to fill with the current world matrix.

void TRenderer::GetViewMatrix (TMat4 * *m*)

Get the current view matrix.

Parameters:

m A matrix to fill with the current view matrix.

void TRenderer::GetProjectionMatrix (TMat4 * *m*)

Get the current projection matrix.

Parameters:

m A matrix to fill with the current projection matrix.

void TRenderer::SetAmbientColor (const TColor & *color*)

Set the scene's ambient color.

Parameters:

color New ambient color.

void TRenderer::SetCullMode (ECullMode *cullMode*)

Set the current cull mode of the 3d render.

Parameters:

cullMode The new cull mode.

void TRenderer::SetMaterial (TMaterial * *mat*)

Set the current rendering material.

Parameters:

mat Material to set.

void TRenderer::SetLight (uint32_t *index*, TLight * *light*)

Set up one of the scene lights.

Parameters:

index Index to light to initialize.
light Light data.

bool TRenderer::ToggleHUD ()

Toggle the HUD.

Returns:

true if HUD was previously active.

bool TRenderer::In2d () const

Currently in Begin2d mode.

Returns:

True if in Begin2d mode.

bool TRenderer::In3d () const

Currently in Begin3d mode.

Returns:

True if in Begin3d mode.

void TRenderer::SetOption (str *option*, str *value*)

Set a renderer option.

Parameters:

option Option to set.
value Value to set.

str TRenderer::GetOption (str *option*)

Get a renderer option.

Parameters:

option Option to get.

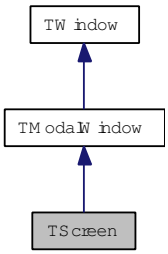
Returns:

Value.

15.54 TScreen Class Reference

`#include <pf/screen.h>`

Inheritance diagram for TScreen:



15.54.1 Detailed Description

The base level modal window.

Top-level application control logic can be handled at this level by deriving from TScreen and implementing a handler for DoModalProcess.

TScreen also handles caching of the window background, so TScreen::Draw should NOT be overridden.

Public Member Functions

- void ClearBackground (bool clear)

  Tell the screen to clear its background.

- void NeverClearBackground (bool neverClear)

  Tell the screen to never clear its backrgound.

- void SetBackgroundColor (const TColor &color)

  Set the background color.

- TColor GetBackgroundColor ()

  Get the current background color.

15.54.2 Member Function Documentation

void TScreen::ClearBackground (bool *clear*)

Tell the screen to clear its background.

This should only be done in a situation where you have a sparse set of windows covering the background.

Parameters:

clear True to clear the background.

void TScreen::NeverClearBackground (bool *neverClear*)

Tell the screen to never clear its backrgound.

This causes the screen to never redraw the background, even if is the only window.

Parameters:

neverClear True to never clear the background.

void TScreen::SetBackgroundColor (const TColor & *color*)

Set the background color.

Note this will only have an effect if ClearBackground() is set to true.

Parameters:

color Color to clear the background.

TColor TScreen::GetBackgroundColor ()

Get the current background color.

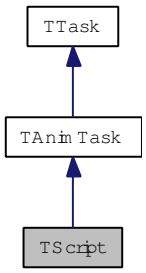
Returns:

The current background color.

15.55 TScript Class Reference

`#include <pf/script.h>`

Inheritance diagram for TScript:



15.55.1 Detailed Description

An encapsulation for a Lua script context.

A TScript can be used in multiple ways. If you create a thread that yields, then you can either use TScript::Resume to explicitly resume the thread, or you can register it as a TAnimTask with an appropriate dispatch (commonly TPlatform::AdoptTask or TModalWindow::AdoptTask). If you're calling functions that exit normally, then you don't need to do anything beyond calling the function or executing Lua code using DoLuaString(), RunScript(), or RunFunction().

The script syntax is Lua-standard. There are a few predefined functions that are useful to know:

- DebugOut() will print a string to the debug log.
- DumpTable() will dump a table recursively to the debug log.
- toparams() will take a table as a parameter and return the indexed members as multiple return values. You can use this to pass the members of a table to a vararg function, for instance.

See also:

Lua Scripting
http://www.lua.org

Global Script Settings

Set the path that all Lua scripts will search.

- void SetLuaPath (str luaPath)
  Set the global Lua search path.
- int32_t RunFunction (int32_t nargs=0, int32_t nresults=1)
  Run the function with parameters on the Lua stack.
- static str GetLuaPath ()
  Get the global Lua search path.

Public Member Functions

Construction/Destruction

- TScript (lua_State *luaState=NULL)
  Default Constructor.
- virtual ~TScript ()
  Destructor.

Lua Data Constructors

Functions that are used to push new data on the Lua stack.

- void SetGlobalLightUserData (const char *name, void *userData)
  Set a global variable to a Lua light userdata value.

Lua Data Accessors

Functions that are used to access parameters from the Lua table on the top of the stack.

- lua_State * GetState ()
  Get the current Lua state.
- TColor PopColor ()
  Pop a TColor from the stack.
- class TFont PopFont ()
  Pop a TFont from the stack.
- str PopString ()
  Pop a string from the stack.
- TRect PopRect ()
  Pop a TRect off the top of the stack.
- lua_Number PopNumber ()
  Pop a number from the stack.
- bool PopBool ()
  Pop a boolean from the stack.

Client Interface

Functions that are commonly used by a client.

- bool RunScript (str filename)
  Run a Lua script in the current environment.
- TScript * NewThread ()
  Create a new TScript-derived class of the same type as this class, but running in a new Lua thread.
- void DoLuaString (str luaCommand, int32_t length=-1, int nresults=0)
  Execute a string in the Lua interpreter.
- virtual bool InjectFunction ()

Inject a function into a running Lua script.

- int32_t Resume (int32_t narg=0)
  Resume a thread that has been suspended by a coroutine yield.
- bool CoroutineActive (int32_t narg=0)
  Test to see if a coroutine can currently be successfully resumed.

Lua Global Data Accessors

Functions that read global data from the Lua state.

These are used both inside a creator and globally.

- str GetGlobalString (str name)
  Get a string from a global Lua variable.
- str GetGlobalTableString (str name, int32_t index)
  Get a string from a global Lua table.
- TLuaTable * GetGlobalTable (str name)
  Get a global table from a Lua state.
- void SetGlobalString (str name, str value)
  Set a string global in a Lua environment.
- void SetGlobalNumber (str name, lua_Number value)
  Set a numeric global in a Lua environment.
- lua_Number GetGlobalNumber (str name)
  Get a number from a global Lua variable.

TTask Overrides

Functions that handle TTask behavior.

- virtual bool Animate ()
  This function is called when it's time to execute this task.

15.55.2 Constructor & Destructor Documentation

TScript::TScript (lua_State * *luaState* = `NULL`)

Default Constructor.

Parameters:

luaState This one is really for internal use only. TScript really only "plays well" with Lua scripts created by Playground.

15.55.3 Member Function Documentation

void TScript::SetGlobalLightUserData (const char * *name*, void * *userData*)

Set a global variable to a Lua light userdata value.

Parameters:

name Name of global.
userData Value of userdata.

TColor TScript::PopColor ()

Pop a TColor from the stack.

Returns:

A TColor from the top of the Lua stack.

class TFont TScript::PopFont ()

Pop a TFont from the stack.

Returns:

A TFont from the top of the Lua stack.

str TScript::PopString ()

Pop a string from the stack.

Returns:

A string from the top of the Lua stack.

TRect TScript::PopRect ()

Pop a TRect off the top of the stack.

Returns:

A TRect

lua_Number TScript::PopNumber ()

Pop a number from the stack.

Returns:

A number from the top of the Lua stack.

bool TScript::PopBool ()

Pop a boolean from the stack.

Returns:

A bool from the top of the Lua stack.

References ASSERT, str::c_str(), and ERROR_WRITE.

bool TScript::RunScript (str *filename*)

Run a Lua script in the current environment.

If the Lua script returns a value at the top level, that value is assumed to be a function that is to be run as a thread.

RunScript loads the script from the resource and then runs it using RunFunction. See RunFunction for important restrictions and limitations.

Parameters:

filename Filename to read

Returns:

true on success.

See also:

RunFunction

TScript* TScript::NewThread ()

Create a new TScript-derived class of the same type as this class, but running in a new Lua thread.

Lua threading is cooperative multithreading: It is non-preemptive, and as such requires that you "yield" to pause processing.

Returns:

A TScript-derived class constructed with similar parameters as the host class.

void TScript::DoLuaString (str *luaCommand*, int32_t *length* = `-1`, int *nresults* = `0`)

Execute a string in the Lua interpreter.

Is not executed as a thread, so will not interfere with an existing thread that has yielded.

Parameters:

luaCommand Command to execute.
length Optional length, for binary buffers.
nresults Optional number of results.

virtual bool TScript::InjectFunction () `[virtual]`

Inject a function into a running Lua script.

Default implementation is empty, but the TWindowManager::GetScript() script has a derived implementation.

Attempts to inject the function on the top of the Lua stack into the current coroutine.

The Lua function can take parameters, but is assumed to not return any results.

Returns:

True on success, false on failure.

See also:

TScript::RunScript

int32_t TScript::Resume (int32_t *narg* = 0)

Resume a thread that has been suspended by a coroutine yield.

Parameters:

narg Number of arguments being passed in on the stack; these arguments are returned as the results of the previous yield.

Returns:

0 if there are no errors running the coroutine, or an error code. See http://www.lua.org/manual/5.0/manual.html#lua_pcall for error codes.

bool TScript::CoroutineActive (int32_t *narg* = 0)

Test to see if a coroutine can currently be successfully resumed.

Parameters:

narg Number of arguments you were planning to pass in to the coroutine.

Returns:

True if a coroutine exists and is ready to be resumed. False otherwise.

str TScript::GetGlobalString (str *name*)

Get a string from a global Lua variable.

Parameters:

name Name of the Lua variable.

Returns:

The string, if found. An empty string otherwise.

str TScript::GetGlobalTableString (str *name*, int32_t *index*)

Get a string from a global Lua table.

Parameters:

name Name of the Lua table.

index Index of item to extract.

Returns:

The string, if table and index found. An empty string otherwise.

TLuaTable* TScript::GetGlobalTable (str *name*)

Get a global table from a Lua state.

The table is created with new, and must be deleted when you are done with it.

Parameters:

name Name of table to retrieve.

Returns:

A TLuaTable wrapping the global table.

void TScript::SetGlobalString (str *name*, str *value*)

Set a string global in a Lua environment.

Parameters:

name Name of the string variable.
value Value to set the string variable.

void TScript::SetGlobalNumber (str *name*, lua_Number *value*)

Set a numeric global in a Lua environment.

Parameters:

name Name of the lua_Number variable.
value Value to set the variable.

lua_Number TScript::GetGlobalNumber (str *name*)

Get a number from a global Lua variable.

Parameters:

name Name of the Lua variable.

Returns:

The corresponding number, or 0 if not found.

virtual bool TScript::Animate () `[virtual]`

This function is called when it's time to execute this task.

Returns:

True to keep the task alive. False to destroy the task.

Implements TAnimTask.

void TScript::SetLuaPath (str *luaPath*)

Set the global Lua search path.

Parameters:

luaPath New path for Lua.

int32_t TScript::RunFunction (int32_t *nargs* = 0, int32_t *nresults* = 1)

Run the function with parameters on the Lua stack.

This is different than the Lua APIs lua_call and lua_pcall in that it relies on InjectFunction to pass a function into a currently paused coroutine. InjectFunction is implemented in the TWindowManager script, but not in the base class. See Implementation Details below.

Example usage:

```
TScript * s = TWindowManager::GetInstance()->GetScript();
// Push the function on the stack.
lua_getglobal( s->GetState(), "MyLuaFunction" );
// Run the function with no parameters or results.
s->RunFunction(0,0);
```
C++

The above will call MyLuaFunction() in Lua with no parameters.

Example with parameters and a return value:

```
TScript * s = TWindowManager::GetInstance()->GetScript();
// Push the function on the stack.
lua_getglobal( s->GetState(), "StopGame" );
// Push the number 25 onto the stack.
lua_pushnumber( s->GetState(), 25 );
// Run the function with one parameter and one result.
s->RunFunction(1,1);
// Get the return value from the top of the stack.
int32_t result = (int32_t)lua_tonumber( s->GetState(), -1 );
```
C++

See Lua API documentation from http://www.lua.org for more information on lua_getglobal, lua_pushnumber, and the rest of the internal Lua API.

Warning:

If you need more than one return value from your function, it cannot yield control, nor call any function that yields control of the coroutine. A single return value is supported by the current API.

15.55.4 Details

When Lua has paused a script using the "coroutine.yield" function or the C call lua_yield, it remains in a suspended state until one calls lua_resume or coroutine.resume. In this state you can actually still use the same interpreter to execute Lua functions, but those functions may not themselves yield, nor can a Lua function that was called via C resume the previous Lua coroutine.

To allow arbitrary function execution from C, the Playground Lua "message loop" takes an extra parameter which is a function to call. In other words, whenever the Lua message loop has yielded to wait for a message, you can pass it in a message and/or a command to execute. The TWindowManager version of InjectFunction calls Resume with that command as a parameter, and it's executed as part of the main thread–so it can therefore enter its own wait loops, call other script functions, etc. However, that function that's passed in as a parameter can take no parameters of its own; RunFunction uses a Playground Lua call GetClosure to wrap your function plus any parameters in a Lua closure, and then passes that closure in to be executed.

In the case that a coroutine is not currently active, RunFunction does the trivial thing and calls lua_pcall with the standard error handler.

Note that the base class implementation of InjectFunction does nothing, and it's only the derived window UI script that InjectFunction will work.

This is not to hide the implementation, but instead because the injection relies on how the Lua script that yielded treats the return values of the yield statement. We can't make any assumptions about how your own custom scripts will process yield results; if you want to create your own version of InjectFunction, derive from TScript and add your own implementation. As an example:

C++
```
bool InjectFunction()
{
    return Resume(1)==0;
}
```

This would assume that your Lua code interpreted the first return value from yield as a function, and then ran the function:

Lua
```
    f = yield();
    if (f) then
       f();
    end
```

That way, the function is executed as part of your thread, rather than outside of it. If you need multiple threads, see TScript::NewThread().

Parameters:

nargs Number of arguments.
nresults Number of results. If there's a chance your function will be executed during a paused coroutine, *and* your function needs to be able to yield, then this number should be zero or one.

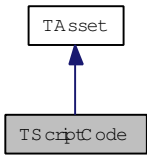
Returns:

0 on success. Lua error code otherwise.

15.56 TScriptCode Class Reference

`#include <pf/script.h>`

Inheritance diagram for TScriptCode:



15.56.1 Detailed Description

An encapsulation of a compiled Lua source file.

Public Member Functions

- str GetCode ()

  Get the (compiled) script code.

- uint32_t GetCodeLength ()

  Get the length of the compiled code block.

Static Public Member Functions

- static TScriptCodeRef Get (str handle, str luaPath="", str *error=NULL)

  Accessor function for Lua script-code asset.

15.56.2 Member Function Documentation

static TScriptCodeRef TScriptCode::Get (str *handle*, str *luaPath* = `""`, str * *error* = `NULL`) `[static]`

Accessor function for Lua script-code asset.

Internally, when loading a Lua script from a file, the internal routines use TScriptCode::Get() to load it. If you keep a reference to the TScriptCode, then when the routine calls TScriptCode::Get(), it will retrieve the cached (and precompiled) copy rather than reloading it from disk.

Parameters:

handle File handle to load.
luaPath Any additional Lua search path necessary.
error An optional str that gets set with an error string.

Returns:

A reference to the compiled code object.

str TScriptCode::GetCode ()

Get the (compiled) script code.

Returns:

A pre-compiled Lua block. Can have embedded null characters; use TScriptCode::GetCodeLength() to determine the length.

uint32_t TScriptCode::GetCodeLength ()

Get the length of the compiled code block.

Returns:

Code block length.

15.57 TSimpleHttp Class Reference

`#include <pf/simplehttp.h>`

15.57.1 Detailed Description

The TSimpleHttp class implements a basic HTTP connection.

Public Types

- enum ERequestFlags { eNoFlag = 0x00000000, eDoNotWriteCache = 0x00000001, eIgnoreCNInvalid = 0x00000002 }
  Request modifiers.
- enum EStatus { eNetWait, eNetDone, eNetError, eFileError }
  Status enumeration.

Public Member Functions

- TSimpleHttp ()
  Construction.
- virtual ~TSimpleHttp ()
  Destruction.
- void Init (const char *url, bool bPost=false, const char *path=NULL)
  Initialize TSimpleHttp.
- void AddArg (const char *name, const char *value)
  Add a POST or GET argument to the query.
- void AddArg (const char *name, int32_t value)
  Add a POST or GET argument to the query.
- void DoRequest (int32_t flags)
  Start the request.
- EStatus GetStatus ()
  Get the current status.
- unsigned long GetContentLengthHeader ()
  Returns the "content-length" field of the HTTP header, if available.
- unsigned long GetBytesReceived ()
  Get the total number of bytes received.

- char * GetContents ()

  Get the content of the reply.

Static Public Member Functions

- static str HttpSafeString (const char *unsafeString)

  HTTP cleanup.

15.57.2 Member Enumeration Documentation

enum TSimpleHttp::ERequestFlags

Request modifiers.

Enumerator:

eNoFlag No modifier.
eDoNotWriteCache Do not write the request to the cache.
eIgnoreCNInvalid Ignore an invalid certificate.

enum TSimpleHttp::EStatus

Status enumeration.

Enumerator:

eNetWait Waiting for a reply.
eNetDone Reply complete and successful.
eNetError Error communicating with server.
eFileError Error writing file.

15.57.3 Member Function Documentation

void TSimpleHttp::Init (const char * *url*, bool *bPost* = `false`, const char * *path* = `NULL`)

Initialize TSimpleHttp.

Parameters:

url The URL to connect with. The underlying library will not accept extremely long URLs, so use POST requests if the request will be more than about 800 characters long.
bPost True for a POST request; otherwise GET.
path Specify path name to download to; otherwise the data is kept in an internal buffer and retrieved using TSimpleHttp::GetContents().

void TSimpleHttp::AddArg (const char * *name*, const char * *value*)

Add a POST or GET argument to the query.

Parameters:

name Name of argument.
value Value of argument.

void TSimpleHttp::AddArg (const char * *name*, int32_t *value*)

Add a POST or GET argument to the query.

Parameters:

name Name of argument.
value Value of argument.

static str TSimpleHttp::HttpSafeString (const char * *unsafeString*) `[static]`

HTTP cleanup.

Parameters:

unsafeString String to clean.

Returns:

Newly cleaned string.

void TSimpleHttp::DoRequest (int32_t *flags*)

Start the request.

Parameters:

flags One or more ERequestFlags bitwise-ORed together.

EStatus TSimpleHttp::GetStatus ()

Get the current status.

Returns:

The status.

unsigned long TSimpleHttp::GetContentLengthHeader ()

Returns the "content-length" field of the HTTP header, if available.

This value is the expected total download size.

Returns:

Expected size of the content.

See also:

GetBytesReceived()

unsigned long TSimpleHttp::GetBytesReceived ()

Get the total number of bytes received.

If the download is complete, this is the same as GetContentLengthHeader(); if the download is still in progress, this will be a lower number that you can use to display a progress bar.

Returns:

Bytes received.

See also:

GetContentLengthHeader()

char* TSimpleHttp::GetContents ()

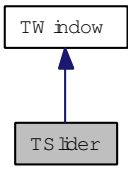
Get the content of the reply.

Available only if downloading to memory: Will fail until status is eNetDone.

15.58 TSlider Class Reference

`#include <pf/slider.h>`

Inheritance diagram for TSlider:



15.58.1 Detailed Description

Slider class.

A class that implements a UI slider that is scalable. It consists of two parts, the rail and the slider. The rail consists of three graphics, the top, middle, and bottom. The middle is stretched to be as long as it needs to match the height or width of the slider. You can set the width or height of the slider to be the long dimension, and the other dimension will be taken from the width of the rail images.

Attributes:

- railtop - Top of the rail. Rail graphics are vertical.
- railmid - Middle of the rail. This part gets stretched.
- railbot - Bottom of the rail.
- sliderimage - Image to use as the slider handle.
- sliderrollimage - Image to use as the rollover for the slider handle.
- yoffset - An offset to use to adjust the centering of the slider handle.

Public Types

- enum ESliderState { **eIdle** = 0, **eMoving** }
  The current state of the slider.

Public Member Functions

- void ShowHandle (bool show)
  Show the slider widget.
- virtual void Draw ()
  Draw the window.
- void SetRailTexture (TTextureRef top, TTextureRef mid, TTextureRef bot)
  Set the textures for the rail.

- void SetSliderTexture (TTextureRef texture, TTextureRef rollover)
  Set the textures for the slider knobs.
- TReal GetValue ()
  Get the current slider handle position.
- void SetValue (TReal value, bool silent=false)
  Set the current slider handle position.
- void SetScale (TReal scale)
  Set the slider scale.
- TReal GetScale ()
  Get the current slider scale.
- void SetAlpha (TReal alpha)
  Set the current slider alpha.
- TReal GetAlpha ()
  Get the current slider alpha.
- virtual bool OnMouseDown (const TPoint &point)
  Mouse handler.
- virtual bool OnMouseUp (const TPoint &point)
  Mouse handler.
- virtual bool OnMouseMove (const TPoint &point)
  Mouse handler.
- virtual bool OnMouseLeave ()
  Mouse handler.
- virtual void Init (TWindowStyle &style)
  Window initialization handler.
- ESliderState GetState ()
  Get the current slider state (whether it's moving or idle).

Static Public Member Functions

- static void Register ()
  Register TSlider with the windowing system.

15.58.2 Member Function Documentation

void TSlider::ShowHandle (bool *show*)

Show the slider widget.

Defaults to being visible.

Parameters:

show True to show handle.

virtual void TSlider::Draw () `[virtual]`

Draw the window.

Derived classes will override this function and provide the draw functionality.

Optionally only redraw portions that have been invalidated since the previous draw.

Reimplemented from TWindow.

void TSlider::SetRailTexture (TTextureRef *top*, TTextureRef *mid*, TTextureRef *bot*)

Set the textures for the rail.

The textures are oriented vertically, as if for a vertical slider, and will be rotated counter-clockwise for horizontal sliders.

To ensure compatiblity across renderers, the height each texture should be a power of two and greater than its width.

Parameters:

top Top of the slider.
mid Middle of the slider. This one gets stretched out to make the slider the right height or width.
bot Bottom of the slider.

void TSlider::SetSliderTexture (TTextureRef *texture*, TTextureRef *rollover*)

Set the textures for the slider knobs.

Parameters:

texture Normal state slider knob.
rollover Rollover state slider knob.

TReal TSlider::GetValue ()

Get the current slider handle position.

Returns:

The position as a value from 0 to 1.

void TSlider::SetValue (TReal *value*, bool *silent* = `false`)

Set the current slider handle position.

Parameters:

value Position from 0 to 1 inclusive.
silent True to prevent a "slider-changed" message from being sent.

void TSlider::SetScale (TReal *scale*)

Set the slider scale.

Parameters:

scale The scale of the slider.

TReal TSlider::GetScale ()

Get the current slider scale.

Returns:

Slider scale.

void TSlider::SetAlpha (TReal *alpha*)

Set the current slider alpha.

Parameters:

alpha Alpha of the slider.

TReal TSlider::GetAlpha ()

Get the current slider alpha.

Returns:

The slider alpha.

virtual bool TSlider::OnMouseDown (const TPoint & *point*) `[virtual]`

Mouse handler.

Parameters:

point Mouse position.

Returns:

True if handled.

Reimplemented from TWindow.

virtual bool TSlider::OnMouseUp (const TPoint & *point*) `[virtual]`

Mouse handler.

Parameters:

point Mouse position.

Returns:

True if handled.

Reimplemented from TWindow.

virtual bool TSlider::OnMouseMove (const TPoint & *point*) `[virtual]`

Mouse handler.

Parameters:

point Mouse position.

Returns:

True if handled.

Reimplemented from TWindow.

virtual bool TSlider::OnMouseLeave () `[virtual]`

Mouse handler.

Returns:

True if handled.

Reimplemented from TWindow.

virtual void TSlider::Init (TWindowStyle & *style*) `[virtual]`

Window initialization handler.

Parameters:

style Window style.

Reimplemented from TWindow.

ESliderState TSlider::GetState ()

Get the current slider state (whether it's moving or idle).

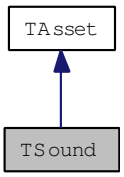
Returns:

The state of the slider.

15.59 TSound Class Reference

`#include <pf/sound.h>`

Inheritance diagram for TSound:



15.59.1 Detailed Description

The TSound class represents an object that can play a sound asset.

By default, TSound only knows how to load Ogg/Vorbis files, but Playground supports the loading of Ogg/Speex files as well. The Speex compression format is optimized for voice compression. For the fullest Playground support of Ogg/Speex files, you must use the encoder that ships with Playground, bin/speexenc.exe.

To activate Speex decoding, you need to add the following code to main.cpp:

```C++
#include <pf/speex.h>

...

// In PlaygroundInit()
TSpeex::Register();
```

This will cause TSound::Get() to handle .spx files correctly. Speex files *only* support streamed playback, and do not support seeking to an arbitrary offset. You can, however, pause, resume, and reset a Speex sound to the beginning, and you can loop a Speex sound.

Public Types

- enum { **kButtonSound** = 0, **kUserSoundBase** = 1000 }

Public Member Functions

- TSoundInstanceRef Play (bool bLoop=false)

  Play a sound.

- bool Pause (bool bPause)

  Pause all instances of this sound.

- bool Kill ()

  Kill all instances of this sound - once a sound is killed it must be restarted with Play(), it cannot be unpaused.

- TReal GetSoundLength ()

  Get the length of the sound in seconds.

- TSoundRef GetRef ()

  Get a reference to this sound.

- str GetName ()

  Get the file handle used to create the sound.

Static Public Member Functions

Factory Methods

- static TSoundRef Get (str filename, bool bInMemory=true, int32_t type=-1)

  Get a sound from a name, associating it with a particular sound group.

15.59.2 Member Function Documentation

static TSoundRef TSound::Get (str *filename*, bool *bInMemory* = `true`, int32_t *type* = `-1`) `[static]`

Get a sound from a name, associating it with a particular sound group.

You can also specify the sound group by appending "?group=###" to the filename instead of passing the group number as the third parameter. This allows you to specify a group number when referencing a sound in Lua (in the TButton sound tags, for example), or when add a group of files using TAssetMap::AddAssets().

Parameters:

filename Name of the sound file.
bInMemory True to load entire sound into memory, false to stream it
type Type of sound, or -1 to be part of a global group. Sounds with the same type will all play at the same volume. User defined groups should start at kUserSoundBase. Anything below that value is reserved for internal library use.

Returns:

A TSoundRef to the sound.

TSoundInstanceRef TSound::Play (bool *bLoop* = `false`)

Play a sound.

Parameters:

bLoop Whether or not to loop a sound

Returns:

A reference to the spawned instance of the sound. To modify or update that instance once it's spawned, keep a reference to the TSoundInstance and use it to modify the instance.

bool TSound::Pause (bool *bPause*)

Pause all instances of this sound.

Parameters:

bPause True to pause, false to unpause

Returns:

true on success, false on failure

bool TSound::Kill ()

Kill all instances of this sound - once a sound is killed it must be restarted with Play(), it cannot be unpaused. If a sound is set as a "continuation" of a sound that is killed, it will also be killed, even if it hasn't started yet.

Returns:

true if any sounds were killed, false otherwise.

TReal TSound::GetSoundLength ()

Get the length of the sound in seconds.

Returns:

Number of seconds.

TSoundRef TSound::GetRef ()

Get a reference to this sound.

Returns:

A reference to this.

Reimplemented from TAsset.

str TSound::GetName ()

Get the file handle used to create the sound.

Returns:

The sound file handle.

15.60 TSoundCallBack Class Reference

`#include <pf/sound.h>`

15.60.1 Detailed Description

TSoundCallBack –a class that you override and attach to a TSound if you want to know when the sound has finished playing.

Public Member Functions

- TSoundCallBack ()
  Constructor.
- virtual ~TSoundCallBack ()
  Destructor.
- virtual void OnComplete (TSoundInstanceRef soundInstance, TSoundRef nextSound)=0
  Function called when a sound has finished playing.

15.60.2 Member Function Documentation

virtual void TSoundCallBack::OnComplete (TSoundInstanceRef *soundInstance*, TSoundRef *nextSound*) `[pure virtual]`

Function called when a sound has finished playing.

This function is guaranteed to be called during the main thread, but may be called up to one second later due to buffering.

Parameters:

soundInstance TSoundInstanceRef of sound that just finished
nextSound TSoundRef of sound that has been queued up to play next with TSound::SetCompleteAction

15.61 TSoundInstance Class Reference

`#include <pf/soundinstance.h>`

15.61.1 Detailed Description

An instance of a sound.

Returned as a TSoundInstanceRef from TSound::Play(), you can then control the playback of that instance using this class.

See also:

TSound

Public Member Functions

- virtual ~TSoundInstance ()

  Destructor.

- void Play ()

  Play the sound!

- void Pause (bool bPause)

  Pause the sound!

- void Kill ()

  Kill the sound instance.

- void SetPosition (TReal seconds)

  Set a sound to a specific play position.

- void SetVolume (float volume)

  Set the volume of the sound.

- int GetGroupId ()

  Get the group this stream belongs to.

- void SetCompleteAction (TSoundCallBack *pCallback, TSoundRef playNext=TSoundRef())

  Setup a sound callback - the sound will call the passed in callback class when the sound has finished playing.

- TSoundRef GetSound ()

  Get a reference to the original sound that spawned this instance.

- TSoundRef GetNextSound ()

  Get a reference to the next sound to be streamed after the current one completes.

15.61.2 Member Function Documentation

void TSoundInstance::Pause (bool *bPause*)

Pause the sound!

Parameters:

bPause True to pause the sound; false to resume.

void TSoundInstance::Kill ()

Kill the sound instance.

Sound instances normally live until they complete or until they are killed. Simply releasing the TSoundInstanceRef will not itself kill a sound.

void TSoundInstance::SetPosition (TReal *seconds*)

Set a sound to a specific play position.

Parameters:

seconds The position, in seconds, to set the play position of the sound.

void TSoundInstance::SetVolume (float *volume*)

Set the volume of the sound.

Parameters:

volume Volume level 0.0-1.0f

int TSoundInstance::GetGroupId ()

Get the group this stream belongs to.

Returns:

The current group id.

void TSoundInstance::SetCompleteAction (TSoundCallBack * *pCallback*, TSoundRef *playNext* = TSoundRef ())

Setup a sound callback - the sound will call the passed in callback class when the sound has finished playing.

The client maintains ownership of the callback, and it is their responsibility to delete it when no longer needed. Deleting the callback automatically unregisters it. A TSoundCallBack contains references to the current sound and the playNext sound. Therefore, you do not need to keep a reference around to a sound used in a sound callback, and the sound can be killed by deleting the sound callback (However - in practice it is wise to keep around your own sound reference so you can pause it, kill it, etc.) This call can also be used to setup a sound to

play immediately after this sound is done. Note that because a callback is called during the main thread, in order to play a sound seamlessly, it is better to setup a 2nd sound with playNext instead of having the pCallback play another sound.

Warning:

When queueing up sounds with the next parameter, the sound must be of the same number of channels (i.e. mono or stereo).

If you queue up a sound with a different number of channels, an ASSERT will trigger in a debug build. In a release build, the resulting sound will likely be incorrect.

Parameters:

pCallback Pointer to callback object that will be called when the sound is done. This parameter can be NULL if the client just wants to setup a playNext sound.

playNext - what sound to play after the current sound finishes. Default is a NULL TSoundRef(). (See note above about playNext restrictions)

TSoundRef TSoundInstance::GetSound ()

Get a reference to the original sound that spawned this instance.

Returns:

A sound reference, or NULL if the sound has been deleted.

15.62 TSoundManager Class Reference

`#include <pf/soundmanager.h>`

15.62.1 Detailed Description

The TSoundManager class controls access to the sound subsystem.

Public Member Functions

- void SetVolume (float volume)

  Set the global sound volume.

- void SetTypeVolume (int32_t type, float volume)

  Set the volume for a specific group of sounds.

- void PauseAllSounds (bool bPause, int32_t type=-1)

  Pause or unpause all sounds.

- void KillAllSounds (int32_t type=-1)

  Kill all sounds.

15.62.2 Member Function Documentation

void TSoundManager::SetVolume (float *volume*)

Set the global sound volume.

Parameters:

volume Global sound volume. 0.0 is off, 1.0 is full volume.

void TSoundManager::SetTypeVolume (int32_t *type*, float *volume*)

Set the volume for a specific group of sounds.

This is multiplied with the global volume setting.

Parameters:

type id of sound group to set.
volume Volume to set. 0.0 is off, 1.0 is full volume.

void TSoundManager::PauseAllSounds (bool *bPause*, int32_t *type* = −1)

Pause or unpause all sounds.

Parameters:

bPause true to pause, false to resume.
type Pass -1 to pause all sounds, or specify a specific sound group ID

void TSoundManager::KillAllSounds (int32_t *type* = −1)

Kill all sounds.

Parameters:

type Pass -1 to kill all sounds, or specify a specific sound group ID.

15.63 TSpeex Class Reference

`#include <pf/speex.h>`

15.63.1 Detailed Description

Interface for class TSpeex.

A class that wraps the registration of Speex support with the sound engine. Once you call TSpeex::Register(), TSound::Get() will properly load a TSound object that streams an Ogg/Speex file, assuming the file has the expected .spx extension.

See more important information in the TSound documentation.

Static Public Member Functions

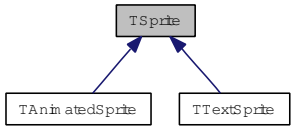
- static void Register ()

  Register the Speex decoder with the sound engine.

15.64 TSprite Class Reference

`#include <pf/sprite.h>`

Inheritance diagram for TSprite:



15.64.1 Detailed Description

A 2d sprite object.

A TSprite functions both as an individual display object and a container for child TSprite objects. Typically a TSprite with no TTexture is used as a container for all of the TSprites in one screen of a game.

How to Position a Sprite

Using TSprite::GetDrawSpec(), you retreive a TDrawSpec class that contains an mMatrix that governs how this sprite is drawn. It contains a position (mMatrix[2]), and a rotation/scale (see TMat3 for more information). The TDrawSpec also allows you to tint a sprite, flip it in the x or y axis, or change the logical center (the point it scales from and rotates around).

See the TDrawSpec documentation for more information.

TSprite References

> A TSprite is always stored in the client as a TSpriteRef—it's a reference counted object. To destroy a TSprite, simply call reset() on the last TSpriteRef that refers to that sprite.

It's worth noting that a TSprite holds a reference to any of its children, so you don't need to worry about keeping an external reference to a child sprite that can be passively attached to an object. An example of this usage would be a shadow that stays at a constant offset from the parent sprite. To make the shadow appear behind the parent you can give it a negative "layer" when you create it.

Design Concerns

> Because it's convenient, many people tend to want to use TSprite as a game object. It's not designed that way, however, and you'll quickly run into limitations if you try. For instance, there's no way to enumerate children of a TSprite. There's also no way to name a TSprite, and then look up a child by name.

This is intentional. TSprite was designed as a very light class, and adding strings and the ability to iterate would weigh it down.

Another option you might consider is deriving your own class from TSprite. It has been requested by developers so often, that it's used as an example in the docs. I've come to believe that creating a game object that way is categorically a bad idea. As Playground has grown, new child classes of TSprite have been created, so if you derive a game object from TSprite, you won't be able to have a game object that is, say, a TFxSprite or a TTextSprite. Aside from that, a TSprite really is all about rendering a TTexture, and mixing game code with your rendering code is poor design when it can be avoided.

If your game objects exist in their own hierarchy, however, and they just own a reference to the TSprite that

represents them in the game, then your game objects can be represented by whatever TSprite-derived object we (or you) devise, and the design of your game will now be cleaner, since the details of the rendering of your game objects will be properly encapsulated in its own class and not mixed willy-nilly in the game object behavior classes.

So that's why you should create your own hierarchy of game objects distinct from the TSprite hierarchy. Having two parallel hierarchies is not ideal, but weighing down TSprite with the plumbing required to promote it to a full-fledged game object is even less desirable.

How Sprites are Drawn

A TSprite has a concept of a layer which indicates how it will be rendered relative to its siblings. For the top sprite in a hierarchy (the one you're calling Draw on yourself), the layer parameter is meaningless—it only applies to sprites with siblings or a parent. The layer determines the relative order of drawing of a sprite with its siblings and parent.

Here are the rules that determine the order of drawing:

- When siblings have the same layer, they will render in an arbitrary order—it is assumed not to matter what order they're in.
- When siblings have different layers, the higher layered siblings will be rendered above the lower layered siblings.
- When a child sprite has a negative layer, it will be rendered *behind* its parent; otherwise it will be rendered in front of its parent.

Changing the layer of a sprite requires that the parent's child list be resorted. In other words, it's a relatively heavy operation if there are a lot of siblings, so try not to do it frequently.

See also:

TAnimatedSprite

Initialization/Destruction

- static TSpriteRef Create (int32_t layer=0, TTextureRef texture=TTextureRef())

  Allocation of a new sprite.

- virtual ~TSprite ()

  Destructor.

Public Types

- typedef std::list< TSpriteRef > SpriteList

  A list of sprites.

Public Member Functions

- virtual void SetTexture (TTextureRef texture)

  Set the texture of the sprite object.

- TTextureRef GetTexture ()

Get the current texture.

- TDrawSpec & GetDrawSpec ()

  Get the associated TDrawSpec.

- const TDrawSpec & GetDrawSpec () const

  Get a const reference to the associated TDrawSpec.

- void SetVisible (bool visible)

  Set a sprite to be visible and enabled.

- bool IsVisible ()

  Is the sprite visible/enabled?

- TSprite * GetParent ()

  Get the current parent of this sprite.

Drawing and Layers

- virtual void Draw (const TDrawSpec &environmentSpec=TDrawSpec(), int32_t depth=-1)

  Draw the sprite and its children.

- void SetLayer (int32_t layer)

  Set the layer of the sprite.

- int32_t GetLayer ()

  Get the current sprite layer.

Parent/Child Access and Management

- void AddChild (TSpriteRef child)

  Add a child sprite.

- virtual bool RemoveChild (TSpriteRef sprite)

  Remove a child sprite.

- void RemoveChildren ()

  Release all children of the sprite.

Bounding Rectangles and Hit Tests.

- virtual TRect GetRect (const TDrawSpec &parentContext, int32_t depth=-1)

  Get the bounding rectangle of this sprite.

- virtual bool HitTest (const TPoint &at, const TDrawSpec &parentContext, int32_t opacity=-1, int32_t depth=-1)

  Test to see if a point is within our sprite.

Protected Member Functions

- TSprite (int32_t layer)

  Internal Constructor. Use TSprite::Create() to get a new sprite.

- virtual void ResortSprites ()

  Resort my children because one of them has changed layer (priority).

Protected Attributes

- SpriteList mChildren

  Sprite children.

Friends

- bool operator< (TSpriteRef first, TSpriteRef second)

  Comparison operator for layer sorting.

15.64.2 Member Function Documentation

static TSpriteRef TSprite::Create (int32_t *layer* = `0`, TTextureRef *texture* = TTextureRef`()`) `[static]`

Allocation of a new sprite.

Construction is restricted to help "encourage" the use of TSpriteRefs to hold your TSprites (as well as to encapsulate the TSpriteRef creation pattern).

Parameters:

layer Initial sprite layer.
texture Initial sprite texture.

Returns:

A newly allocated TSprite wrapped in a TSpriteRef.

See also:

SetLayer
SetTexture

virtual void TSprite::Draw (const TDrawSpec & *environmentSpec* = TDrawSpec`()`, int32_t *depth* = `-1`) `[virtual]`

Draw the sprite and its children.

A common mistake is to assume that the incoming TDrawSpec here is what you use to position and configure this sprite. The way to position a sprite is to modify its internal TDrawSpec, which you retrieve with GetDrawSpec(). The environment parameter is combined with the local TDrawSpec to determine the actual position of the sprite,

though it only inherits those features marked in TDrawSpec as inheritable. See TDrawSpec::GetRelative() for more details.

Parameters:

environmentSpec The 'parent' or environment drawspec—the frame of reference that this sprite is to be rendered in. In general you shouldn't pass anything in for this parameter and instead should modify the position of the sprite using its GetDrawSpec() member.

Defaults to a default-constructed TDrawSpec. See TDrawSpec for more details on what is inherited.

Parameters:

depth How many generations of children to draw; -1 means all children.

See also:

TDrawSpec

Reimplemented in TAnimatedSprite, and TTextSprite.

void TSprite::SetLayer (int32_t *layer*)

Set the layer of the sprite.

A relatively expensive call; use with care.

The sprite's layer determines the order in which it will be rendered relative to its immediate parent and siblings. A negative layer will be rendered behind its parent, while a positive layer will be rendered in front. Higher layer numbers are rendered in front of lower layer numbers.

Parameters:

layer New sprite layer.

void TSprite::AddChild (TSpriteRef *child*)

Add a child sprite.

Children are drawn relative to the parent sprite: When you set the position, it will be added to the position of the parent.

The child should *not* be added independently to the sprite manager.

Children with a negative sprite layer are drawn behind the parent sprite. Zero or positive layers are drawn after the parent sprite.

Parameters:

child The sprite to add as a child of this sprite.

virtual bool TSprite::RemoveChild (TSpriteRef *sprite*) `[virtual]`

Remove a child sprite.

Returns true if child found.

Parameters:

sprite Child to remove.

Returns:

True if child found.

void TSprite::RemoveChildren ()

Release all children of the sprite.

"If you love them, set them free."

virtual TRect TSprite::GetRect (const TDrawSpec & *parentContext*, int32_t *depth* = −1) [virtual]

Get the bounding rectangle of this sprite.

Parameters:

parentContext The parent context to test within—where is this sprite being drawn, and with what matrix? Alpha and color information is ignored.
depth Depth of children to test

Returns:

Rectangle that includes this sprite.

Reimplemented in TAnimatedSprite, and TTextSprite.

virtual bool TSprite::HitTest (const TPoint & *at*, const TDrawSpec & *parentContext*, int32_t *opacity* = −1, int32_t *depth* = −1) [virtual]

Test to see if a point is within our sprite.

Will test children as well, unless depth is set to zero.

By default will *not* test children unless the TPlatform setting TPlatform::kSpriteAlwaysTestChildren is enabled with TPlatform::SetConfig().

Parameters:

at Point to test.
parentContext The parent context to test within—where is this sprite being drawn, and with what matrix? Alpha and color information is ignored.
opacity Level of opacity to test for; -1 for a simple bounding box test, or 0-255 for alpha color value, where 0 is transparent (and will therefore always succeed).
depth Depth of children to test. Zero means only test this sprite. -1 means recurse children as deep as they go.

Returns:

true if point hits us.

Reimplemented in TAnimatedSprite, and TTextSprite.

virtual void TSprite::SetTexture (TTextureRef *texture*) [virtual]

Set the texture of the sprite object.

Parameters:

texture Texture to use.

Reimplemented in TAnimatedSprite.

TTextureRef TSprite::GetTexture ()

Get the current texture.

Returns:

A reference to the current texture.

TDrawSpec& TSprite::GetDrawSpec ()

Get the associated TDrawSpec.

Returns:

A modifiable reference to the TDrawSpec associated with this sprite.

const TDrawSpec& TSprite::GetDrawSpec () const

Get a const reference to the associated TDrawSpec.

Returns:

A non-modifiable reference to the TDrawSpec associated with this sprite.

void TSprite::SetVisible (bool *visible*)

Set a sprite to be visible and enabled.

Sprites are initially visible.

Parameters:

visible True to set visible.

bool TSprite::IsVisible ()

Is the sprite visible/enabled?

Returns:

True if it's enabled.

TSprite* TSprite::GetParent ()

Get the current parent of this sprite.

Returns:

A pointer to the current parent, or NULL if this sprite is free.

15.64.3 Friends And Related Function Documentation

bool operator< (TSpriteRef *first*, TSpriteRef *second*) `[friend]`

Comparison operator for layer sorting.

Parameters:

first Left hand item to compare.
second Right hand item to compare.

Returns:

True if first sprite has a lower layer than the second.

15.65 TStringTable Class Reference

`#include <pf/stringtable.h>`

15.65.1 Detailed Description

The interface class for a string table.

The global string table contains a mapping of string identifiers to localized strings. When each string is requested at run time using GetString(), the parameters passed into GetString() are substituted into slots specified by %1%, %2%, %3%... etc.

You can also use 's to reference other lookups, such as "Today's date is %date%", where date is another string to look up in the table.

To create "%" in a string, use "%%".

Each string supports up to 9 parameters.

A string can contain formatting information as described in the documentation for TTextGraphic.

See Translation Issues and the String Table for a general description of the string table requirements.

Public Member Functions

- str GetString (str id, str param1=str(), str param2=str(), str param3=str(), str param4=str(), str param5=str())

  Get a string - get a string from the string table, filling in the string with passed in strings.

- str GetString (str id, uint32_t numStr, str *strArray)

  Get a string - get a string from the string table, filling in the string with passed in array of strings.

- bool SetSilent (bool silent)

  Set the string table to not write errors for missing strings.

15.65.2 Member Function Documentation

str TStringTable::GetString (str *id*, str *param1* = str(), str *param2* = str(), str *param3* = str(), str *param4* = str(), str *param5* = str())

Get a string - get a string from the string table, filling in the string with passed in strings.

Parameters:

id id of string to look up in the string table.

param1-param5 Optional strings used to fill in the placeholders in the string returned from the string table. If you want a parameter to be looked up in the string table as well, enclose it with %param% to signify that it should be looked up in the string table (use "%%" to just output a "%").

Returns:

a formatted str that is the result of looking up the id in the string table and adding in all the optional parameters. If any of the ids do not exist in the str table, the string returned will be "#####" to signify an invalid lookup.

str TStringTable::GetString (str *id*, uint32_t *numStr*, str * *strArray*)

Get a string - get a string from the string table, filling in the string with passed in array of strings.

Parameters:

id id of string to look up in the string table.
numStr number of Str in the str array
strArray Pointer to array of strings used to fill in the placeholders in the string returned from the string table. If you want a parameter to be looked up in the string table as well, enclose it with %param% to signify that it should be looked up in the string table (use "%%" to just output a "%").

Returns:

a formatted str that is the result of looking up the id in the string table and adding in all the optional parameters. If any of the ids do not exist in the str table, the string returned will be "#####" to signify an invalid lookup.

bool TStringTable::SetSilent (bool *silent*)

Set the string table to not write errors for missing strings.

Parameters:

silent True to suppress errors.

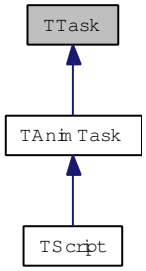
Returns:

Old value of silent.

15.66 TTask Class Reference

`#include <pf/task.h>`

Inheritance diagram for TTask:



15.66.1 Detailed Description

The task interface.

Used as a "callback" for events and periodic tasks.

Public Types

- enum ETaskContext { eNormal, eOnDraw }

  Task context.

Public Member Functions

- virtual ~TTask ()

  Virtual Destructor.

- virtual bool Ready (ETaskContext context=eNormal)

  Is this task ready?

- virtual bool DoTask ()=0

  This function is called when it's time to execute this task.

15.66.2 Member Enumeration Documentation

enum TTask::ETaskContext

Task context.

Enumerator:

eNormal Normal update context.
eOnDraw Immediately prior to the next screen draw context.

15.66.3 Member Function Documentation

virtual bool TTask::Ready (ETaskContext *context* = `eNormal`) `[virtual]`

Is this task ready?

A virtual function that returns whether this task is ready to be executed. Derived classes should override this function to provide more control over when a task is executed.

Parameters:

context The context in which we're being called.

Returns:

eReady if it's ready, eNotReady to wait, or eOnDraw to execute prior to the next screen draw. Default implementation returns eReady.

virtual bool TTask::DoTask () `[pure virtual]`

This function is called when it's time to execute this task.

Returns:

True to keep the task alive. False to destroy the task.

15.67 TTaskList Class Reference

`#include <pf/tasklist.h>`

15.67.1 Detailed Description

A list of TTask-derived objects.

Public Types

- typedef std::list< TTask * > TaskList
  The internal task list type.

Public Member Functions

- ~TTaskList ()
  Destructor.
- bool OrphanTask (TTask *task)
  Remove a task from the task list.
- void AdoptTask (TTask *task)
  Add a task to the task list.
- void DoAll (TTask::ETaskContext context=TTask::eNormal)
  Perform all of the tasks in the task list.
- void DestroyAll ()
  Destroy all the tasks in the list.

15.67.2 Member Function Documentation

bool TTaskList::OrphanTask (TTask * *task*)

Remove a task from the task list.

Does not delete the task, but rather releases ownership; calling function now owns task.

Parameters:

task Task to remove.

Returns:

true if task was removed, false if task was not found

void TTaskList::AdoptTask (TTask * *task*)

Add a task to the task list.

Task list takes ownership of the task and will delete it when the task notifies that it is complete.

Parameters:

task Task to add.

void TTaskList::DoAll (TTask::ETaskContext *context* = `TTask::eNormal`)

Perform all of the tasks in the task list.

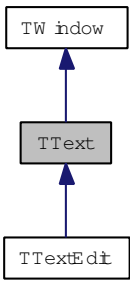
Parameters:

context A calling context, if any. Used to trigger tasks when drawing, for example.

15.68 TText Class Reference

`#include <pf/text.h>`

Inheritance diagram for TText:



15.68.1 Detailed Description

A text window.

Contains a TTextGraphic and renders it transparently.

See also:

TTextGraphic

Public Types

- enum EFlags {

  kHAlignLeft = TTextGraphic::kHAlignLeft, kHAlignCenter = TTextGraphic::kHAlignCenter, kHAlignRight = TTextGraphic::kHAlignRight, kVAlignTop = TTextGraphic::kVAlignTop,

  kVAlignCenter = TTextGraphic::kVAlignCenter, kVAlignBottom = TTextGraphic::kVAlignBottom }

  TText window child flags.

Public Member Functions

- TText (bool staticText=false)

  Constructor.

- ~TText ()

  Destructor.

- bool Create (str text, uint32_t w, uint32_t h, uint32_t flags, const char *fontFilename, uint32_t lineHeight, const TColor &textColor)

  Create a TText window.

- virtual void Draw ()

  TWindow::Draw handler.

- uint32_t GetLineCount ()
  Get the number of lines in the text output.
- void SetStartLine (TReal startLine=0)
  Set the first line in the text output.
- void GetTextBounds (TRect *pBounds)
  Get the actual boundary of the rendered text.
- virtual void SetText (str text)
  Set the current text content.
- virtual str GetText ()
  Get the current text content.
- TTextGraphic * GetTextGraphic ()
  Get the associated TTextGraphic object.
- void SetColor (const TColor &color)
  Set the current text color.
- void SetAlpha (TReal alpha)
  Set the text alpha.
- void SetScale (TReal scale)
  Set the current text scale.
- void SetLinePadding (int32_t linePadding)
  Set the current line padding.
- virtual bool OnMouseUp (const TPoint &point)
  Mouse up handler.
- virtual bool OnMouseDown (const TPoint &point)
  Mouse down handler.
- virtual bool OnMouseMove (const TPoint &point)
  Mouse motion handler.
- virtual bool OnMouseLeave ()
  Notification that the mouse has left the window.
- void SetRotation (TReal degrees, int32_t originX, int32_t originY)
  Change the rotation - default is 0 degrees, 0,0 origin.
- uint32_t GetMaxScroll ()
  Get the maximum line you need to scroll the text to in order to display the last line of text.
- virtual void SetScroll (TReal vScroll, TReal hScroll=0)

A virtual function to override if your window can scroll.

- void SetScrollPadding (TReal pad)

  This value controls how far past (in lines) the end of the text the scroll can go.

- virtual void Init (TWindowStyle &style)

  Initialize the Window.

15.68.2 Member Enumeration Documentation

enum TText::EFlags

TText window child flags.

Enumerator:

kHAlignLeft Align horizontally with the left edge.
kHAlignCenter Align horizontally with the center.
kHAlignRight Align horizontally with the right edge.
kVAlignTop Align vertically with the top.
kVAlignCenter Align vertically with the center.
kVAlignBottom Align vertically with the bottom.

15.68.3 Constructor & Destructor Documentation

TText::TText (bool *staticText* = `false`)

Constructor.

Parameters:

staticText True if this is a static text field.

15.68.4 Member Function Documentation

bool TText::Create (str *text*, uint32_t *w*, uint32_t *h*, uint32_t *flags*, const char * *fontFilename*, uint32_t *lineHeight*, const TColor & *textColor*)

Create a TText window.

Parameters:

text Initial text for window.
w Width of client rectangle.
h Height of client rectangle.
flags Flags from TText::EFlags
fontFilename Font name.
lineHeight Font size.
textColor Font color.

Returns:

True on success.

uint32_t TText::GetLineCount ()

Get the number of lines in the text output.

Returns:

Number of lines.

void TText::SetStartLine (TReal *startLine* = 0)

Set the first line in the text output.

Lines are 0 -> linecount-1.

Parameters:

startLine Line to display as the first line of text.

void TText::GetTextBounds (TRect * *pBounds*)

Get the actual boundary of the rendered text.

Parameters:

pBounds A rectangle in client coordinates.

virtual void TText::SetText (str *text*) [virtual]

Set the current text content.

Parameters:

text Text to set.

Reimplemented in TTextEdit.

virtual str TText::GetText () [virtual]

Get the current text content.

Returns:

A string containing the current text.

Reimplemented in TTextEdit.

TTextGraphic* TText::GetTextGraphic ()

Get the associated TTextGraphic object.

Returns:

The TTextGraphic that draws this window's text.

void TText::SetColor (const TColor & *color*)

Set the current text color.

Parameters:

color New text color.

void TText::SetAlpha (TReal *alpha*)

Set the text alpha.

Parameters:

alpha Opacity of text. 1.0==opaque. Multiplied by alpha component of color.

void TText::SetScale (TReal *scale*)

Set the current text scale.

Default is 1.0. Useful for zooming effects.

Parameters:

scale New text scale.

void TText::SetLinePadding (int32_t *linePadding*)

Set the current line padding.

Default is 0. Can be positive or negative - extends/compresses a fonts natural line spacing

Parameters:

linePadding New Line Padding.

virtual bool TText::OnMouseUp (const TPoint & *point*) `[virtual]`

Mouse up handler.

Used to detect clicks on embedded text links. Returns false if no text link was previously clicked on.

Parameters:

point Location of mouse release in client coordinates.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented from TWindow.

virtual bool TText::OnMouseDown (const TPoint & *point*) [virtual]

Mouse down handler.

Used to detect clicks on embedded text links. Returns false if no text link is found.

Parameters:

point Location of mouse press in client coordinates.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented from TWindow.

virtual bool TText::OnMouseMove (const TPoint & *point*) [virtual]

Mouse motion handler.

Parameters:

point Location of mouse in client coordinates.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented from TWindow.

virtual bool TText::OnMouseLeave () [virtual]

Notification that the mouse has left the window.

Warning:

This message is only sent if TWindowManager::AddMouseListener() has been called for this window previously.

Returns:

True if handled.

Reimplemented from TWindow.

void TText::SetRotation (TReal *degrees*, int32_t *originX*, int32_t *originY*)

Change the rotation - default is 0 degrees, 0,0 origin.

Parameters:

degrees rotation angle in degress (not radians because degress are friendlier)
originX offset from left of text rect to use as center point of rotation
originY offset from top of text rect to use as center point of rotation

uint32_t TText::GetMaxScroll ()

Get the maximum line you need to scroll the text to in order to display the last line of text.

Returns:

The highest integer you need to set the top line to.

virtual void TText::SetScroll (TReal *vScroll*, TReal *hScroll* = 0) [virtual]

A virtual function to override if your window can scroll.

Parameters:

vScroll Vertical scroll percentage (0.0-1.0).
hScroll Horizontal scroll percentage (0.0-1.0).

Reimplemented from TWindow.

void TText::SetScrollPadding (TReal *pad*)

This value controls how far past (in lines) the end of the text the scroll can go.

The default value is 0.5f, so this means that if you called SetScroll(1.0f), then the text would scroll so it was 0.5 lines up off the bottom of the window.

Parameters:

pad How many lines to pad the scrolling text

virtual void TText::Init (TWindowStyle & *style*) [virtual]

Initialize the Window.

Called by the system only in Lua initialization.

When you create your own custom window, this is where you put your own custom initialization that needs to happen before children are created. Fundamental window initialization is handled in every class by this function, so **when you override this function you almost always want to call your base class to handle base class initialization.**

Parameters:

style The Lua style that was in effect when this window was created. This style contains all parameters specified explicitly for the window as well as parameters defined in the current style. Parameters set locally override ones in the style.

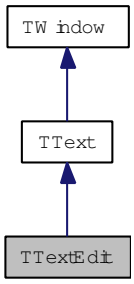
Reimplemented from TWindow.

Reimplemented in TTextEdit.

15.69 TTextEdit Class Reference

`#include <pf/textedit.h>`

Inheritance diagram for TTextEdit:



15.69.1 Detailed Description

The TTextEdit class represents an editable text TWindow.

When creating from Lua, in addition to the TWindow tags, you can specify the following tags:

Warning:

Once this class finds an ancestor modal window, it will register itself with the modal and link itself to any other TTextEdit windows it finds under that modal window. If you then remove it (or a parent) from the hierarchy without destroying it or calling Unregister(), the resulting behavior is **undefined**.

Public Types

- enum eKeyType {
  kKeyChar, kKeyMove, kKeyEnter, kKeyTab,
  kKeyPaste, kKeyIllegal }
  Key category.

Public Member Functions

- TTextEdit ()
  Default Constructor.
- virtual ~TTextEdit ()
  Destructor.
- virtual bool KeyHit (eKeyType type, char key=0)
  Virtual function to notify child that a key was pressed.
- virtual bool KeyHitUTF8 (eKeyType type, uint32_t key=0)
  New virtual function that supports UTF-8 characters.

- virtual bool OnNewParent ()
  Handle any initialization or setup that is required when this window is assigned to a new parent.
- virtual bool OnKeyDown (char key, uint32_t flags)
  Raw key hit on keyboard.
- virtual void Init (TWindowStyle &style)
  Initialize the Window.
- virtual bool OnChar (char key)
  Translated character handler.
- virtual bool OnUTF8Char (str key)
  UTF-8 Translated character handler.
- void Unregister ()
  Tell this TTextEdit that it should unlink itself from its parent modal window and any related TTextEdit windows in its tab ring.
- void SetPassword (bool bPassword)
  Sets the textedit field into password mode, meaning that the displayed text will be all asterisks.
- void SetEditable (bool bEditable)
  Set this field to be editable.
- bool GetEditable ()
  Query as to whether this TTextEdit is currently editable.
- void UpdateText ()
  Update the text in the text field.
- void Backspace ()
  Call this function to simulate pressing "Backspace" in the text field.
- void SetIgnoreChars (str ignoreStr)
  Any character in ignoreStr will be ignored and not entered into the textedit field.
- str GetIgnoreChars ()
  Get the characters this TTextEdit is ignoring.
- virtual str GetText ()
  Get the current editable text.
- virtual void SetText (str newText)
  Set the current editable text.
- virtual void Draw ()
  Function to draw dynamic elements.
- void SetMaxLength (uint32_t maxLength)

Set the maximum length of the input field.

- void SetUTF8Mode (bool utf8mode)

  Turn on UTF-8 key entry mode.

Static Public Attributes

- static const int kCursorFlashMS = 800

  Speed of the cursor flash.

- static const int kDefaultMaxLength = 10

  Default length if none given.

Protected Member Functions

- virtual bool OnTaskAnimate ()

  Animate the cursor flashing.

- uint32_t GetCursor ()

  Retrieve the current cursor position.

- void SetCursor (uint32_t cursor)

  Set the current cursor position.

Protected Attributes

- uint32_t mMaxLength

  Maximum number of characters in a string.

15.69.2 Member Enumeration Documentation

enum TTextEdit::eKeyType

Key category.

Enumerator:

kKeyChar Key is a normal character.
kKeyMove Key is a cursor or backspace character.
kKeyEnter Key is enter or return.
kKeyTab Key is the tab character.
kKeyPaste Key is the "Paste" character.
kKeyIllegal Key is illegal.

15.69.3 Member Function Documentation

virtual bool TTextEdit::KeyHit (eKeyType *type*, char *key* = 0) `[virtual]`

Virtual function to notify child that a key was pressed.

Parameters:

type key type
key key that was hit when appropriate

Returns:

True to accept the key. False to ignore.

virtual bool TTextEdit::KeyHitUTF8 (eKeyType *type*, uint32_t *key* = 0) `[virtual]`

New virtual function that supports UTF-8 characters.

Will return false for any UTF-8 character that is not in the current font.

Parameters:

type Type of key event.
key Actual key hit.

Returns:

True to accept the key. False to ignore.

virtual bool TTextEdit::OnNewParent () `[virtual]`

Handle any initialization or setup that is required when this window is assigned to a new parent.

No initialization of the window has happened prior to this call.

Returns:

True on success; false on failure.

See also:

Init
PostChildrenInit

Reimplemented from TWindow.

virtual bool TTextEdit::OnKeyDown (char *key*, uint32_t *flags*) `[virtual]`

Raw key hit on keyboard.

Parameters:

key Key pressed on keyboard. Returns either a (low-ASCII) key name, or one of the constants defined in the Key Codes section.
flags TEvent::EKeyFlags mask representing the state of other keys on the keyboard when this key was hit.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented from TWindow.

virtual void TTextEdit::Init (TWindowStyle & *style*) `[virtual]`

Initialize the Window.

Called by the system only in Lua initialization.

When you create your own custom window, this is where you put your own custom initialization that needs to happen before children are created. Fundamental window initialization is handled in every class by this function, so **when you override this function you almost always want to call your base class to handle base class initialization.**

Parameters:

style The Lua style that was in effect when this window was created. This style contains all parameters specified explicitly for the window as well as parameters defined in the current style. Parameters set locally override ones in the style.

Reimplemented from TText.

virtual bool TTextEdit::OnChar (char *key*) `[virtual]`

Translated character handler.

In UTF-8 mode, this function will only be called if OnUTF8Char() returns false, or there is no OnUTF8Char() handler.

Parameters:

key Key hit on keyboard, along with shift translations.

Returns:

true if message was handled, false to keep searching for a handler.

See also:

TPlatform::kUTF8Mode

Reimplemented from TWindow.

virtual bool TTextEdit::OnUTF8Char (str *key*) `[virtual]`

UTF-8 Translated character handler.

This function is ONLY active when UTF-8 support has been enabled.

Parameters:

key Key hit on keyboard, along with shift translations.

Returns:

true if message was handled, false to keep searching for a handler.

See also:

TPlatform::kUTF8Mode

Reimplemented from TWindow.

void TTextEdit::SetPassword (bool *bPassword*)

Sets the textedit field into password mode, meaning that the displayed text will be all asterisks.

Parameters:

bPassword - true to enable, false to disable

void TTextEdit::SetEditable (bool *bEditable*)

Set this field to be editable.
Enables the cursor, which will display when the TTextEdit is focused.

Parameters:

bEditable True to make the field editable.

bool TTextEdit::GetEditable ()

Query as to whether this TTextEdit is currently editable.

Returns:

True if editable.

void TTextEdit::SetIgnoreChars (str *ignoreStr*)

Any character in ignoreStr will be ignored and not entered into the textedit field.

Parameters:

ignoreStr A string of characters to ignore.

str TTextEdit::GetIgnoreChars ()

Get the characters this TTextEdit is ignoring.

Returns:

A string that contains the characters to ignore.

virtual str TTextEdit::GetText () `[virtual]`

Get the current editable text.

Returns:

The text in the edit box.

Reimplemented from TText.

virtual void TTextEdit::SetText (str *newText*) `[virtual]`

Set the current editable text.

Parameters:

newText Text to set to.

Reimplemented from TText.

void TTextEdit::SetMaxLength (uint32_t *maxLength*)

Set the maximum length of the input field.

Current field is not evaluated to test the new length.

Parameters:

maxLength Maximum number of characters you can type.

void TTextEdit::SetUTF8Mode (bool *utf8mode*)

Turn on UTF-8 key entry mode.

This allows the TTextEdit field to take any arbitrary UTF-8 character. At present only characters that have glyphs will be displayed.

Parameters:

utf8mode True to enable UTF-8 mode.

virtual bool TTextEdit::OnTaskAnimate () `[protected, virtual]`

Animate the cursor flashing.

Returns:

True to continue animating. False to stop.

Reimplemented from TWindow.

void TTextEdit::SetCursor (uint32_t *cursor*) `[protected]`

Set the current cursor position.

Parameters:

cursor New cursor position.

15.70 TTextGraphic Class Reference

`#include <pf/textgraphic.h>`

15.70.1 Detailed Description

Formatted text class.
A class that allows you to format text using the following tags:

-
 Line break
- Paragraph
- Bold
- Italic
- <u></u> Underline
- Center
- <left></left> Left Justify
- <right></right> Right Justify
- <tab pos="100"> Move the cursor to a specific offset, in pixels.
- <outline color="ff0000" size=2> Outline text
- Font characteristics
- Trigger a button named 'buttonname' when clicking this text.
- <cursor> Cursor icon

Text can be rendered directly to screen. For text that is automatically rendered in a window, use the TText class.
Use the static function TTextGraphic::Create to create a new instance of a TTextGraphic.

See also:

TText

Public Types

- enum EFlags {
  kHAlignLeft = 0x00, kHAlignCenter = 0x01, kHAlignRight = 0x02, kVAlignTop = 0x00,
  kVAlignCenter = 0x04, kVAlignBottom = 0x08 }
  TTextGraphic window child flags.

Public Member Functions

- void SetSpriteRender (bool activate=true)

  Activate the sprite-render style for TTextGraphic.

- void Destroy ()

  Call to destroy a TTextGraphic.

- void Draw (const TRect &destRect, TReal scale=1.0, int32_t linePadding=0, TReal alpha=1.0, TTextureRef target=TTextureRef())

  Draw the text to a rectangle on the screen.

- void Draw (const TVec2 &at, uint32_t height, TReal scale=1.0, int32_t linePadding=0, TReal alpha=1.0, TTextureRef target=TTextureRef())

  Draw the text to the screen.

- void SetNoBlend ()

  When drawing to an offscreen texture, copy pixels to the texture, instead of alpha blending them.

- void SetAlphaBlend ()

  When drawing to an offscreen texture, blend pixels into the texture and accumulate alpha.

- uint32_t GetLineCount ()

  Get the number of lines in the text output.

- void SetStartLine (TReal startLine=0)

  Set the first line in the text output.

- TReal GetStartLine ()

  Get the index of the first line of text output.

- void GetTextBounds (TRect *bounds, TReal scale=1.0F)

  Get the actual boundary of the rendered text.

- void SetText (str text)

  Set the current text content.

- str GetText () const

  Get the current text content.

- void SetColor (const TColor &color)

  Set the current text color.

- void SetLineHeight (uint32_t newHeight)

  Set a new line height for this text.

- void SetFont (const char *fontFilename)

  Set the font to be used to draw this text.

- str GetFont ()

Get the current font.

- void SetFlags (uint32_t flags)
  Set the flags used to render this text.
- void SetTextRect (uint32_t w, uint32_t h)
  Change the text rectangle.
- void SetRotation (TReal degrees, uint32_t originX, uint32_t originY)
  Change the rotation - default is 0 degrees, 0,0 origin.
- const TColor & GetColor () const
  Get the current text color.
- str Pick (const TPoint &point, int32_t linePadding=0)
  Pick an anchor record within text.
- bool Rollover (const TPoint *pPoint, int32_t linePadding=0)
  Handle rollover state for links.
- uint32_t GetMaxScroll (TReal scale, int32_t linePadding) const
  Get the number of lines this text needs to be scrolled to fit in the current text region.

Static Public Member Functions

- static TTextGraphic * Create (str text, uint32_t w, uint32_t h, uint32_t flags=0, const char *fontFilename="", uint32_t lineHeight=10, const TColor &textColor=TColor(0, 0, 0, 1))
  Create a TTextGraphic.

15.70.2 Member Enumeration Documentation

enum TTextGraphic::EFlags

TTextGraphic window child flags.

Enumerator:

kHAlignLeft Align horizontally with the left edge.
kHAlignCenter Align horizontally with the center.
kHAlignRight Align horizontally with the right edge.
kVAlignTop Align vertically with the top.
kVAlignCenter Align vertically with the center.
kVAlignBottom Align vertically with the bottom.

15.70.3 Member Function Documentation

void TTextGraphic::SetSpriteRender (bool *activate* = `true`)

Activate the sprite-render style for TTextGraphic.

The legacy style for TTextGraphic is for it to render as if the text surface is a texture, and all alignment happens relative to this "virtual" texture surface.

In the newer "sprite" render mode, which was created to help support the new TTextSprite class, rendering is done as if to a constant virtual surface size. This has no effect on left and top justified text, but centered and right or bottom justified text are drawn relative to the original surface offsets rather than scaling those offsets along with the text size.

Parameters:

activate True to activate sprite rendering for this TTextGraphic.

See also:

TPlatform::kTextGraphicSpriteRender

static TTextGraphic* TTextGraphic::Create (str *text*, uint32_t *w*, uint32_t *h*, uint32_t *flags* = `0`, const char * *fontFilename* = `""`, uint32_t *lineHeight* = `10`, const TColor & *textColor* = TColor`(0, 0, 0, 1))` `[static]`

Create a TTextGraphic.

Parameters:

text Initial text for graphic. Can be empty.
w Width of client rectangle. Must be non-zero.
h Height of client rectangle. Must be non-zero.
flags Flags from TTextGraphic::EFlags
fontFilename Font name.
lineHeight Font size.
textColor Font color.

Returns:

True on success.

void TTextGraphic::Destroy ()

Call to destroy a TTextGraphic.

Deletes the object and releases all resources.

void TTextGraphic::Draw (const TRect & *destRect*, TReal *scale* = `1.0`, int32_t *linePadding* = `0`, TReal *alpha* = `1.0`, TTextureRef *target* = TTextureRef`()`)

Draw the text to a rectangle on the screen.

Deprecated

This overload uses an older, more confusing, syntax, and doesn't allow for sub-pixel placement.

Must be called between a TRenderer::Begin2d/TRendererEnd2d pair.

Parameters:

destRect Destination rectangle in screen coordinates.

scale Scale factor - useful for zooming effects.
linePadding Additional spacing between text lines.
alpha Alpha multiplier. 1.0 is opaque.
target [optional] Target texture to draw to. Leave as default to draw to current context.

References TRect::GetHeight(), TRect::x1, and TRect::y1.

void TTextGraphic::Draw (const TVec2 & *at*, uint32_t *height*, TReal *scale* = `1.0`, int32_t *linePadding* = `0`, TReal *alpha* = `1.0`, TTextureRef *target* = TTextureRef`()`)

Draw the text to the screen.

Must be called between a TRenderer::Begin2d/TRendererEnd2d pair.

Parameters:

at Location to draw the text.
height Height of text region in pixels. Used to determine how many lines of text to draw.
scale Scale factor - useful for zooming effects.
linePadding Additional spacing between text lines.
alpha Alpha multiplier. 1.0 is opaque.
target [optional] Target texture to draw to. Leave as default to draw to current context.

void TTextGraphic::SetNoBlend ()

When drawing to an offscreen texture, copy pixels to the texture, instead of alpha blending them.

Does not work with text outlines, since the text and outline need to be blended with each other.

The default state is to blend the pixels into the destination texture, leaving alpha alone. If you call SetNoBlend(), then pixels will be set directly to the colors and alpha values, so that the texture can then be used blended with an arbitrary background. TTextGraphic::SetAlphaBlend() is similar, but uses a more complex (i.e., slower) blend algorithm that supports text outlines.

This setting has no effect when drawing to the screen.

See also:

TTextGraphic::SetAlphaBlend()

void TTextGraphic::SetAlphaBlend ()

When drawing to an offscreen texture, blend pixels into the texture and accumulate alpha.

SetAlphaBlend() uses a more sophisticated (and slower) blend algorithm than TTextGraphic::SetNoBlend() that causes it to work correctly with text outlines, as well as blending the text in with other translucent layers.

This setting has no effect when drawing to the screen.

See also:

TTextGraphic::SetNoBlend()

uint32_t TTextGraphic::GetLineCount ()

Get the number of lines in the text output.

Returns:

Number of lines.

void TTextGraphic::SetStartLine (TReal *startLine* = 0)

Set the first line in the text output.

Parameters:

startLine Line to display as the first line of text. First line is 0.

TReal TTextGraphic::GetStartLine ()

Get the index of the first line of text output.

Returns:

An index between 0 and GetLineCount()-1.

void TTextGraphic::GetTextBounds (TRect * *bounds*, TReal *scale* = 1.0F)

Get the actual boundary of the rendered text.

Parameters:

bounds A rectangle in client coordinates.
scale The scale you want the text to be at when you query the bounds.

void TTextGraphic::SetText (str *text*)

Set the current text content.

Will return immediately if the text hasn't changed.

Parameters:

text Text to set.

str TTextGraphic::GetText () const

Get the current text content.

Returns:

A string containing the current text.

void TTextGraphic::SetColor (const TColor & *color*)

Set the current text color.

Parameters:

color New text color.

void TTextGraphic::SetLineHeight (uint32_t *newHeight*)

Set a new line height for this text.

Text is re-calculated as to line wraps and tags based on new height value.

Parameters:

newHeight New text height.

void TTextGraphic::SetFont (const char * *fontFilename*)

Set the font to be used to draw this text.

Can change text bounds.

Parameters:

fontFilename Font to use.

str TTextGraphic::GetFont ()

Get the current font.

Returns:

The current font name.

void TTextGraphic::SetFlags (uint32_t *flags*)

Set the flags used to render this text.

Parameters:

flags Font alignment flags.

void TTextGraphic::SetTextRect (uint32_t *w*, uint32_t *h*)

Change the text rectangle.

Parameters:

w Width of client rectangle. Must be non-zero.
h Height of client rectangle. Must be non-zero.

void TTextGraphic::SetRotation (TReal *degrees*, uint32_t *originX*, uint32_t *originY*)

Change the rotation - default is 0 degrees, 0,0 origin.

Parameters:

degrees rotation angle in degrees (not radians because degrees are friendlier)
originX offset from left of text rect to use as center point of rotation
originY offset from top of text rect to use as center point of rotation

const TColor& TTextGraphic::GetColor () const

Get the current text color.

Returns:

The current default text color. Note this can be changed by the text markup.

str TTextGraphic::Pick (const TPoint & *point*, int32_t *linePadding* = 0)

Pick an anchor record within text.

(link)

Parameters:

point Point to test within text.
linePadding Line padding to use when Picking.

Returns:

A button name to trigger, if one is found. An empty string if no button is found.

bool TTextGraphic::Rollover (const TPoint * *pPoint*, int32_t *linePadding* = 0)

Handle rollover state for links.

Parameters:

pPoint Point to test within text, or NULL to clear the state.
linePadding Line padding to use when handling roll-over.

Returns:

True if a rollover link was triggered.

uint32_t TTextGraphic::GetMaxScroll (TReal *scale*, int32_t *linePadding*) const

Get the number of lines this text needs to be scrolled to fit in the current text region.

Parameters:

scale Scale of text.
linePadding Line padding.

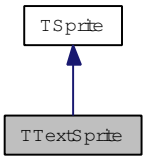
Returns:

Number of lines to scroll to get to bottom.

15.71 TTextSprite Class Reference

`#include <pf/textsprite.h>`

Inheritance diagram for TTextSprite:



15.71.1 Detailed Description

A 2d text sprite object.

A sprite that's primary purpose is to draw text at the sprite's location. Will also respect simple rotation and scaling from its matrix, and from the parent's sprite context. Any skew or non-symmetric scaling will be ignored, however.

TTextSprites are always stored in the client as TTextSpriteRef–reference counted objects. To destroy a TTextSprite, simply call reset() on the last TTextSpriteRef that refers to that sprite.

See also:

TSprite

Initialization/Destruction

- static TTextSpriteRef Create (uint32_t width, uint32_t height, uint32_t justify=0, const char *fontFilename="", uint32_t lineHeight=10, const TColor &textColor=TColor(0, 0, 0, 1), int32_t layer=0)

  Allocation of a new text sprite.

- virtual ~TTextSprite ()

  Destructor.

Public Types

- enum EFlags {
  kHAlignLeft = 0x00, kHAlignCenter = 0x01, kHAlignRight = 0x02, kVAlignTop = 0x00,
  kVAlignCenter = 0x04, kVAlignBottom = 0x08 }

  TTextSprite text alignment flags.

Public Member Functions

Drawing

- virtual void Draw (const TDrawSpec &environmentSpec=TDrawSpec(), int32_t depth=-1)

Draw the sprite and its children.

- void SetNoBlend ()
  When drawing to an offscreen texture, copy pixels to the texture, instead of alpha blending them.
- void SetAlphaBlend ()
  When drawing to an offscreen texture, blend pixels into the texture and accumulate alpha.

Bounding Rectangles and Hit Tests.

- virtual TRect GetRect (const TDrawSpec &parentContext, int32_t depth=-1)
  Get the bounding rectangle of this sprite.
- const TVec2 & GetOrigin ()
  Get the logical origin of the text in the virtual text area.
- virtual bool HitTest (const TPoint &at, const TDrawSpec &parentContext, int32_t opacity=-1, int32_t depth=-1)
  Test to see if a point is within our sprite.

Embedded Links

- str Pick (const TPoint &point)
  Pick an anchor record within text.
- bool Rollover (const TPoint *pPoint)
  Handle rollover state for links.

Text Parameters

- void SetTextRect (uint32_t w, uint32_t h)
  Set the virtual text rectangle; used for determining text wrapping and vertical alignment.
- void SetFont (const char *fontFilename)
  Set the font to be used to draw this text.
- void SetFlags (uint32_t flags)
  Set the flags used to render this text.
- uint32_t GetFlags ()
  Get the current sprite flags.
- void SetColor (const TColor &color)
  Set the default text color.
- const TColor & GetColor () const
  Get the default text color.
- str GetText () const
  Get the current text of the sprite.
- void SetText (str text)
  Set the current text of the sprite.

- void SetLineHeight (uint32_t newHeight)
  Set the rendered height of a line of text in pixels; this changes the font size.
- uint32_t GetTextHeight () const
  Query the height of the virtual text region.
- uint32_t GetTextWidth () const
  Query the width of the virtual text region.
- TRect **GetTextBounds** (const TDrawSpec &environmentSpec=TDrawSpec()) const
- uint32_t GetMaxScroll (const TDrawSpec &environmentSpec=TDrawSpec()) const
  Get the number of lines this text needs to be scrolled so that the final line fits in the current text region.
- uint32_t GetLineCount () const
  Get the number of lines in the text output.
- void SetStartLine (TReal startLine)
  Set the first line in the text output.
- TReal GetStartLine () const
  Get the index of the first line of text output.
- void SetLinePadding (int32_t padding)
  Set any additional padding between text lines.
- void SetDrawingOrigin (const TVec2 &origin)
  Set the current drawing origin.

Protected Member Functions

- TTextSprite (uint32_t w, uint32_t h, int32_t layer)
  Internal Constructor. Use TTextSprite::Create() to get a new sprite.

15.71.2 Member Enumeration Documentation

enum TTextSprite::EFlags

TTextSprite text alignment flags.

Enumerator:

kHAlignLeft Align text horizontally with the left edge of the virtual space; text will "grow" to the right from the drawing location.

kHAlignCenter Align text horizontally with the center the virtual space; text will be centered on the drawing location horizontally.

kHAlignRight Align text horizontally with the right edge of the virtual space; text will "grow" to the left from the drawing location.

kVAlignTop Align text vertically with the top edge of the virtual space; text will "grow" down from the drawing location.

kVAlignCenter Align text vertically with the center the virtual space; text will be centered on the drawing location vertically.

kVAlignBottom Align text vertically with the bottom of the virtual space; text will "grow" up from the drawing location.

15.71.3 Member Function Documentation

static TTextSpriteRef TTextSprite::Create (uint32_t *width*, uint32_t *height*, uint32_t *justify* = 0, const char * *fontFilename* = "", uint32_t *lineHeight* = 10, const TColor & *textColor* = TColor(0, 0, 0, 1), int32_t *layer* = 0) [static]

Allocation of a new text sprite.

Construction is restricted to help "encourage" the use of TTextSpriteRefs to hold your TTextSprites (as well as to encapsulate the TTextSpriteRef creation pattern).

Parameters:

width The logical width of the text region, in pixels; determines text wrapping.
height The logical height of the text region, in pixels. Determines the maximum height of drawn text.
justify The alignment of the text; see EFlags.
fontFilename The handle of the font to use.
lineHeight The initial line height.
textColor The initial color of the text.
layer Initial sprite layer.

Returns:

A newly allocated TTextSprite wrapped in a TTextSpriteRef.

See also:

TSprite::SetLayer

virtual void TTextSprite::Draw (const TDrawSpec & *environmentSpec* = TDrawSpec(), int32_t *depth* = -1) [virtual]

Draw the sprite and its children.

A common mistake is to assume that the incoming TDrawSpec here is what you use to position and configure this sprite. The way to position a sprite is to modify its internal TDrawSpec, which you retrieve with GetDrawSpec(). The environment parameter is combined with the local TDrawSpec to determine the actual position of the sprite, though it only inherits those features marked in TDrawSpec as inheritable. See TDrawSpec::GetRelative() for more details.

Parameters:

environmentSpec The 'parent' or environment drawspec–the frame of reference that this sprite is to be rendered in. In general you shouldn't pass anything in for this parameter and instead should modify the position of the sprite using its GetDrawSpec() member.

Defaults to a default-constructed TDrawSpec. See TDrawSpec for more details on what is inherited.

Parameters:

depth How many generations of children to draw; -1 means all children.

See also:

TDrawSpec

Reimplemented from TSprite.

void TTextSprite::SetNoBlend ()

When drawing to an offscreen texture, copy pixels to the texture, instead of alpha blending them.

Does not work with text outlines, since the text and outline need to be blended with each other.

The default state is to blend the pixels into the destination texture, leaving alpha alone. If you call SetNoBlend(), then pixels will be set directly to the colors and alpha values, so that the texture can then be used blended with an arbitrary background. TTextGraphic::SetAlphaBlend() is similar, but uses a more complex (i.e., slower) blend algorithm that supports text outlines.

This setting has no effect when drawing to the screen.

See also:

TTextSprite::SetAlphaBlend()

void TTextSprite::SetAlphaBlend ()

When drawing to an offscreen texture, blend pixels into the texture and accumulate alpha.

SetAlphaBlend() uses a more sophisticated (and slower) blend algorithm than TTextGraphic::SetNoBlend() that causes it to work correctly with text outlines, as well as blending the text in with other translucent layers.

This setting has no effect when drawing to the screen.

See also:

TTextSprite::SetNoBlend()

virtual TRect TTextSprite::GetRect (const TDrawSpec & *parentContext*, int32_t *depth* = -1) [virtual]

Get the bounding rectangle of this sprite.

Parameters:

parentContext The parent context to test within—where is this sprite being drawn, and with what matrix? Alpha and color information is ignored.
depth Depth of children to test

Returns:

Rectangle that includes this sprite.

Reimplemented from TSprite.

const TVec2& TTextSprite::GetOrigin ()

Get the logical origin of the text in the virtual text area.

Returns:

The point in the virtual text rectangle relative to which text will grow.

virtual bool TTextSprite::HitTest (const TPoint & *at*, const TDrawSpec & *parentContext*, int32_t *opacity* = -1, int32_t *depth* = -1) [virtual]

Test to see if a point is within our sprite.

Parameters:

at Point to test.
parentContext The parent context to test within–where is this sprite being drawn, and with what matrix? Alpha and color information is ignored.
opacity Level of opacity to test for; -1 for a simple bounding box test, or 0-255 for alpha color value, where 0 is transparent (and will therefore always succeed).
depth Depth of children to test. Zero means only test this sprite. -1 means test

Returns:

true if point hits us.

Reimplemented from TSprite.

str TTextSprite::Pick (const TPoint & *point*)

Pick an anchor record within text.

(link)

Parameters:

point Point to test within text.

Returns:

A button name to trigger, if one is found. An empty string if no button is found.

bool TTextSprite::Rollover (const TPoint * *pPoint*)

Handle rollover state for links.

Parameters:

pPoint Point to test within text, or NULL to clear the state.

Returns:

True if a rollover link was triggered.

void TTextSprite::SetTextRect (uint32_t *w*, uint32_t *h*)

Set the virtual text rectangle; used for determining text wrapping and vertical alignment.

Parameters:

w Width of virtual text region, in pixels.
h Height of virtual text region, in pixels.

void TTextSprite::SetFont (const char * *fontFilename*)

Set the font to be used to draw this text.

Can change text bounds.

Parameters:

fontFilename Font to use.

void TTextSprite::SetFlags (uint32_t *flags*)

Set the flags used to render this text.

Parameters:

flags Font alignment flags.

uint32_t TTextSprite::GetFlags ()

Get the current sprite flags.

Returns:

Current flags.

void TTextSprite::SetColor (const TColor & *color*)

Set the default text color.

Parameters:

color New text color.

const TColor& TTextSprite::GetColor () const

Get the default text color.

Returns:

Text color.

str TTextSprite::GetText () const

Get the current text of the sprite.

Returns:

Current text.

void TTextSprite::SetText (str *text*)

Set the current text of the sprite.

Will return immediately if the text hasn't changed.

Parameters:

text New text.

void TTextSprite::SetLineHeight (uint32_t *newHeight*)

Set the rendered height of a line of text in pixels; this changes the font size.

Text line wraps are re-calculated, and tag locations are adjusted, based on the new size of the font.

Parameters:

newHeight New height in pixels.

uint32_t TTextSprite::GetTextHeight () const

Query the height of the virtual text region.

Returns:

The height text region width.

uint32_t TTextSprite::GetTextWidth () const

Query the width of the virtual text region.

Returns:

The virtual text region width.

uint32_t TTextSprite::GetMaxScroll (const TDrawSpec & *environmentSpec* = TDrawSpec ( )) const

Get the number of lines this text needs to be scrolled so that the final line fits in the current text region.

Parameters:

environmentSpec The parent environment of this sprite; typically empty. Only relevant if a parent sprite contains any scaling.

Returns:

Number of lines to scroll to get to bottom.

uint32_t TTextSprite::GetLineCount () const

Get the number of lines in the text output.

Returns:

Number of lines.

void TTextSprite::SetStartLine (TReal *startLine*)

Set the first line in the text output.

Parameters:

startLine Line to display as the first line of text. First line is 0.

TReal TTextSprite::GetStartLine () const

Get the index of the first line of text output.

Returns:

An index between 0 and GetLineCount()-1.

void TTextSprite::SetLinePadding (int32_t *padding*)

Set any additional padding between text lines.

If you want "double spaced" text, you'd put the line height in this field.

Parameters:

padding The number of pixels to add between lines. Can be negative to squeeze text lines together.

void TTextSprite::SetDrawingOrigin (const TVec2 & *origin*)

Set the current drawing origin.

This value is overwritten internally if you call SetTextRect() or SetFlags().

Rotation is performed around this point, and scaling is performed relative to this point.

The default value for the origin depends on the horizontal and vertical alignment flags. When "center" is used, it defaults to the center on that axis; when one side or the other is used, the origin will be aligned with that side.

So left-top justified text will default to an origin of (0,0), while right-bottom justified text will default to (width,height). Fully centered text will default to (width/2,height/2).

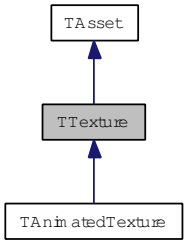
Parameters:

origin The point on the virtual text surface to draw relative to. Typically this point is between 0,0 and width,height of your virtual drawing surface, but that isn't required.

15.72 TTexture Class Reference

`#include <pf/texture.h>`

Inheritance diagram for TTexture:



15.72.1 Detailed Description

This class encapsulates the concept of a texture.

A TTexture can be used to:

- Texture a 3d object on the screen
- Draw a sprite
- Draw a screen widget (background, button, etc.)
- Be a target for 3d rendering *(which will restrict it to being a 2d blit source)*

Texture size is limited to 1024x1024 for textures that are in video RAM (i.e., any texture that isn't created as "slow").

Additionally, some textures are created from bitmaps, and those bitmaps can be dynamically MIPMAPped. Bitmaps can be read from JPG or PNG files, and the PNG files will correctly read the transparency information, if present.

The image extension can be omitted to allow the decision between JPG and PNG to be made on a an image-by-image basis without needing to change code.

Public Types

- enum ETextureCreateFlags { eDefaultAlpha = 0, eGenerateMipmaps = 1, eForceAlpha = 2, eForceNoAlpha = 4 }

  Texture loading flags used with TTexture::Get and TTexture::GetMerged.

Public Member Functions

Drawing Methods

- virtual void DrawSprite (TReal x, TReal y, TReal alpha=1, TReal scale=1, TReal rotRad=0, uint32_t flags=0)

  Draw a normal texture to a render target surface as a sprite.

- virtual void DrawSprite (const TDrawSpec &drawSpec)
  Draw a normal texture to a render target surface or backbuffer as a sprite.

- virtual void CopyPixels (int32_t x, int32_t y, const TRect *sourceRect=NULL, TTextureRef _dst=TTextureRef())
  Draw a normal or simple texture to any target TTexture surface with the same alpha as this surface.

Surface access.

- bool Lock (TColor32 **data, uint32_t *pixelPitch)
  Lock a surface for reading and writing pixel data.

- void Unlock ()
  Unlock a surface.

Information Query.

- virtual uint32_t GetWidth ()
  Get the width of the texture.

- virtual uint32_t GetHeight ()
  Get the height of the texture.

- TPoint GetInternalSize ()
  Gets the internal width and height of the texture in pixels, rather than the requested width and height.

- str GetName ()
  Get the name of the texture.

- bool IsSimple ()
  Query whether this texture is a "simple" type; simple textures can not be used in DrawSprite(), but can be locked and can be used in CopyPixels().

- bool IsSlow ()
  Query whether this texture is a "slow" texture, i.e.

- bool HasAlpha ()
  Query whether this texture has an alpha channel.

- void Clear ()
  Clear the texture to black and transparent alpha.

- bool Save (str fileName, TReal quality=1.0f)
  Save the texture to a file.

- TTextureRef GetRef ()
  Get a shared pointer (TTextureRef) to this texture.

Static Public Member Functions

Factory Methods

- static TTextureRef Get (str assetName, uint32_t flags=eDefaultAlpha)

Get a texture from a handle with optional alpha information and optional auto-generated mipmaps.

- static TTextureRef GetMerged (str colorAssetName, str alphaAssetName, uint32_t flags=eDefaultAlpha)
  Get a texture from 2 assets, one which specifies the color map and one which specifies the alpha map.
- static TTextureRef Create (uint32_t width, uint32_t height, bool alpha)
  Create a texture at a particular size.
- static TTextureRef GetSimple (str assetName)
  Get a simple texture from an asset name.
- static TTextureRef CreateSimple (uint32_t width, uint32_t height, bool slow=false, bool alpha=false)
  Create a simple texture surface.

Public Attributes

- TTextureData * mData
  Internal implementation data.

Static Protected Member Functions

- static TTextureRef InternalNew (str handle="")
  Internal function to create a TTexture.

15.72.2 Member Enumeration Documentation

enum TTexture::ETextureCreateFlags

Texture loading flags used with TTexture::Get and TTexture::GetMerged.

Enumerator:

eDefaultAlpha Select alpha based on the image content.
In the case of a PNG image that contains alpha information, alpha will be enabled; otherwise it will be disabled.
eGenerateMipmaps Generate MIPMAPs for the image.
eForceAlpha Force the image to have an alpha channel.
eForceNoAlpha Force the image to not have an alpha channel.

15.72.3 Member Function Documentation

static TTextureRef TTexture::Get (str *assetName*, uint32_t *flags* = `eDefaultAlpha`) `[static]`

Get a texture from a handle with optional alpha information and optional auto-generated mipmaps.

The handle refers to a file in the assets folder.

When passing in an assetName, flags can be passed following a "?" and seperated by commas. Example "texture.png?slow"

Maximum texture size for non-slow textures is 1024x1024.

Available flags are:

- slow - Load this texture into slow system RAM. Allows for non-square, non-power-of-two textures, as well as textures larger than 1024x1024. This flag implies "simple"–no direct drawing from this surface is allowed.
- simple - Create this texture as a simple surface. See GetSimple() for an explanation.
- alpha - Create this texture with an alpha layer.
- mipmap - Create this texture with mipmaps.

Parameters:

assetName Name of the asset. File extension (.jpg, .png) is not necessary.
flags ETextureCreateFlags

Returns:

A TTextureRef to the texture.

static TTextureRef TTexture::GetMerged (str *colorAssetName*, str *alphaAssetName*, uint32_t *flags* = `eDefaultAlpha`) `[static]`

Get a texture from 2 assets, one which specifies the color map and one which specifies the alpha map.

So that the alpha map can be a highly compressed image, the alpha value will be pulled from the "red" channel of the alpha asset.

See TTexture::Get() for information on passing flags inside of asset names.

Note that eForceNoAlpha has no effect, since the point of this function is to add an alpha channel.

Warning:

The two assets must be the same dimensions, or the alpha map will not show up correctly.

Parameters:

colorAssetName Name of the asset used for the color map.File extension (.jpg, .png) is not necessary. Supports "?" flags like TTexture::Get(), but only include those options on the colorAssetName parameter.
alphaAssetName Name of the asset used for the alpha map. File extension (.jpg, .png) is not necessary. Do not use "?" flags on alphaAssetName.
flags ETextureCreateFlags

Returns:

A TTextureRef to the texture.

static TTextureRef TTexture::Create (uint32_t *width*, uint32_t *height*, bool *alpha*) `[static]`

Create a texture at a particular size.

Maximum texture size for a created texture is 1024x1024.

Warning:

Cannot be called within a BeginDraw/EndDraw or BeginRenderTarget/EndRenderTarget block.

Parameters:

width Width
height Height
alpha Create with alpha channel

Returns:

A TTextureRef to the texture.

static TTextureRef TTexture::GetSimple (str *assetName*) `[static]`

Get a simple texture from an asset name.

Simple textures are not usable as 3d textures, and in fact can only be copied with CopyPixels(). Advantages include speed and decreased memory usage: A "Simple" texture can be created without the common restrictions of 3d textures, which often need to be powers of two in size and square.

Maximum texture size for non-slow textures is 1024x1024; slow textures have problems with either dimension larger than 4096.

Look in the documentation for TTexture::Get for information on flags that can be appended to the asset name.

Parameters:

assetName Name of the asset to load or acquire a reference to. File extension (.jpg, .png) is not necessary.

See also:

TTexture::Get()

Returns:

A TTextureRef to the texture.

static TTextureRef TTexture::CreateSimple (uint32_t *width*, uint32_t *height*, bool *slow* = `false`, bool *alpha* = `false`) `[static]`

Create a simple texture surface.

See GetSimple for more information on "simple" textures.

Maximum texture size for non-slow textures is 1024x1024; slow textures have problems with either dimension larger than 4096.

Warning:

Cannot be called within a BeginDraw/EndDraw or BeginRenderTarget/EndRenderTarget block.

Parameters:

width Width
height Height
slow True to create a slow (RAM based) texture.
alpha True to have this texture include alpha.

See also:

TTexture::GetSimple

Returns:

A TTextureRef to the created texture.

virtual void TTexture::DrawSprite (TReal *x*, TReal *y*, TReal *alpha* = 1, TReal *scale* = 1, TReal *rotRad* = 0, uint32_t *flags* = 0) [virtual]

Draw a normal texture to a render target surface as a sprite.

This draws a texture with optional rotation and scaling. Only capable of drawing an entire surface–not a sub-rectangle. See TTexture::DrawSprite(const TDrawSpec&) for more drawing control.

Will draw the sprite within the currently active viewport. X and Y are relative to the upper left corner of the current viewport.

DrawSprite can be called inside TWindow::Draw() or a BeginRenderTarget/EndRenderTarget block.

Parameters:

x X of Center.
y Y of Center.
alpha Alpha to apply to the entire texture. Set to a negative value to entirely disable alpha during blit, including alpha within the source TTexture.
scale Scaling to apply to the texture. 1.0 is no scaling.
rotRad Rotation in radians around center point.
flags Define how textures are drawn. Use ETextureDrawFlags for the flags. Default behavior is eDefaultDraw.

Reimplemented in TAnimatedTexture.

virtual void TTexture::DrawSprite (const TDrawSpec & *drawSpec*) [virtual]

Draw a normal texture to a render target surface or backbuffer as a sprite.

Uses the TDrawSpec to decide where to put the texture and how to draw it. See TDrawSpec for details.

Will draw the sprite within the currently active viewport. TDrawSpec position is relative to the upper left corner of the current viewport.

DrawSprite can be called inside TWindow::Draw() or a TRenderer::BeginRenderTarget()/EndRenderTarget block.

Parameters:

drawSpec The TDrawSpec to use to draw the sprite.

Reimplemented in TAnimatedTexture.

virtual void TTexture::CopyPixels (int32_t *x*, int32_t *y*, const TRect * *sourceRect* = NULL, TTextureRef *_dst* = TTextureRef()) [virtual]

Draw a normal or simple texture to any target TTexture surface with the same alpha as this surface.

In other words, CopyPixels can go from an alpha surface to another alpha surface, or to a non-alpha to another non-alpha surface, but not between the two.

There are no restrictions on the blit source rectangle or the destination of the blit within the target surface.

CopyPixels() does not respect alpha in its copy; it performs a bitwise, opaque copy only.

Unlike the other texture calls, CopyPixels() can be called outside of a TWindow::Draw() or a BeginRenderTarget/EndRenderTarget block.

Parameters:

x Left side of resulting rectangle.
y Top edge of resulting rectangle.
sourceRect Source rectangle to blit. NULL to blit the entire surface.
_dst Destination texture. NULL to draw to back buffer.

Reimplemented in TAnimatedTexture.

bool TTexture::Lock (TColor32 ** *data*, uint32_t * *pixelPitch*)

Lock a surface for reading and writing pixel data.

Surface will be stored in 32 bit pixels, in the binary order defined by TColor32.

Parameters:

data [out] Receives a pointer to the locked data.
pixelPitch Number of pixels to add to a pointer to advance one row.

Returns:

True on success.

virtual uint32_t TTexture::GetWidth () `[virtual]`

Get the width of the texture.

This gets the width of the texture as requested at creation or load; the actual internal width of the texture may vary. If you're using this texture as a source for TRenderer::DrawVertices, see GetInternalSize().

Returns:

Width of the texture in pixels.

Reimplemented in TAnimatedTexture.

virtual uint32_t TTexture::GetHeight () `[virtual]`

Get the height of the texture.

This gets the height of the texture as requested at creation or load; the actual internal height of the texture may vary. If you're using this texture as a source for TRenderer::DrawVertices, see GetInternalSize().

Returns:

Height of the texture in pixels.

Reimplemented in TAnimatedTexture.

TPoint TTexture::GetInternalSize ()

Gets the internal width and height of the texture in pixels, rather than the requested width and height.

Relevant if you're using the texture as a texture source for TRenderer::DrawVertices.

The texture coordinates that you set in vertices that refer to this texture need to be calculated based on the internal representation size, and not the "logical" size that GetWidth() and GetHeight() return. In other words, if

the original width is 800, and the internal width is 1024, then your U coordinate for the right edge would be 800.0F/1024.0F.

If your textures are always powers of two and square in size (equal width and height), GetInternalSize should always return the same values as GetWidth() and GetHeight().

Returns:

A point with width and height of the internal texture size, in pixels.

str TTexture::GetName ()

Get the name of the texture.

Returns:

The handle of the image used to create the texture, along with any alpha texture handle.

bool TTexture::IsSimple ()

Query whether this texture is a "simple" type; simple textures can not be used in DrawSprite(), but can be locked and can be used in CopyPixels().

See also:

TTexture::GetSimple

Returns:

True if simple

bool TTexture::IsSlow ()

Query whether this texture is a "slow" texture, i.e.

one that's been allocated in system RAM.

Returns:

True if slow.

bool TTexture::HasAlpha ()

Query whether this texture has an alpha channel.

Returns:

True if the texture has an alpha channel, false otherwise.

bool TTexture::Save (str *fileName*, TReal *quality* = `1.0f`)

Save the texture to a file.

Parameters:

fileName path of file to create including extension that defines what format to save the file as. Currently supported file extensions are: ".jpg" and ".png".

quality For file formats that use image compression, this specifies what level of compression to use (1.0 means highest quality, 0.0 means lowest quality) .png files currently ignore the quality parameterT

Returns:

true if file was successfully saved, false otherwise.

TTextureRef TTexture::GetRef ()

Get a shared pointer (TTextureRef) to this texture.

Returns:

A TTextureRef that shares ownership with other Refs to this texture.

Reimplemented from TAsset.

Reimplemented in TAnimatedTexture.

static TTextureRef TTexture::InternalNew (str *handle* = "") [static, protected]

Internal function to create a TTexture.

Parameters:

handle (optional) Handle of asset to add to asset manager. Empty to create a unique TTexture.

Returns:

A TTextureRef to the new texture.

15.73 TTransformedLitVert Struct Reference

`#include <pf/vertexset.h>`

15.73.1 Detailed Description

2d Transformed and lit vertex.
Should be used with Begin2d() operations.

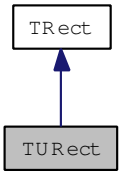
Public Attributes

- TVec3 pos
  Position in screen coordinates.
- TReal rhw
  RESERVED; must set to 0.5F.
- TColor32 color
  Vertex color.
- TColor32 specular
  Vertex specular component.
- TVec2 uv
  Vertex texture coordinate.

15.74 TURect Class Reference

`#include <pf/rect.h>`

Inheritance diagram for TURect:



15.74.1 Detailed Description

A TRect that's forced to be unsigned at all times.

Will ASSERT in debug builds if you construct a TURect with a TRect that has negative values.

See also:

TRect

Public Member Functions

- TURect ()

  Default constructor.

- TURect (uint32_t X1, uint32_t Y1, uint32_t X2, uint32_t Y2)

  Construct from four values.

- TURect (const TRect &rect)

  Construct from a TRect.

- TURect & operator= (const TRect &rect)

  Assign from a TRect.

- TURect & operator= (const TURect &rect)

  Assign from a TURect.

- TURect (const TPoint &topLeft, const TPoint &bottomRight)

  Construct a TRect from two points.

15.74.2 Constructor & Destructor Documentation

TURect::TURect (uint32_t *X1*, uint32_t *Y1*, uint32_t *X2*, uint32_t *Y2*)

Construct from four values.

Parameters:

X1 Left edge.
Y1 Top edge.
X2 One past right edge.
Y2 One past bottom edge.

TURect::TURect (const TRect & *rect*)

Construct from a TRect.

Parameters:

rect Source TRect to construct from.

TURect::TURect (const TPoint & *topLeft*, const TPoint & *bottomRight*)

Construct a TRect from two points.

Parameters:

topLeft Upper left corner of the TRect.
bottomRight Lower right corner of the TRect.

See also:

TPoint

15.74.3 Member Function Documentation

TURect& TURect::operator= (const TRect & *rect*)

Assign from a TRect.

Parameters:

rect Source rectangle.

TURect& TURect::operator= (const TURect & *rect*)

Assign from a TURect.

Parameters:

rect Source rectangle.

15.75 TVec2 Class Reference

`#include <pf/vec.h>`

15.75.1 Detailed Description

A 2d vector class.
This class is a POD (plain-old-data) type with public member data.

Public Member Functions

- TVec2 ()
  Constructor.
- TVec2 (TReal X, TReal Y)
  Initializing constructor.
- TVec2 (const TVec2 &rhs)
  Copy construction.
- TVec2 (const TVec3 &rhs)
  Conversion from a TVec3.
- TVec2 & operator= (const TVec2 &rhs)
  Assignment.
- TReal & operator[] (TIndex i)
  Member accessor.
- const TReal & operator[] (TIndex i) const
  Member accessor.
- TIndex Dim () const
  The dimension of this vector (2).
- TReal LengthSquared () const
  The length of this vector squared.
- TReal Length () const
  The length of this vector.
- TVec2 & operator+= (const TVec2 &rhs)
  Addition-assignment operator.
- TVec2 & operator-= (const TVec2 &rhs)
  Subtraction-assignment operator.

- TVec2 & operator*= (TReal s)
  Scaling operator.
- TVec2 & operator/= (TReal s)
  Scaling operator.
- TVec2 & Normalize ()
  Normalize this vector.
- TVec2 operator- () const
  Unary negation.

Public Attributes

- TReal x
  Public X dimension.
- TReal y
  Public Y dimension.

Related Functions

(Note that these are not member functions.)

- bool operator== (const TVec2 &lhs, const TVec2 &rhs)
  Equality operator.
- bool operator!= (const TVec2 &lhs, const TVec2 &rhs)
  Inequality operator.
- TVec2 operator+ (const TVec2 &lhs, const TVec2 &rhs)
  Addition operator.
- TVec2 operator- (const TVec2 &lhs, const TVec2 &rhs)
  Subtraction operator.
- TVec2 operator* (const TVec2 &lhs, TReal s)
  Scaling operator.
- TVec2 operator* (TReal s, const TVec2 &rhs)
  Scaling operator.
- TVec2 operator/ (const TVec2 &lhs, TReal s)
  Scaling operator.
- TReal DotProduct (const TVec2 &lhs, const TVec2 &rhs)
  Dot product function.

15.75.2 Constructor & Destructor Documentation

TVec2::TVec2 (TReal *X*, TReal *Y*)

Initializing constructor.

Parameters:

X X value
Y Y value

TVec2::TVec2 (const TVec3 & *rhs*) `[explicit]`

Conversion from a TVec3.

Parameters:

rhs Source TVec3. Constructor drops the z parameter.

15.75.3 Member Function Documentation

TVec2& TVec2::operator= (const TVec2 & *rhs*)

Assignment.

Returns:

A reference to this.

TReal& TVec2::operator[ ] (TIndex *i*)

Member accessor.

Parameters:

i Zero-based index.

Returns:

A reference to the i'th member.

const TReal& TVec2::operator[ ] (TIndex *i*) const

Member accessor.

Parameters:

i Zero-based index.

Returns:

A reference to the i'th member.

TIndex TVec2::Dim () const

The dimension of this vector (2).

Returns:

2

TReal TVec2::LengthSquared () const

The length of this vector squared.

Returns:

$x^2 + y^2$

TReal TVec2::Length () const

The length of this vector.

Returns:

$\sqrt{x^2 + y^2}$

TVec2& TVec2::operator+= (const TVec2 & *rhs*)

Addition-assignment operator.

Returns:

A reference to this.

TVec2& TVec2::operator-= (const TVec2 & *rhs*)

Subtraction-assignment operator.

Returns:

A reference to this.

TVec2& TVec2::operator$*$= (TReal *s*)

Scaling operator.

Parameters:

s Scale factor.

Returns:

A reference to this, scaled as $(x * s, y * s)$

TVec2& TVec2::operator/= (TReal *s*)

Scaling operator.

Parameters:

s Scale divisor.

Returns:

A reference to this, scaled as $(x/s, y/s)$

TVec2& TVec2::Normalize ()

Normalize this vector.

Changes this vector to $(x/Length(), y/Length())$

Returns:

A reference to this.

TVec2 TVec2::operator- () const

Unary negation.

Returns:

$(-x, -y)$

15.75.4 Friends And Related Function Documentation

bool operator== (const TVec2 & *lhs*, const TVec2 & *rhs*) `[related]`

Equality operator.

Returns:

True if equal.

bool operator!= (const TVec2 & *lhs*, const TVec2 & *rhs*) `[related]`

Inequality operator.

Returns:

True if not equal.

TVec2 operator+ (const TVec2 & *lhs*, const TVec2 & *rhs*) `[related]`

Addition operator.

Returns:

$(x_1 + x_2, y_1 + y_2)$

TVec2 operator- (const TVec2 & *lhs*, const TVec2 & *rhs*) `[related]`

Subtraction operator.

Returns:

$(x_1 - x_2, y_1 - y_2)$

TVec2 operator$*$ (const TVec2 & *lhs*, TReal *s*) `[related]`

Scaling operator.

Returns:

$(x * s, y * s)$

TVec2 operator$*$ (TReal *s*, const TVec2 & *rhs*) `[related]`

Scaling operator.

Returns:

$(x * s, y * s)$

TVec2 operator/ (const TVec2 & *lhs*, TReal *s*) `[related]`

Scaling operator.

Returns:

$(x/s, y/s)$

TReal DotProduct (const TVec2 & *lhs*, const TVec2 & *rhs*) `[related]`

Dot product function.

Parameters:

lhs Left-hand side of the dot product.
rhs Right-hand side of the dot product.

Returns:

The dot product of the two vectors: $(x_1 * x_2 + y_1 * y_2)$.

15.76 TVec3 Class Reference

`#include <pf/vec.h>`

15.76.1 Detailed Description

A 3d vector class.
This class is a POD (plain-old-data) type with public member data.

Public Member Functions

- TVec3 ()
  Constructor.
- TVec3 (TReal X, TReal Y, TReal Z)
  Initializing constructor.
- TVec3 (const TVec2 &rhs, TReal z=0)
  Conversion constructor.
- TVec3 (const TVec4 &rhs)
  Conversion constructor.
- TVec3 (const TVec3 &rhs)
  Copy construction.
- TVec3 & operator= (const TVec3 &rhs)
  Assignment.
- TReal & operator[] (TIndex i)
  Member accessor.
- const TReal & operator[] (TIndex i) const
  Member accessor.
- TIndex Dim () const
  The dimension of this vector (3).
- TReal LengthSquared () const
  The length of this vector squared.
- TReal Length () const
  The length of this vector.
- TVec3 & Normalize ()
  Normalize this vector.

- TVec3 & operator+= (const TVec3 &rhs)
  Addition-assignment operator.
- TVec3 & operator-= (const TVec3 &rhs)
  Subtraction-assignment operator.
- TVec3 & operator*= (TReal s)
  Scaling operator.
- TVec3 & operator/= (TReal rhs)
  Scaling operator.
- TVec3 operator- () const
  Unary negation.

Public Attributes

- TReal x
  X dimension.
- TReal y
  Y dimension.
- TReal z
  Z dimension.

Related Functions

(Note that these are not member functions.)

- bool operator== (const TVec3 &lhs, const TVec3 &rhs)
  Equality operator.
- bool operator!= (const TVec3 &lhs, const TVec3 &rhs)
  Inequality operator.
- TVec3 operator+ (const TVec3 &lhs, const TVec3 &rhs)
  Addition operator.
- TVec3 operator- (const TVec3 &lhs, const TVec3 &rhs)
  Subtraction operator.
- TVec3 operator* (const TVec3 &lhs, TReal s)
  Scaling operator.
- TVec3 operator* (TReal s, const TVec3 &rhs)
  Scaling operator.

- TVec3 operator/ (const TVec3 &lhs, TReal s)

  Scaling operator.

- TReal DotProduct (const TVec3 &lhs, const TVec3 &rhs)

  Dot product function.

- TVec3 CrossProduct (const TVec3 &lhs, const TVec3 &rhs)

  Cross product function.

- bool IntersectTriangle (const TVec3 &pvOrig, const TVec3 &pvDir, const TVec3 &pv0, const TVec3 &pv1, const TVec3 &pv2, TReal *pfDist, TVec3 *pvHit)

  Detects if a ray intersects a triangle.

15.76.2 Constructor & Destructor Documentation

TVec3::TVec3 (TReal *X*, TReal *Y*, TReal *Z*)

Initializing constructor.

Parameters:

X X value.
Y Y value.
Z Z value.

TVec3::TVec3 (const TVec2 & *rhs*, TReal *z* = 0) `[explicit]`

Conversion constructor.

Parameters:

rhs TVec2 to convert from.
z Additional z component to add; defaults to zero.

TVec3::TVec3 (const TVec4 & *rhs*) `[explicit]`

Conversion constructor.

Parameters:

rhs TVec4 to convert from. Drops the w component.

15.76.3 Member Function Documentation

TVec3& TVec3::operator= (const TVec3 & *rhs*)

Assignment.

Returns:

A reference to this.

TReal& TVec3::operator[] (TIndex *i*)

Member accessor.

Parameters:

i Zero-based index.

Returns:

A reference to the i'th member.

const TReal& TVec3::operator[] (TIndex *i*) const

Member accessor.

Parameters:

i Zero-based index.

Returns:

A reference to the i'th member.

TIndex TVec3::Dim () const

The dimension of this vector (3).

Returns:

3

TReal TVec3::LengthSquared () const

The length of this vector squared.

Returns:

$x^2 + y^2 + z^2$

TReal TVec3::Length () const

The length of this vector.

Returns:

$\sqrt{x^2 + y^2 + z^2}$

TVec3& TVec3::Normalize ()

Normalize this vector.

Changes this vector to $(x/Length(), y/Length(), z/Length())$

Returns:

A reference to this.

TVec3& TVec3::operator+= (const TVec3 & *rhs*)

Addition-assignment operator.

Returns:

A reference to this.

TVec3& TVec3::operator-= (const TVec3 & *rhs*)

Subtraction-assignment operator.

Returns:

A reference to this.

TVec3& TVec3::operator$*$= (TReal *s*)

Scaling operator.

Parameters:

s Scale factor.

Returns:

A reference to this, scaled as $(x*s, y*s, z*s)$

TVec3& TVec3::operator/= (TReal *rhs*)

Scaling operator.

Parameters:

s Scale divisor.

Returns:

A reference to this, scaled as $(x/s, y/s, z/s)$

TVec3 TVec3::operator- () const

Unary negation.

Returns:

$(-x, -y, -z)$

15.76.4 Friends And Related Function Documentation

bool operator== (const TVec3 & *lhs*, const TVec3 & *rhs*) `[related]`

Equality operator.

Returns:

True if equal.

bool operator!= (const TVec3 & *lhs*, const TVec3 & *rhs*) `[related]`

Inequality operator.

Returns:

True if not equal.

TVec3 operator+ (const TVec3 & *lhs*, const TVec3 & *rhs*) `[related]`

Addition operator.

Returns:

$(x_1 + x_2, y_1 + y_2, z_1 + z_2)$

TVec3 operator- (const TVec3 & *lhs*, const TVec3 & *rhs*) `[related]`

Subtraction operator.

Returns:

$(x_1 - x_2, y_1 - y_2, z_1 - z_2)$

TVec3 operator$*$ (const TVec3 & *lhs*, TReal *s*) `[related]`

Scaling operator.

Returns:

$(x * s, y * s, z * s)$

TVec3 operator∗ (TReal *s*, const TVec3 & *rhs*) [related]

Scaling operator.

Returns:

$(x*s, y*s, z*s)$

TVec3 operator/ (const TVec3 & *lhs*, TReal *s*) [related]

Scaling operator.

Returns:

$(x/s, y/s, z/s)$

TReal DotProduct (const TVec3 & *lhs*, const TVec3 & *rhs*) [related]

Dot product function.

Returns:

The dot product of the two vectors: $(x_1*x_2 + y_1*y_2 + z_1*z_2)$.

TVec3 CrossProduct (const TVec3 & *lhs*, const TVec3 & *rhs*) [related]

Cross product function.

Returns:

The cross product of the two vectors.

bool IntersectTriangle (const TVec3 & *pvOrig*, const TVec3 & *pvDir*, const TVec3 & *pv0*, const TVec3 & *pv1*, const TVec3 & *pv2*, TReal ∗ *pfDist*, TVec3 ∗ *pvHit*) [related]

Detects if a ray intersects a triangle.

Parameters:

← ***pvOrig*** Points to ray origin.
← ***pvDir*** Points to ray direction from origin.
← ***pv0*** Points to triangle vertex0.
← ***pv1*** Points to triangle vertex1.
← ***pv2*** Points to triangle vertex2.
→ ***pfDist*** Points to where the distance from vOrig to the point of intersection gets stored.
→ ***pvHit*** Points to where the actual point of intersection gets stored.

Returns:

true if ray intersects triangle.

15.77 TVec4 Class Reference

`#include <pf/vec.h>`

15.77.1 Detailed Description

A 4d vector class.
This class is a POD with public member data.

Public Member Functions

- TVec4 ()
  Constructor.
- TVec4 (TReal X, TReal Y, TReal Z, TReal W)
  Initializing constructor.
- TVec4 (const TVec3 &rhs, TReal W=0.0)
  Conversion from a TVec3.
- TVec4 (const TVec4 &rhs)
  Copy construction.
- TVec4 & operator= (const TVec4 &rhs)
  Assignment.
- TReal & operator[ ] (TIndex i)
  Member accessor.
- const TReal & operator[ ] (TIndex i) const
  Member accessor.
- TIndex Dim () const
  The dimension of this vector (4).
- TReal LengthSquared () const
  The length of this vector squared.
- TReal Length () const
  The length of this vector.
- TVec4 & Normalize ()
  Normalize this vector.
- TVec4 & operator+= (const TVec4 &rhs)
  Addition-assignment operator.

- TVec4 & operator-= (const TVec4 &rhs)
  Subtraction-assignment operator.
- TVec4 & operator*= (TReal rhs)
  Scaling operator.
- TVec4 & operator/= (TReal rhs)
  Scaling operator.
- TVec4 operator- () const
  Unary negation.

Public Attributes

- TReal x
  X dimension.
- TReal y
  Y dimension.
- TReal z
  Z dimension.
- TReal w
  W dimension.

Related Functions

(Note that these are not member functions.)

- bool operator== (const TVec4 &lhs, const TVec4 &rhs)
  Equality operator.
- bool operator!= (const TVec4 &lhs, const TVec4 &rhs)
  Inequality operator.
- TVec4 operator+ (const TVec4 &lhs, const TVec4 &rhs)
  Addition operator.
- TVec4 operator- (const TVec4 &lhs, const TVec4 &rhs)
  Subtraction operator.
- TVec4 operator* (const TVec4 &lhs, TReal s)
  Scaling operator.
- TVec4 operator* (TReal s, const TVec4 &rhs)
  Scaling operator.

- TVec4 operator/ (const TVec4 &lhs, TReal rhs)

  Scaling operator.

- TReal DotProduct (const TVec4 &lhs, const TVec4 &rhs)

  Dot product function.

- TVec4 CrossProduct (const TVec4 &a, const TVec4 &b, const TVec4 &c)

  Cross product function.

15.77.2 Constructor & Destructor Documentation

TVec4::TVec4 (TReal *X*, TReal *Y*, TReal *Z*, TReal *W*)

Initializing constructor.

Parameters:

X X value (v[0]).
Y Y value (v[1]).
Z Z value (v[2]).
W W value (v[3]).

TVec4::TVec4 (const TVec3 & *rhs*, TReal *W* = `0.0`) **`[explicit]`**

Conversion from a TVec3.

Parameters:

rhs Source TVec3.
W W component to add. Defaults to zero.

15.77.3 Member Function Documentation

TVec4& TVec4::operator= (const TVec4 & *rhs*)

Assignment.

Returns:

A reference to this.

TReal& TVec4::operator[] (TIndex *i*)

Member accessor.

Parameters:

i Zero-based index.

Returns:

A reference to the i'th member.

const TReal& TVec4::operator[ ] (TIndex *i*) const

Member accessor.

Parameters:

i Zero-based index.

Returns:

A reference to the i'th member.

TIndex TVec4::Dim () const

The dimension of this vector (4).

Returns:

4

TReal TVec4::LengthSquared () const

The length of this vector squared.

Returns:

$x^2 + y^2 + z^2 + w^2$

TReal TVec4::Length () const

The length of this vector.

Returns:

$\sqrt{x^2 + y^2 + z^2 + w^2}$

TVec4& TVec4::Normalize ()

Normalize this vector.

Changes vector to $(x/Length(), y/Length(), z/Length(), w/Length())$

Returns:

A reference to this.

TVec4& TVec4::operator+= (const TVec4 & *rhs*)

Addition-assignment operator.

Returns:

A reference to this.

TVec4& TVec4::operator-= (const TVec4 & *rhs*)

Subtraction-assignment operator.

Returns:

A reference to this.

TVec4& TVec4::operator∗= (TReal *rhs*)

Scaling operator.

Parameters:

s Scale factor.

Returns:

A reference to this, scaled as $(x*s, y*s)$

TVec4& TVec4::operator/= (TReal *rhs*)

Scaling operator.

Parameters:

s Scale divisor.

Returns:

A reference to this, scaled as $(x/s, y/s)$

TVec4 TVec4::operator- () const

Unary negation.

Returns:

$(-x, -y, -z, -w)$

15.77.4 Friends And Related Function Documentation

bool operator== (const TVec4 & *lhs*, const TVec4 & *rhs*) `[related]`

Equality operator.

Returns:

True if equal.

bool operator!= (const TVec4 & *lhs*, const TVec4 & *rhs*) `[related]`

Inequality operator.

Returns:

True if not equal.

TVec4 operator+ (const TVec4 & *lhs*, const TVec4 & *rhs*) `[related]`

Addition operator.

Returns:

$(x_1 + x_2, y_1 + y_2, z_1 + z_2, w_1 + w_2)$

TVec4 operator- (const TVec4 & *lhs*, const TVec4 & *rhs*) `[related]`

Subtraction operator.

Returns:

$(x_1 - x_2, y_1 - y_2, z_1 - z_2, w_1 - w_2)$

TVec4 operator∗ (const TVec4 & *lhs*, TReal *s*) `[related]`

Scaling operator.

Returns:

$(x * s, y * s)$

TVec4 operator∗ (TReal *s*, const TVec4 & *rhs*) `[related]`

Scaling operator.

Returns:

$(x * s, y * s)$

TVec4 operator/ (const TVec4 & *lhs*, TReal *rhs*) `[related]`

Scaling operator.

Returns:

$(x/s, y/s, z/s, w/s)$

TReal DotProduct (const TVec4 & *lhs*, const TVec4 & *rhs*) `[related]`

Dot product function.

Returns:

The dot product of the two vectors: $(x_1 * x_2 + y_1 * y_2 + z_1 * z_2 + w_1 * w * 2)$.

TVec4 CrossProduct (const TVec4 & *a*, const TVec4 & *b*, const TVec4 & *c*) `[related]`

Cross product function.

Returns:

The cross product of the two vectors.

15.78 TVert Struct Reference

`#include <pf/vertexset.h>`

15.78.1 Detailed Description

3d untransformed, unlit vertex.

Public Attributes

- TVec3 pos

  Position in 3d space.

- TVec3 normal

  Vertex normal. Must not *be (0,0,0), and must be normalized.*

- TVec2 uv

  Vertex texture coordinate.

15.79 TVertexSet Class Reference

`#include <pf/vertexset.h>`

15.79.1 Detailed Description

A helper/wrapper for the Vertex Types which allows TRenderer::DrawVertices to identify the vertex type being passed in without making the vertex types polymorphic.

Public Member Functions

- TVertexSet (TTransformedLitVert *v, uint32_t count)

  Create a vertex set from an existing external array of TTransformedLitVerts.

- TVertexSet (TLitVert *v, uint32_t count)

  Create a vertex set from an existing external array of TLitVerts.

- TVertexSet (TVert *v, uint32_t count)

  Create a vertex set from an existing, external array of TVerts.

- TVertexSet (const TVertexSet &)

  Copy construction: Needs to be implemented because of an inane rule in the ISO standard.

- TVertexSet & operator= (const TVertexSet &)

  Assignment.

- ~TVertexSet ()

  Destructor.

- void SetCount (uint32_t count)

  Change the vertex count.

Static Public Attributes

- static const uint32_t kMaxVertices = 65535

  A hard limit on the number of vertices that you can specify.

15.79.2 Constructor & Destructor Documentation

TVertexSet::TVertexSet (TTransformedLitVert * *v*, uint32_t *count*)

Create a vertex set from an existing external array of TTransformedLitVerts.

Does NOT copy the vertices–only keeps a pointer to them.

Parameters:

v Pointer to array of vertices.
count Number of vertices in array. Must not be more than kMaxVertices.

TVertexSet::TVertexSet (TLitVert * *v*, uint32_t *count*)

Create a vertex set from an existing external array of TLitVerts.

Does NOT copy the vertices–only keeps a pointer to them.

Parameters:

v Pointer to array of vertices.
count Number of vertices in array. Must not be more than kMaxVertices.

TVertexSet::TVertexSet (TVert * *v*, uint32_t *count*)

Create a vertex set from an existing, external array of TVerts.

Does NOT copy the vertices–only keeps a pointer to them.

Parameters:

v Pointer to array of vertices.
count Number of vertices in array. Must not be more than kMaxVertices.

TVertexSet::TVertexSet (const TVertexSet &)

Copy construction: Needs to be implemented because of an inane rule in the ISO standard.

See https://developer.playfirst.com/node/158 for details.

15.79.3 Member Function Documentation

TVertexSet& TVertexSet::operator= (const TVertexSet &)

Assignment.

Returns:

A reference to this

void TVertexSet::SetCount (uint32_t *count*)

Change the vertex count.

This allows you to create a single TVertexSet and reuse its data repeatedly without reconstructing it.

Maximum count is kMaxVertices.

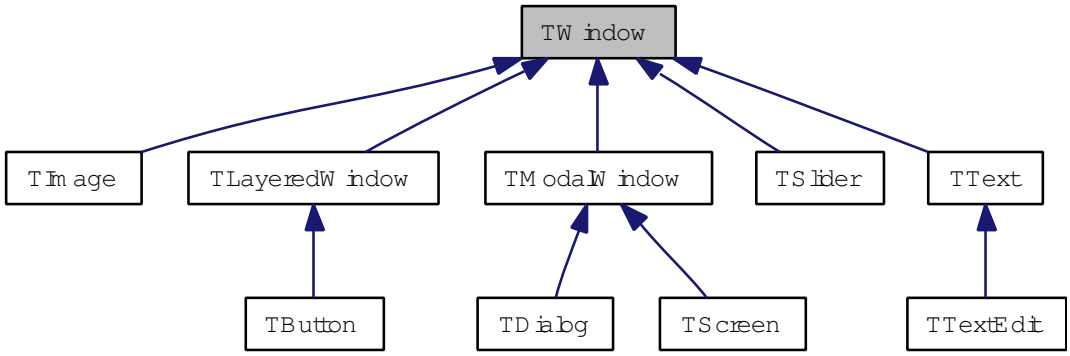
Parameters:

count Number of vertices in the set.

15.80 TWindow Class Reference

`#include <pf/window.h>`

Inheritance diagram for TWindow:



15.80.1 Detailed Description

The TWindow class is the base class of any object that needs to draw to the screen.

As you create your own custom TWindow-derived classes, you should be aware that a TWindow "owns" its children: When a TWindow is destroyed, it expects to be able to delete its children. Therefore a TWindow should never be allocated on the stack. If you want a window to persist longer than its parent, you need to ensure that it is removed from the parent prior to the parent's destruction.

Child and Parent Window Functions.

Functions to find child windows and to retreive and set parent windows.

- virtual bool AdoptChild (TWindow *child, bool initWindow=true)
  Add a child to this window.
- virtual void OrphanChild (TWindow *child)
  Remove a child from this window.
- void DestroyAllChildren ()
  Destroy (delete) all child windows.
- void FitToChildren ()
  Fit this window to its childrens' sizes.
- TWindow * ChildWindowFromPoint (const TPoint &point, int32_t depth=1)
  Recursively find a child window from the given point.
- TModalWindow * FindParentModal ()
  Find the nearest direct-ancestor modal window.

- virtual void OnParentModalPopped ()
  This method is called when this window's parent modal has been removed from the window stack.
- void ForEachChild (TWindowSpider *spider, bool reverse=false)
  Iterate through all children and call the Process() member function of TWindowSpider.
- bool HasChildren ()
  Return true if this window has children.
- TWindow * GetChildWindow (str name, int32_t depth=-1)
  Return the descendant window with the given name, if one exists.
- TWindow * GetParent () const
  Get the current window parent.
- void SetWindowDepth (TWindow *inFrontOf=NULL)
  Reposition this window to be immediately in front of a given sibling.
- void SetWindowDepth (EDepth depth)
  Reposition this window to be in the position specified by the given constant.

Construction and Initialization

- TWindow ()
  Default Constructor.
- virtual ~TWindow ()
  Destructor.
- virtual bool OnNewParent ()
  Handle any initialization or setup that is required when this window is assigned to a new parent.
- virtual void Init (TWindowStyle &style)
  Initialize the Window.
- virtual void PostChildrenInit (TWindowStyle &style)
  Do post-children-added initialization when being created from Lua.
- void SizeAndPositionFromStyle (TWindowStyle &style)
  A function that takes the default window parameters and applies them to the window's position and size.

Internal Utility Functions

Functions that are only called internally by the library.

- bool HandleEvent (const TEvent *eIn, bool focusChanged=false)
  Internal event handling function.

- str GetID () const
  Get a unique window identifier.

- TWindow * SetParent (TWindow *newParent)
  Set the current parent window.

Public Types

Locally Defined Types

Types defined in the TWindow scope.

- enum ETypeFlags {
  kModal = 0x00000001, kFocusTarget = 0x00000002, kInfrequentChanges = 0x00000004, kStartGroup = 0x00000008,
  kTypeMask = 0x000000FF }
  Static Window Types.

- enum EDepth {
  kBackMost, kFrontMost, kOneHigher, kOneLower,
  kDepthCount }
  Position constants for SetWindowDepth.

- enum EStateFlags {
  kEnabled = 0x00000100, kChecked = 0x00000200, kCached = 0x00000400, kOpaque = 0x00000800,
  kStateMask = 0x0000FF00 }
  Dynamic Window States.

- enum EDrawMode { eAll = 0, eCached, eDynamic }
  Window drawing mode.

- typedef std::list< TWindow * > WindowList
  A list of owned windows. Used for children.

- typedef std::list< TRect > RectList
  List of rectangles.

Public Member Functions

Type Information and Casting

- PFClassId ClassId ()
  Get the ClassId.

- virtual bool IsKindOf (PFClassId type)
  Determine whether this window is derived from type.

- template<class TO>
  TO * GetCast ()
  Safely cast this window to another type.

Update Functions

Functions related to the drawing of windows.

- virtual void Draw ()
  Draw the window.
- virtual void PostDraw ()
  Draw any overlays that should appear on top of this window's children.

Window Coordinates and Rectangles.

Functions to calculate window point conversions relative to two windows, and to acquire the window and client rectangles.

- void GetWindowRect (TRect *rect) const
  Get the rectangle that specifies the current window in top-level TScreen coordinates.
- void SetWindowPos (const TPoint &point)
  Set the position of the upper left corner of the window in parent client coordinates.
- void SetWindowPos (int32_t x, int32_t y)
  Set the position of the upper left corner of the window in parent client coordinates.
- const TPoint & GetWindowPos () const
  Get the current window position.
- void SetWindowSize (uint32_t width, uint32_t height)
  Set the size of the window.
- void GetClientRect (TRect *rect) const
  Get the "client" rectangle of the current window.
- uint32_t GetWindowWidth () const
  Get the width of the client area of this window.
- uint32_t GetWindowHeight () const
  Get the height of the client area of this window.
- void ScreenToClient (TPoint *point)
  Convert between top-level screen and client coordinates.
- void ScreenToClient (TRect *rect)
  Convert between top-level screen and client coordinates.
- void ClientToScreen (TPoint *point)
  Convert between client and top-level screen coordinates.
- void ClientToScreen (TRect *rect)
  Convert between client and top-level screen coordinates.
- void ParentToClient (TPoint *point) const
  Convert between parent and client coordinates.
- void ParentToClient (TRect *rect) const
  Convert between parent and client coordinates.

- void ClientToParent (TPoint *point)
  Convert between client and parent coordinates.
- void ClientToParent (TRect *rect)
  Convert between parent and client coordinates.
- void GetParentRelativeRect (TRect *rect) const
  Get the rectangle that represents this window in the client space of its parent window.
- const TRect & GetParentRelativeRect () const
  Get the rectangle that represents this window in the client space of its parent window.

Window Information Accessors.

Functions to get or set information about a window.

- virtual void SetScroll (float vScroll, float hScroll=0)
  A virtual function to override if your window can scroll.
- uint32_t GetFlags () const
  Get the window's state and style flags.
- void SetFlags (uint32_t flags)
  Set the state flags of the window.
- str GetName () const
  Get the window name, if any.
- void SetName (str name)
  Set the window name.
- bool IsOpaque ()
  Query this window's opacity.
- bool IsModal ()
  Query this window's modal status.
- bool IsEnabled ()
  Return whether this window and all of its ancestors are enabled.

Event Handlers

Functions to override to handle events in a window, and functions to trigger events on a window.

- void SendWindowMessage (TMessage *message)
  Send a message to a window (or its ancestor).
- void StartWindowAnimation (int32_t delay, bool autoRepeat=true, bool resetTime=true, bool forceFrequency=false)
  Start a window animation.
- void StopWindowAnimation ()
  Stop a window from receiving OnTaskAnimate calls.
- virtual bool OnMessage (TMessage *message)
  Handle a message.

- virtual bool OnTaskAnimate ()
  Called if you have initiated a window animation with TWindow::StartWindowAnimation.
- virtual bool OnMouseDown (const TPoint &point)
  Mouse down handler.
- virtual bool OnExtendedMouseEvent (const TPoint &point, TPlatform::ExtendedMouseEvents event)
  Extended mouse button handler.
- virtual bool OnMouseUp (const TPoint &point)
  Mouse up handler.
- virtual bool OnMouseMove (const TPoint &point)
  Mouse motion handler.
- virtual bool OnMouseLeave ()
  Notification that the mouse has left the window.
- virtual bool CanAcceptFocus ()
  Returns true if this window can accept the keyboard focus.
- virtual bool OnChar (char key)
  Translated character handler.
- virtual bool OnUTF8Char (str key)
  UTF-8 Translated character handler.
- virtual bool OnKeyDown (char key, uint32_t flags)
  Raw key hit on keyboard.
- virtual bool OnKeyUp (char key)
  Raw key released on keyboard.
- virtual bool OnMouseHover (const TPoint &point)
  Called if the mouse hovers over a point on the window.
- virtual void OnSetFocus (TWindow *previous)
  This window is receiving the keyboard focus.
- virtual void OnKillFocus (TWindow *newFocus)
  This window is losing the keyboard focus.

Protected Member Functions

- bool IsEnabledSimple ()
  Return whether this window is enabled, looking at parents, but ignoring whether it descends from a TModalWindow.
- void AddWindowType (uint32_t type)
  Function that allows a derived window to add type flags to the TWindow.
- TAnimTask * GetWindowAnim ()
  Get the associated TAnimTask.

Protected Attributes

- WindowList mChildren

  We own our children. Our descendants can play with our children, though.

15.80.2 Member Enumeration Documentation

enum TWindow::ETypeFlags

Static Window Types.

Flags that define what type and/or class a window is. These do not change after window creation.

Enumerator:

kModal Flag that this window is modal.
kFocusTarget Flag that this window can accept focus.
kInfrequentChanges Hint that we could cache this window. Only works if all ancestors are also flagged.
kStartGroup This window is the start of a group of siblings (i.e. for radio buttons).
kTypeMask A flag mask that isolates the window types.

enum TWindow::EDepth

Position constants for SetWindowDepth.

See also:

TWindow::SetWindowDepth

Enumerator:

kBackMost Set this window to be the backmost window.
kFrontMost Set this window to be the frontmost window.
kOneHigher Set this window to be one higher than its current position.
kOneLower Set this window to be one lower than its current position.
kDepthCount Number of depth options.

enum TWindow::EStateFlags

Dynamic Window States.

States that may change frequently after window creation.

Enumerator:

kEnabled This window is enabled, and therefore can be rendered to and clicked upon.
kChecked This window is in its "selected" or "checked" state.
kCached This window is rendered to the cache.
kOpaque This window uses no alpha blending when it draws itself, and covers its rectangle completely. It's important to set this flag on a window when it's full screen and should completely obscure the windows behind it–this will allow Playground to prevent the deeper window from drawing.
kStateMask A flag mask that isolates the window states.

enum TWindow::EDrawMode

Window drawing mode.

Enumerator:

eAll Draw all layers.
eCached Draw only cacheable layers.
eDynamic Draw only dynamic layers.

15.80.3 Member Function Documentation

virtual bool TWindow::OnNewParent () `[virtual]`

Handle any initialization or setup that is required when this window is assigned to a new parent.

No initialization of the window has happened prior to this call.

Returns:

True on success; false on failure.

See also:

Init
PostChildrenInit

Reimplemented in TButton, TDialog, TModalWindow, and TTextEdit.

virtual void TWindow::Init (TWindowStyle & *style*) `[virtual]`

Initialize the Window.

Called by the system only in Lua initialization.

When you create your own custom window, this is where you put your own custom initialization that needs to happen before children are created. Fundamental window initialization is handled in every class by this function, so **when you override this function you almost always want to call your base class to handle base class initialization.**

Parameters:

style The Lua style that was in effect when this window was created. This style contains all parameters specified explicitly for the window as well as parameters defined in the current style. Parameters set locally override ones in the style.

Reimplemented in TButton, TImage, TSlider, TText, and TTextEdit.

virtual void TWindow::PostChildrenInit (TWindowStyle & *style*) `[virtual]`

Do post-children-added initialization when being created from Lua.

Any initialization that needs to happen after a window's children have been added can be placed in a derived version of this function.

Warning:

Remember to always call the base class if you're overriding this function.

Parameters:

style Current style environment that this window was created in.

Reimplemented in TButton, and TModalWindow.

void TWindow::SizeAndPositionFromStyle (TWindowStyle & *style*) `[protected]`

A function that takes the default window parameters and applies them to the window's position and size.

Note:

This function is for advanced users only.

Called by TWindow::Init() *and* by TWindow::PostChildrenInit().

Has no effect if it can't calculate the position and size based on current information available: If a position is set to kCenter, or a size set to kMax, but the parent window size hasn't yet been calculated, this function won't do anything.

Note:

Implementation details.

Since some windows set their size based on their calculated children's sizes (using TWindow::FitToChildren), TWindow::PostChildrenInit() needs to call this to adjust the position after the size has been calculated. Since other windows must have their full position and size specified in order to properly initialize, TWindow::Init() needs to call this function.

Parameters:

style The style of the window to apply.

PFClassId TWindow::ClassId ()

Get the ClassId.

Returns:

A ClassId that can be passed to IsKindOf.

See also:

Type Information and Casting

bool TWindow::IsKindOf (PFClassId *type*) `[virtual]`

Determine whether this window is derived from type.

Parameters:

type ClassId() of type to test.

See also:

Type Information and Casting

template<class TO> template< class TO > TO * TWindow::GetCast ()

Safely cast this window to another type.

Returns:

A cast pointer, or an empty reference.

See also:

Type Information and Casting

virtual void TWindow::Draw () `[virtual]`

Draw the window.

Derived classes will override this function and provide the draw functionality.

Reimplemented in TImage, TSlider, TText, and TTextEdit.

virtual bool TWindow::AdoptChild (TWindow * *child*, bool *initWindow* = `true`) `[virtual]`

Add a child to this window.

Warning:

If you override this in a derived class, be sure to call the base class to actually add the child from the list of children.

Parameters:

child Child that's being added.
initWindow True to call OnNewParent().

Returns:

True if successful. On false, the window has NOT been adopted and the calling class still has responsibility for destruction.

Reimplemented in TLayeredWindow.

virtual void TWindow::OrphanChild (TWindow * *child*) `[virtual]`

Remove a child from this window.

Warning:

If you override this in a derived class, be sure to call the base class to actually remove the child from the list of children.

Parameters:

child Child that's being removed.

Reimplemented in TLayeredWindow.

void TWindow::DestroyAllChildren ()

Destroy (delete) all child windows.

Those windows will destroy their own children. Actual deletion is deferred using TWindowManager::SafeDestroyWindow(), so the windows will be actually deleted in the next event loop.

TWindow* TWindow::ChildWindowFromPoint (const TPoint & *point*, int32_t *depth* = 1)

Recursively find a child window from the given point.

The point is assumed to be inside *this* window.

When searching for children, ChildWindowFromPoint() will ignore any windows that are disabled (kEnabled is not set).

Parameters:

point Point to test in client coordinates.
depth The number of times to recurse. 1 gives you only immediate children. -1 gives you the deepest child. Defaults to 1.

Returns:

A pointer to the window containing the point. If the point is not inside any of the child windows, the function will return this.

TModalWindow* TWindow::FindParentModal ()

Find the nearest direct-ancestor modal window.

Returns:

A pointer to a modal window, or NULL if none is found.

virtual void TWindow::OnParentModalPopped () [virtual]

This method is called when this window's parent modal has been removed from the window stack.

Because window deletion is delayed until it is safe to delete the window, this method can be used to detect immediately when a window has been removed from the stack, whereas the destructor will only be called when the window is actually deleted.

void TWindow::ForEachChild (TWindowSpider * *spider*, bool *reverse* = false)

Iterate through all children and call the Process() member function of TWindowSpider.

Iteration happens in front-to-back order by default, or back-to-front if reverse is true

Parameters:

spider The derived class which contains a Process() function to be called on each child window.
reverse If this is true, back-to-front order is used, default is false

bool TWindow::HasChildren ()

Return true if this window has children.

Returns:

True if we're parents; false otherwise.

TWindow* TWindow::GetChildWindow (str *name*, int32_t *depth* = −1)

Return the descendant window with the given name, if one exists.

Parameters:

name Window name to search for.
depth Number of levels deep to look. Set to -1 for no limit. Defaults to -1.

Returns:

A TWindow to the descendant with the given name. Children are searched recursively up to the level indicated in depth.

TWindow* TWindow::GetParent () const

Get the current window parent.

Returns:

A shared pointer to the current parent window.

Note:

This will return NULL if the current window has no parent.

Referenced by IsEnabled(), and IsEnabledSimple().

void TWindow::SetWindowDepth (TWindow * *inFrontOf* = NULL)

Reposition this window to be immediately in front of a given sibling.

Pass the results of GetParent()->GetFirstChild() to bring this window to the front.

Parameters:

inFrontOf Sibling we should be visually in front of. If NULL, will place this window in the back.

void TWindow::SetWindowDepth (EDepth *depth*)

Reposition this window to be in the position specified by the given constant.

Parameters:

depth Enumeration that specifies a logical window depth.

See also:

EDepth

void TWindow::GetWindowRect (TRect * *rect*) const

Get the rectangle that specifies the current window in top-level TScreen coordinates.

Parameters:

rect TRect to fill with resulting rectangle.

void TWindow::SetWindowPos (const TPoint & *point*)

Set the position of the upper left corner of the window in parent client coordinates.

Parameters:

point New window position.

See also:

SetWindowSize
SetWindowDepth

void TWindow::SetWindowPos (int32_t *x*, int32_t *y*)

Set the position of the upper left corner of the window in parent client coordinates.

Parameters:

x New window x coordinate.
y New window y coordinate.

See also:

SetWindowSize
SetWindowDepth

const TPoint& TWindow::GetWindowPos () const

Get the current window position.

Returns:

The current window position relative to its parent.

void TWindow::SetWindowSize (uint32_t *width*, uint32_t *height*)

Set the size of the window.

Parameters:

width New window width.
height New window height.

void TWindow::GetClientRect (TRect * *rect*) const

Get the "client" rectangle of the current window.

This mimics the Windows functionality of getting a rect that has top and left set to 0, with right and bottom set to width and height, respectively.

Parameters:

rect The TRect to fill with the client rectangle.

uint32_t TWindow::GetWindowWidth () const

Get the width of the client area of this window.

Returns:

Window client width in pixels.

uint32_t TWindow::GetWindowHeight () const

Get the height of the client area of this window.

Returns:

Window client height in pixels.

void TWindow::ScreenToClient (TPoint * *point*)

Convert between top-level screen and client coordinates.

Parameters:

point in: Screen coordinates, out:client coordinates.

void TWindow::ScreenToClient (TRect * *rect*)

Convert between top-level screen and client coordinates.

Parameters:

rect in: Screen coordinates, out:client coordinates.

References TRect::GetBottomRight(), and TRect::GetTopLeft().

void TWindow::ClientToScreen (TPoint * *point*)

Convert between client and top-level screen coordinates.

Parameters:

point in: client coordinates, out: screen coorditates.

void TWindow::ClientToScreen (TRect * *rect*)

Convert between client and top-level screen coordinates.

Parameters:

rect in: client coordinates, out: screen coorditates.

References TRect::GetBottomRight(), and TRect::GetTopLeft().

void TWindow::ParentToClient (TPoint * *point*) const

Convert between parent and client coordinates.

Parameters:

point in: a point in parent's coordinate system, out:client coordinates

void TWindow::ParentToClient (TRect * *rect*) const

Convert between parent and client coordinates.

Parameters:

rect in: a rect in parent's coordinate system, out:client coordinates

References TRect::GetBottomRight(), and TRect::GetTopLeft().

void TWindow::ClientToParent (TPoint * *point*)

Convert between client and parent coordinates.

Parameters:

point in: client coordinates, out:a point in parent's coordinate system.

void TWindow::ClientToParent (TRect * *rect*)

Convert between parent and client coordinates.

Parameters:

rect in: a rect in parent's coordinate system, out:client coordinates

References TRect::GetBottomRight(), and TRect::GetTopLeft().

void TWindow::GetParentRelativeRect (TRect * *rect*) const

Get the rectangle that represents this window in the client space of its parent window.

Parameters:

rect A rectangle to fill with the window's rectangle in it's parent's coordinate system.

const TRect& TWindow::GetParentRelativeRect () const

Get the rectangle that represents this window in the client space of its parent window.

Returns:

A reference to a TRect that describes this window in its parents coordinates.

virtual void TWindow::SetScroll (float *vScroll*, float *hScroll* = 0) `[virtual]`

A virtual function to override if your window can scroll.

Parameters:

vScroll Vertical scroll ratio (0.0-1.0).
hScroll Horizontal scroll percentage (0.0-1.0).

Reimplemented in TText.

uint32_t TWindow::GetFlags () const

Get the window's state and style flags.

Returns:

Current state and style of the window.

void TWindow::SetFlags (uint32_t *flags*)

Set the state flags of the window.

Does not change the "type" or style flags of the window.

Parameters:

flags Complete set of flags to update.

str TWindow::GetName () const

Get the window name, if any.

Returns:

A str containing the window's name.

void TWindow::SetName (str *name*)

Set the window name.

Parameters:

name New name for the window.

bool TWindow::IsOpaque ()

Query this window's opacity.

Returns:

true if the window is "opaque": When it draws, none of the background will show through. If any part of the window is transparent, it should not have the kOpaque style set.

bool TWindow::IsModal ()

Query this window's modal status.

A modal window blocks further event handling by its parent and receives DoModalProcess() calls.

Returns:

Return true if this is a modal window, false if it is not.

Referenced by IsEnabled(), and IsEnabledSimple().

bool TWindow::IsEnabled ()

Return whether this window and all of its ancestors are enabled.

Returns:

True if this window is really enabled and (eventual) child of a TModalWindow.

References GetParent(), IsModal(), and mFlags.

void TWindow::SendWindowMessage (TMessage * *message*)

Send a message to a window (or its ancestor).

Takes ownership of message and will delete it after it has been delivered.

Calls OnMessage for this window and its parents until one returns "true", indicating the message has been handled. Stops searching at the first modal window.

Parameters:

message Message to send, including potential payload. Will be deleted after delivery.

void TWindow::StartWindowAnimation (int32_t *delay*, bool *autoRepeat* = `true`, bool *resetTime* = `true`, bool *forceFrequency* = `false`)

Start a window animation.

Can be called to reset an animation delay. The virtual function OnTaskAnimate() will be called at the frequency given by the parameters to StartWindowAnimation until StopWindowAnimation is called or this window is destroyed.

This window must already be in a hierarchy and have a parent TModalWindow to attach its animation to, or StartWindowAnimation() will ASSERT (or crash in release build). In other words, you cannot call this function in a constructor.

Parameters:

delay Delay, in ms., before OnTaskAnimate will be called.
autoRepeat True to cause delay to be auto-reset, i.e., to call OnTaskAnimate every delay ms. instead of just once.
resetTime Reset the time after each call. See TAnimTask::SetDelay for details.
forceFrequency Force the animation frequency. See TAnimTask::SetDelay for details.

See also:

TWindow::StopWindowAnimation
TWindow::OnTaskAnimate

void TWindow::StopWindowAnimation ()

Stop a window from receiving OnTaskAnimate calls.

See also:

TWindow::StartWindowAnimation
TWindow::OnTaskAnimate

virtual bool TWindow::OnMessage (TMessage * *message*) `[virtual]`

Handle a message.

Parameters:

message Payload of message.

Returns:

True if message handled; false otherwise.

Reimplemented in TDialog, and TModalWindow.

virtual bool TWindow::OnTaskAnimate () `[virtual]`

Called if you have initiated a window animation with TWindow::StartWindowAnimation.

Returns:

True to continue animating. False to stop.

Reimplemented in TTextEdit.

virtual bool TWindow::OnMouseDown (const TPoint & *point*) `[virtual]`

Mouse down handler.

Parameters:

point Location of mouse press in client coordinates.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented in TButton, TSlider, and TText.

virtual bool TWindow::OnExtendedMouseEvent (const TPoint & *point*, TPlatform::ExtendedMouseEvents *event*) `[virtual]`

Extended mouse button handler.

Parameters:

point Location of mouse event in client coordinates.
event Event that happened.

Returns:

true if message was handled, false to keep searching for a handler.

virtual bool TWindow::OnMouseUp (const TPoint & *point*) `[virtual]`

Mouse up handler.

Parameters:

point Location of mouse release in client coordinates.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented in TButton, TSlider, and TText.

virtual bool TWindow::OnMouseMove (const TPoint & *point*) `[virtual]`

Mouse motion handler.

Parameters:

point Location of mouse in client coordinates.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented in TButton, TSlider, and TText.

virtual bool TWindow::OnMouseLeave () `[virtual]`

Notification that the mouse has left the window.

Warning:

This message is only sent if TWindowManager::AddMouseListener() has been called for this window previously.

Returns:

True if handled.

Reimplemented in TButton, TSlider, and TText.

virtual bool TWindow::CanAcceptFocus () `[virtual]`

Returns true if this window can accept the keyboard focus.

Override to return true if your derived window can accept focus.

Returns:

Return true if this window can accept keyboard focus. If it can accept keyboard focus, it should respond to the On∗Focus() message to update its appearance when its focus state changes.

virtual bool TWindow::OnChar (char *key*) `[virtual]`

Translated character handler.

In UTF-8 mode, this function will only be called if OnUTF8Char() returns false, or there is no OnUTF8Char() handler.

Parameters:

key Key hit on keyboard, along with shift translations.

Returns:

true if message was handled, false to keep searching for a handler.

See also:

TPlatform::kUTF8Mode

Reimplemented in TModalWindow, and TTextEdit.

virtual bool TWindow::OnUTF8Char (str *key*) [virtual]

UTF-8 Translated character handler.

This function is ONLY active when UTF-8 support has been enabled.

Parameters:

key Key hit on keyboard, along with shift translations.

Returns:

true if message was handled, false to keep searching for a handler.

See also:

TPlatform::kUTF8Mode

Reimplemented in TTextEdit.

virtual bool TWindow::OnKeyDown (char *key*, uint32_t *flags*) [virtual]

Raw key hit on keyboard.

Parameters:

key Key pressed on keyboard. Returns either a (low-ASCII) key name, or one of the constants defined in the Key Codes section.

flags TEvent::EKeyFlags mask representing the state of other keys on the keyboard when this key was hit.

Returns:

true if message was handled, false to keep searching for a handler.

Reimplemented in TModalWindow, and TTextEdit.

virtual bool TWindow::OnKeyUp (char *key*) [virtual]

Raw key released on keyboard.

Parameters:

key Key released.

Returns:

true if message was handled, false to keep searching for a handler.

virtual bool TWindow::OnMouseHover (const TPoint & *point*) [virtual]

Called if the mouse hovers over a point on the window.

Parameters:

point Point the mouse was last hovering over.

Returns:

True if processed; false to keep looking.

virtual void TWindow::OnSetFocus (TWindow * *previous*) `[virtual]`

This window is receiving the keyboard focus.

Parameters:

previous The window that was previously focused. Can be NULL.

Reimplemented in TModalWindow.

Referenced by TWindowManager::SetFocus().

virtual void TWindow::OnKillFocus (TWindow * *newFocus*) `[virtual]`

This window is losing the keyboard focus.

Parameters:

newFocus The window that's receiving focus.

bool TWindow::HandleEvent (const TEvent * *eIn*, bool *focusChanged* = `false`)

Internal event handling function.

Warning:

This function should normally only be called by the library.
NOTE it's currently not virtual. If it's ever made virtual, it will likely be VERY important for derived classes to call the base class function.

Parameters:

eIn Event to process.
focusChanged true if this event has already caused a focus change, so that this event will only cause one focus change total.

Returns:

True if the event is handled; false otherwise.

str TWindow::GetID () const

Get a unique window identifier.

Returns:

The ID of the window.

References str::getFormatted().

TWindow* TWindow::SetParent (TWindow * *newParent*) `[protected]`

Set the current parent window.

Parameters:

newParent A TWindow * to the new parent window.

Returns:

A TWindow * to the old parent window.

bool TWindow::IsEnabledSimple () `[protected]`

Return whether this window is enabled, looking at parents, but ignoring whether it descends from a TModalWindow.

Returns:

True if enabled.

References GetParent(), IsModal(), and mFlags.

void TWindow::AddWindowType (uint32_t *type*) `[protected]`

Function that allows a derived window to add type flags to the TWindow.

This isn't public because in most circumstances a window's type should be invariant once created.

Parameters:

type Flags to add.

TAnimTask* TWindow::GetWindowAnim () `[protected]`

Get the associated TAnimTask.

A TAnimTask is associated with this TWindow when StartWindowAnimation() is called.

Will create a new TAnimTask if StartWindowAnimation() hasn't been called yet. In that case, you may want to call SetDelay() on it, or call StartWindowAnimation() with the parameters you want.

Returns:

A pointer to a TAnimTask, or NULL if none exists.

See also:

StartWindowAnimation

15.81 TWindowHoverHandler Class Reference

`#include <pf/windowmanager.h>`

15.81.1 Detailed Description

A callback that receives notification that a window has had the mouse hover over it.

See also:

TWindowManager::AdoptHoverHandler

Public Member Functions

- virtual bool Handle (TWindow *window, const TPoint &point)=0
  Abstract virtual function for handling hover events.

15.81.2 Member Function Documentation

virtual bool TWindowHoverHandler::Handle (TWindow * *window*, const TPoint & *point*) [pure virtual]

Abstract virtual function for handling hover events.

Parameters:

window Window that mouse is hovering over.
point Point in window client coordinates.

Returns:

True if event handled; false to continue processing.

15.82 TWindowManager Class Reference

`#include <pf/windowmanager.h>`

15.82.1 Detailed Description

The TWindowManager class manages, controls, and delegates messages to the window system.

TWindowManager contains a stack of TModalWindows, the top of which is considered to be the active window in the system. If the a TModalWindow covers the entire viewable area and is flagged as opaque, only the top modal window and its children are drawn.

The main message pump hands messages to TWindowManager using TWindowManager::HandleEvent(), which then dispatches the event to the appropriate listener(s).

Construction, Destruction, and Singleton Access

- TWindowManager ()
  Default Constructor.
- virtual ~TWindowManager ()
  Destructor.
- static TWindowManager * GetInstance ()
  Get the global TWindowManager instance.

Public Member Functions

Message handling.

- void PostWindowMessage (TMessage *message)
  Post a message to the queue.
- void AdoptMessageListener (TMessageListener *messageListener)
  Add a message listener that will be able to receive messages that aren't targeted at a specific window.
- bool OrphanMessageListener (TMessageListener *messageListener)
  Remove a message listener from the TWindowManager.

Top-level Screen Related Functions

- class TScreen * GetScreen ()
  Get the global screen object (the top level application TWindow).
- void InvalidateScreen ()
  Mark the current screen as needing to be redrawn.
- void SetScreen (TScreen *screen)

Set the screen to a particular TScreen and initializes it to be the correct size.

Modal window handling and supporting functions.

- void PushModal (TModalWindow *w)
  Push a modal window onto the modal window stack.
- void PopModal (str windowName)
  Pop a modal window off the modal window stack.
- str GetModalReturnStr ()
  Get the return value from a modal window.
- int32_t GetModalReturnInt ()
  Get the return value from a modal window.
- class TDialog * DisplayDialog (str dialogSpec, str body, str title, str name="")
  Display a modal dialog box.
- class TModalWindow * GetTopModalWindow ()
  Get the current top-most modal window.
- void SetTopModalOnly (bool enable)
  Enable "Draw Top Modal Window Only" mode.

Pop-up help handling

Customize the default pop-up help features of your application.

- void AdoptHoverHandler (TWindowHoverHandler *handler)
  Set the current pop-up default help handler.
- TWindowHoverHandler * GetHoverHandler ()
  Get the current pop-up help handler.

Overlay Window management.

An overlay window allows you to draw to a window that lives "on top" of the hierarchy.

- void AdoptOverlayWindow (TWindow *overlay)
  Add an overlay window to the TWindowManager.
- bool OrphanOverlayWindow (TWindow *overlay)
  Release an overlay window from the TWindowManager.

Event Management and Routing

Member functions that manipulate how messages our routed through the system.

- void HandleEvent (TEvent *e)
  Handle a system event.
- void AddMouseListener (TWindow *window)
  Capture the mouse and other input events.

- void RemoveMouseListener (TWindow *window)
  Stop listening to all mouse messages.
- void SetFocus (TWindow *focus)
  Set the window that is to receive the keyboard focus.
- TWindow * GetFocus ()
  Get the window that is currently receiving keyboard events.

Lua GUI Script Access

Functions that access and manipulate the Lua GUI script supplied by TWindowManager.

- TScript * GetScript ()
  Get the current TWindowManager GUI script.
- void RunScript (TWindow *window, const char *filename)
  Use a Lua Script in an external resource to populate a Window.
- void DoLuaString (TWindow *window, str script)
  Use a Lua Script in a str to populate a Window.
- void OnScriptMessage (TMessage *message, TLuaFunction *command=NULL)
  Dispatch a message to the GUI script.
- void AddWindowType (str command, PFClassId classId)
  Add a custom-defined window type to the script context.
- bool EnableStringTable (bool bEnable)
  Toggle on/off using the string table to convert labels found in LUA to properly localized strings (see TStringTable for more information).

Utility Functions

- void AddText (TWindow *window, str bodyText, str style)
  Convenience function that creates a TText child of the given window, using the given bodyText, in the given Lua style.
- void SafeDestroyWindow (TWindow *window)
  Safely destroy a window at the beginning of an event loop.

15.82.2 Member Function Documentation

static TWindowManager* TWindowManager::GetInstance () `[static]`

Get the global TWindowManager instance.

Returns:

A pointer to the TWindowManager.

void TWindowManager::PostWindowMessage (TMessage * *message*)

Post a message to the queue.

Takes ownership of the message, and will expect to be able to delete the message when it has been delivered.

Parameters:

message Message to post.

void TWindowManager::AdoptMessageListener (TMessageListener * *messageListener*)

Add a message listener that will be able to receive messages that aren't targeted at a specific window.

Parameters:

messageListener Listener to adopt. Will be destroyed by TWindowManager unless it is orphaned prior to the destruction of the TWindowManager.

bool TWindowManager::OrphanMessageListener (TMessageListener * *messageListener*)

Remove a message listener from the TWindowManager.

Parameters:

messageListener Listener to remove.

Returns:

True if it was found and removed; false otherwise. If true it's safe to delete, otherwise it's been deleted already.

class TScreen* TWindowManager::GetScreen ()

Get the global screen object (the top level application TWindow).

Returns:

A pointer to the application TScreen.

void TWindowManager::SetScreen (TScreen * *screen*)

Set the screen to a particular TScreen and initializes it to be the correct size.

Also calls Init() on the screen, since no one will be calling AdoptChild() with the TScreen as a parameter.

This function can only be called once in a Playground game. It is called internally by the library when you first call TPlatform::SetDisplay(), so if you want to create a custom-derived TScreen then you need to add it before you call SetDisplay() the first time.

Parameters:

screen Screen to set to be the one-and-only top level screen class.

void TWindowManager::PushModal (TModalWindow * *w*)

Push a modal window onto the modal window stack.

Parameters:

w Window to push.

void TWindowManager::PopModal (str *windowName*)

Pop a modal window off the modal window stack.

Safe to do at any time–window will be deleted at next event.

Parameters:

windowName Name of the window to pop. Will pop that window and any of its descendents from the stack, if found.

str TWindowManager::GetModalReturnStr ()

Get the return value from a modal window.

Returns:

A string return value.

int32_t TWindowManager::GetModalReturnInt ()

Get the return value from a modal window.

Returns:

An integer return value.

class TDialog* TWindowManager::DisplayDialog (str *dialogSpec*, str *body*, str *title*, str *name* = "")

Display a modal dialog box.

Parameters:

dialogSpec Handle to a Lua dialog specification file.
body String to be placed in the dialog body. Selects style DialogBodyText and sets gDialogTable.body to the given body. The Lua dialog specification can then use gDialogTable.body to set the text of the body of the dialog.
title Title of dialog. Selects style DialogTitleText and adds the text to the dialog as gDialogTable.title.
name The name to be given to the resulting dialog window.

Returns:

A pointer to the dialog. The dialog will already have been pushed as the top modal window, but you may need this pointer to set additional fields.

class TModalWindow* TWindowManager::GetTopModalWindow ()

Get the current top-most modal window.

Note that this window may have some number of children–this is just the modal window that is currently receiving the processing.

Returns:

A reference to the top-most modal window.

void TWindowManager::SetTopModalOnly (bool *enable*)

Enable "Draw Top Modal Window Only" mode.

Only the top layer will be drawn when true.

Parameters:

enable True to enable.

void TWindowManager::AdoptHoverHandler (TWindowHoverHandler * *handler*)

Set the current pop-up default help handler.

To add pop-up help to your application, you can either override individual TWindow::OnMouseHover handlers, or you can allow TWindow to call the default handler, which you can set using this function.

The previous handler, if any, is deleted when you call this function.

Parameters:

handler A handler to add.

TWindowHoverHandler* TWindowManager::GetHoverHandler ()

Get the current pop-up help handler.

Returns:

A pointer to the current handler, if any.

void TWindowManager::AdoptOverlayWindow (TWindow * *overlay*)

Add an overlay window to the TWindowManager.

As always, the "Adopt" semantics implies ownership, so when TWindowManager is destroyed, it will attempt to delete this window. To prevent this behavior, call OrphanOverlayWindow to release it from TWindowManager.

An overlay window allows you to draw to a window that lives "on top" of the hierarchy. Set the window rectangle to the area that should be redrawn next frame.

If you want your overlay window to be able to process mouse messages, you need to globally enable the feature by calling:

```C++
TPlatform::SetConfig(TPlatform::kOverlayWindowMouseEvents,"1");
```

...in your program initialization code.

Parameters:

overlay An overlay window to add.

bool TWindowManager::OrphanOverlayWindow (TWindow * *overlay*)

Release an overlay window from the TWindowManager.

Parameters:

overlay Window to release.

Returns:

True if it was found and released. False if it was not found.

void TWindowManager::HandleEvent (TEvent * *e*)

Handle a system event.

Processes the event and passes it along as a message or a callback to the appropriate window. Typically called in the main loop message pump.

Parameters:

e Event.

void TWindowManager::AddMouseListener (TWindow * *window*)

Capture the mouse and other input events.

Implemenation is low-overhead, and so can be safely called in OnMouseMove(). If you request capture a second time with the same window pointer, the new window will not be added to the list of listeners, so a window that wants capture does not need to remember whether it has called AddMouseListener() already–it can just add itself again.

Events are dispatched to all registered mouse listeners, regardless of return values from handled functions.

Parameters:

window Window that wants to receive all mouse events.

void TWindowManager::RemoveMouseListener (TWindow * *window*)

Stop listening to all mouse messages.

Parameters:

window Window to release from capturing the mouse. Silently fails if window is not currently a mouse listener.

void TWindowManager::SetFocus (TWindow * *focus*)

Set the window that is to receive the keyboard focus.

Note that the window will lose the focus if someone clicks unless it has the style kFocusTarget. On construction of a window that is to receive the focus, call

```C++
AddWindowType( kFocusTarget );
```

...and this will flag that window as being able to accept focus when clicked. Otherwise focus goes it its nearest kFocusTarget ancestor, or is delegated to the default-focus defined by the window's parent modal.

Parameters:

focus New focus window.

References TWindow::OnSetFocus().

TWindow* TWindowManager::GetFocus ()

Get the window that is currently receiving keyboard events.

Returns:

A reference to the current focused window.

TScript* TWindowManager::GetScript ()

Get the current TWindowManager GUI script.

Returns:

A pointer to the current script.

void TWindowManager::RunScript (TWindow * *window*, const char * *filename*)

Use a Lua Script in an external resource to populate a Window.

Parameters:

window Window to apply script to.
filename Name of Lua file.

void TWindowManager::DoLuaString (TWindow * *window*, str *script*)

Use a Lua Script in a str to populate a Window.

Parameters:

window Window to apply script to.
script Lua commands to run.

void TWindowManager::OnScriptMessage (TMessage * *message*, TLuaFunction * *command* = NULL)

Dispatch a message to the GUI script.

Parameters:

message Message to pass to Lua.
command Command to pass to Lua to execute in GUI thread.

void TWindowManager::AddWindowType (str *command*, PFClassId *classId*)

Add a custom-defined window type to the script context.

Parameters:

command Window type name.
classId The TWindow::ClassId() of the custom defined window. Note that the window needs to have PFTYPEDEF_DC() in the header and PFTYPEIMPL_DC() in the implementation file for this to work.

bool TWindowManager::EnableStringTable (bool *bEnable*)

Toggle on/off using the string table to convert labels found in LUA to properly localized strings (see TStringTable for more information).

Parameters:

bEnable true to use the table, false to not use the table

Returns:

returns true if string table was previously enabled, false otherwise

void TWindowManager::AddText (TWindow * *window*, str *bodyText*, str *style*)

Convenience function that creates a TText child of the given window, using the given bodyText, in the given Lua style.

Parameters:

window The window to add a new TText child to.
bodyText Text to add.
style Name of the Lua style to use (must already be loaded in the TWindowManager::GetScript() Lua script), or a style definition in curly brackets.

void TWindowManager::SafeDestroyWindow (TWindow * *window*)

Safely destroy a window at the beginning of an event loop.

Allows you to mark a window for destruction when processing an event that may need to continue accessing the window.

Parameters:

window Window to destroy.

15.83 TWindowSpider Class Reference

`#include <pf/window.h>`

15.83.1 Detailed Description

A class used with TWindow::ForEachChild to iterate over the children of a window with a single "callback" function.

Public Member Functions

- virtual ~TWindowSpider ()
  Virtual destructor.
- virtual bool Process (TWindow *window)=0
  The function called once for each window.

15.83.2 Member Function Documentation

virtual bool TWindowSpider::Process (TWindow * *window*) `[pure virtual]`

The function called once for each window.

Parameters:

window The window being iterated.

Returns:

True to continue the traversal.

15.84 TWindowStyle Class Reference

`#include <pf/windowstyle.h>`

15.84.1 Detailed Description

An encapsulation of a Lua window style.

Public Types

Local Types

- enum {
  kCenter = 80000, kMax = 160000, kHAlignLeft = 0, kHAlignCenter = 1,
  kHAlignRight = 2, kVAlignTop = 0, kVAlignCenter = 4, kVAlignBottom = 8,
  kDefault = 128, **kPushButtonAlignment** = kHAlignCenter+kVAlignCenter, **kRadioButtonAlignment** = kHAlignLeft+kVAlignCenter, **kToggleButtonAlignment** = kHAlignLeft+kVAlignCenter }
  Various window constants.

Public Member Functions

- TWindowStyle (TLuaTable *table)
  Construction.
- virtual ~TWindowStyle ()
  Destructor.
- str GetString (str key, str defaultValue="") const
  Get a string parameter from the style.
- double GetNumber (str key, double defaultValue=0) const
  Get a numeric parameter from the style.
- bool GetBool (str key, bool defaultValue=false) const
  Get a boolean parameter from the style.
- TColor GetColor (str key, TColor defaultValue=TColor(0, 0, 0, 1)) const
  Get a color parameter from the style.
- TLuaFunction * GetFunction (str key) const
  Get a Lua function closure from the style.
- TLuaTable * GetTable (str key)
  Get a Lua table from the style.
- int32_t GetInt (str key, int32_t defaultValue=0) const
  Get an integer parameter from the style.

15.84.2 Member Enumeration Documentation

anonymous enum

Various window constants.

Enumerator:

kCenter Select center for a coordinate.
kMax Select max for a width or height.
kHAlignLeft Align text to the left.
kHAlignCenter Align text to the center.
kHAlignRight Align text to the right.
kVAlignTop Align text to the top.
kVAlignCenter Align text vertically to the center.
kVAlignBottom Align text to the bottom.
kDefault Default text alignment.

15.84.3 Member Function Documentation

str TWindowStyle::GetString (str *key*, str *defaultValue* = `""`) const

Get a string parameter from the style.

Parameters:

key Name of the parameter to query.
defaultValue Default if parameter value not found.

Returns:

Value if found; defaultValue otherwise.

double TWindowStyle::GetNumber (str *key*, double *defaultValue* = `0`) const

Get a numeric parameter from the style.

Parameters:

key Name of the parameter to query.
defaultValue Default if parameter value not found.

Returns:

Value if found; defaultValue otherwise.

bool TWindowStyle::GetBool (str *key*, bool *defaultValue* = `false`) const

Get a boolean parameter from the style.

Parameters:

key Name of the parameter to query.

defaultValue Default if parameter value not found.

Returns:

Value if found; defaultValue otherwise.

TColor TWindowStyle::GetColor (str *key*, TColor *defaultValue* = TColor(0, 0, 0, 1)) const

Get a color parameter from the style.

Parameters:

key Name of the parameter to query.
defaultValue Default if parameter value not found.

Returns:

Value if found; defaultValue otherwise.

TLuaFunction* TWindowStyle::GetFunction (str *key*) const

Get a Lua function closure from the style.

You will need to delete the function when you're done with it.

Parameters:

key Name of the parameter to query.

Returns:

Function if found; NULL otherwise.

TLuaTable* TWindowStyle::GetTable (str *key*)

Get a Lua table from the style.

You will need to delete the table when you're done with it.

Parameters:

key Name of the parameter to query.

Returns:

Table if found; NULL otherwise.

int32_t TWindowStyle::GetInt (str *key*, int32_t *defaultValue* = 0) const

Get an integer parameter from the style.

Parameters:

key Name of the parameter to query.
defaultValue Default if parameter value not found.

Returns:

Value if found; defaultValue otherwise.

15.85 TXmlNode Class Reference

`#include <pf/simplexml.h>`

15.85.1 Detailed Description

The TXmlNode class is a limited XML parser.

It does not support comments with nested tags, nor most non-trivial XML extensions.

Any XML files in your assets folder will be obfuscated by default (in addition to being included in the flat file) in a production build, so if you need them to remain human-readable, let your producer know.

Public Member Functions

- TXmlNode ()
  Default constructor.
- TXmlNode (const char *name)
  Create this node with a name.
- virtual ~TXmlNode ()
  Destructor.
- uint32_t ParseStream (const char *data, uint32_t len, bool bOneTag=false)
  Parse a stream as XML, loading contents as children of this node.
- void ParseString (const char *data)
  Parse a string as XML, loading contents as children of this node.
- void ParseFile (const char *filename)
  Parse a file as XML, loading contents as children of this node.
- void Clear ()
  Remove all children and attributes.
- bool HasChildren () const
  Query whether this node has children.
- TXmlNode * GetChild (const char *name)
  Get a pointer to a child of the node.
- void OrphanChild (TXmlNode *child)
  Remove a child from the parent and take ownership.
- void DeleteChild (TXmlNode *child)
  Delete a child from this node.

- void ResetChildren ()

  Reset internal child iterator.

- bool GetNextChild (str *pName, TXmlNode **pChild)

  Iterate through children.

- TXmlNode * CreateChild (const char *name)

  Create a new child and add it to our list of children.

- void SetName (const char *name)

  Set the name of this node.

- str GetName () const

  Get the name of this node.

- str GetContent ()

  Get the content of this node (the part between the opening and closing tags).

- void SetContent (const char *content)

  Set the content of this node.

- str GetAttribute (const char *name, bool *pbQuoted=NULL)

  Get an attribute of this node's tag.

- void SetAttribute (const char *name, const char *value)

  Set an attribute to a particular value.

- void SetAttribute (const char *name, int32_t value)

  Set an attribute to a particular value.

- void ResetAttributes ()

  Reset the internal attribute iterator.

- bool GetNextAttribute (str *pName, str *pValue)

  Get the next attribute in an iteration.

- str GetChildContent (const char *name)

  Get the content of a child node.

- str AsString ()

  Parse the XML tree into an XML formatted string that can be written to a file.

15.85.2 Constructor & Destructor Documentation

TXmlNode::TXmlNode (const char * *name*)

Create this node with a name.

Parameters:

name Initial name for the node.

15.85.3 Member Function Documentation

uint32_t TXmlNode::ParseStream (const char * *data*, uint32_t *len*, bool *bOneTag* = `false`)

Parse a stream as XML, loading contents as children of this node.

Does not consume all data. Consumes a balanced open/close tag (and everything in between)

Parameters:

data buffer to parse
len length of buffer
bOneTag normally false, set to true to consume just one tag, not a balanced open/close

Returns:

Number of Characters consumed from the buffer - will be 0, if not enough data available

void TXmlNode::ParseString (const char * *data*)

Parse a string as XML, loading contents as children of this node.

Parameters:

data String to parse.

void TXmlNode::ParseFile (const char * *filename*)

Parse a file as XML, loading contents as children of this node.

Parameters:

filename File to read.

bool TXmlNode::HasChildren () const

Query whether this node has children.

Returns:

True if has children.

TXmlNode* TXmlNode::GetChild (const char * *name*)

Get a pointer to a child of the node.

Parameters:

name The name of the child to find.

Returns:

A pointer to the child. The child is still owned by the parent, so this pointer will become invalid when the parent is deleted.

void TXmlNode::OrphanChild (TXmlNode * *child*)

Remove a child from the parent and take ownership.

Parameters:

child A pointer to the child.

void TXmlNode::DeleteChild (TXmlNode * *child*)

Delete a child from this node.

Parameters:

child Child to delete.

void TXmlNode::ResetChildren ()

Reset internal child iterator.

See also:

GetNextChild

bool TXmlNode::GetNextChild (str * *pName*, TXmlNode ** *pChild*)

Iterate through children.

MUST call ResetChildren() to start iteration.

Parameters:

pName Pointer to str to receive name of child that was found
pChild Pointer to receive next child in iteration.

Returns:

TXmlNode* TXmlNode::CreateChild (const char * *name*)

Create a new child and add it to our list of children.

Parameters:

name Name of the new child.

Returns:

A pointer to the new child.

void TXmlNode::SetName (const char * *name*)

Set the name of this node.

Parameters:

name New name.

str TXmlNode::GetName () const

Get the name of this node.

Returns:

The node's name.

str TXmlNode::GetContent ()

Get the content of this node (the part between the opening and closing tags).

Returns:

The node content.

void TXmlNode::SetContent (const char * *content*)

Set the content of this node.

Parameters:

content New content.

str TXmlNode::GetAttribute (const char * *name*, bool * *pbQuoted* = `NULL`)

Get an attribute of this node's tag.

Parameters:

name Name of attribute to query.
pbQuoted Whether the attribute's value is surrounded by quotes.

Returns:

The value of the attribute, if found. Otherwise an empty string.

void TXmlNode::SetAttribute (const char * *name*, const char * *value*)

Set an attribute to a particular value.

Parameters:

name Name of the attribute to set.
value New value for that attribute.

void TXmlNode::SetAttribute (const char * *name*, int32_t *value*)

Set an attribute to a particular value.

Parameters:

name Name of the attribute to set.
value New value for that attribute.

bool TXmlNode::GetNextAttribute (str * *pName*, str * *pValue*)

Get the next attribute in an iteration.

Call ResetAttributes() to reset the iteration.

Parameters:

pName [return] Name of the attribute.
pValue [return] Value of the attribute.

Returns:

True if another attribute was found.

str TXmlNode::GetChildContent (const char * *name*)

Get the content of a child node.

Parameters:

name Node of child to find.

Returns:

Child content.

str TXmlNode::AsString ()

Parse the XML tree into an XML formatted string that can be written to a file.

Returns:

The data as a string.

15.86 debug.h File Reference

15.86.1 Detailed Description

Playground debug and error utility routines.

Defines

- #define ERROR_WRITE(OUT_STATEMENT)

  In all builds, this function will write the contained printf-style string and parameters to the debug log, and also to the debug monitor in debug builds.

- #define **pfDebugBreak**()
- #define TRACE_WRITE()

  When INCLUDE_TRACE_STATEMENTS in debug.h is set to 1, this function will output the name of the function, file, and line to the debug log.

- #define ASSERT(STATEMENT)

  ASSERT that a statement is returning non-zero.

- #define VERIFY(STATEMENT) (void)(STATEMENT)

  Verify that a statement is returning non-zero.

- #define DEBUG_WRITE(OUT_STATEMENT)

  In debug builds (when _DEBUG is 1), this function will send the contained printf-style string and parameters to the debug log, and to the debug monitor.

- #define VERBOSE_ERROR(X)

  This macro will write an error string to the debug log in debug and release builds if you enable verbose debugging by calling TPlatform::SetConfig(TPlatform::kVerboseDebug);.

- #define VERBOSE_DEBUG(X)

  This macro will write an error string to the debug log in debug builds if you enable verbose debugging by calling TPlatform::SetConfig(TPlatform::kVerboseDebug);.

- #define VERBOSE_TRACE()

  This macro will write an error string to the debug log in debug builds indicating the name of the function, file, and line to the debug log if you enable verbose debugging by calling TPlatform::SetConfig(TPlatform::kVerboseDebug);.

15.86.2 Define Documentation

#define ASSERT(STATEMENT)

ASSERT that a statement is returning non-zero.

In builds with INCLUDE_DEBUG_STATEMENTS set to zero (by default any non-debug build), STATEMENT is NOT executed.

Parameters:

STATEMENT Statement to test.

Referenced by TParticleState::GetReal(), TParticleMachineState::GetSize(), TParticleMachineState::Param(), TScript::PopBool(), TParamSet::ResetPartial(), and TLuaTable::TLuaTable().

#define DEBUG_WRITE(OUT_STATEMENT)

In debug builds (when _DEBUG is 1), this function will send the contained printf-style string and parameters to the debug log, and to the debug monitor.

```
DEBUG_WRITE(("Format␣String␣%d",5));
```
C++

All printf format codes are supported.

Note you need to enclose the parameters in a second pair of parenthesis:

```
DEBUG_WRITE(("Format␣String␣%d",5));
```
C++

All printf format codes are supported.

#define ERROR_WRITE(OUT_STATEMENT)

In all builds, this function will write the contained printf-style string and parameters to the debug log, and also to the debug monitor in debug builds.

Note you need to enclose the parameters in a second pair of parenthesis:

```
ERROR_WRITE(("Format␣String␣%d",5));
```
C++

All printf format codes are supported.

Referenced by TScript::PopBool().

#define VERBOSE_DEBUG(X)

This macro will write an error string to the debug log in debug builds if you enable verbose debugging by calling TPlatform::SetConfig(TPlatform::kVerboseDebug);.

See also:

DEBUG_WRITE

#define VERBOSE_ERROR(X)

This macro will write an error string to the debug log in debug and release builds if you enable verbose debugging by calling TPlatform::SetConfig(TPlatform::kVerboseDebug);.

See also:

DEBUG_WRITE

#define VERBOSE_TRACE()

This macro will write an error string to the debug log in debug builds indicating the name of the function, file, and line to the debug log if you enable verbose debugging by calling TPlatform::SetConfig(TPlatform::kVerboseDebug);.

See also:

DEBUG_WRITE

#define VERIFY(STATEMENT) (void)(STATEMENT)

Verify that a statement is returning non-zero.

Acts like ASSERT, only STATEMENT is always included in the program.

Parameters:

STATEMENT Statement to test.

15.87 pftypeinfo.h File Reference

15.87.1 Detailed Description

Runtime type information handling support macros.

This file contains a number of macros that add a more flexible runtime-type information facility than is supported natively by C++. For any class you want to decorate with additional runtime information, you'll need one call in the class definition, with a corresponding call in the implementation file. The base class definition has its own call as well.

The basic form is PFTYPEDEF() in the class, PFTYPEIMPL() in the implementation file, with the base class using PFTYPEDEFBASE(). This set would be used in classes that do not need dynamic creation or to be stored in shared pointers.

The PFSHAREDTYPEDEF() variation is used in classes that will be held in shared pointers. It requires that you use PFSHAREDTYPEDEFBASE() in the base class, and that the base class be derived from enable_shared_from_this<BASECLASS>. In the implementation files you can use PFTYPEIMPL() with PFSHAREDTYPEDEF() or PFSHAREDTYPEDEFBASE().

The _DC variations are similar to the ones described above, only they also support dynamic creation by defining a function CreateFromId() that takes the ClassId() of a class and creates a new instance of the class. PFTYPEDEF_DC() matches PFTYPEIMPL_DC(). The PFTYPEIMPL_DCA() variation needs to be used for any abstract class.

See also:

Type Information and Casting

Defines

- #define PFTYPEDEF(THISCLASS, BASECLASS)

  Additional definitions for a class that needs run time type information.

- #define PFTYPEDEF_DC(THISCLASS, BASECLASS)

  Additional definitions for a class that needs run time type information and dynamic creation.

- #define PFSHAREDTYPEDEF(BASECLASS)

  Additional definitions for a class that needs run time type information, and that is controlled through shared pointers.

- #define PFSHAREDTYPEDEF_DC(THISCLASS, BASECLASS)

  Additional definitions for a class that needs run time type information and dynamic creation, and that is controlled through shared pointers.

- #define PFTYPEDEFBASE()

  Additional definitions for the base class of a polymorphic type that needs run time type information.

- #define PFTYPEDEFBASE_DC(THISCLASS)

  Additional definitions for the base class of a polymorphic type that needs run time type information and dynamic creation.

- #define PFSHAREDTYPEDEFBASE(THISCLASS)

  Additional definitions for the base class of a polymorphic type that needs run time type information, and that will be stored in shared pointers (reference counted, like TTextureRef).

- #define PFSHAREDTYPEDEFBASE_DC(THISCLASS)

  Additional definitions for the base class of a polymorphic type that needs run time type information and dynamic creation, and that will be stored in shared pointers (reference counted, like TTextureRef).

- #define PFTYPEIMPL(THISCLASS)

  Implementation for run time type information.

- #define PFTYPEIMPL_DC(THISCLASS)

  Implementation for run time type information for a class with dynamic creation.

- #define PFTYPEIMPL_DCA(THISCLASS)

  Implementation for run time type information for an abstract class, descendents of which will require dynamic creation.

15.87.2 Define Documentation

#define PFSHAREDTYPEDEF(BASECLASS)

Additional definitions for a class that needs run time type information, and that is controlled through shared pointers.

Putting this macro in a class will declare and define IsKindOf(), GetCast() and ClassId() for that class.

Parameters:

BASECLASS Our base class.

See also:

Type Information and Casting

#define PFSHAREDTYPEDEF_DC(THISCLASS, BASECLASS)

Additional definitions for a class that needs run time type information and dynamic creation, and that is controlled through shared pointers.

Putting this macro in a class will declare and define DynamicCreate(), IsKindOf(), GetCast() and ClassId() for that class.

Parameters:

THISCLASS The class we're defining.
BASECLASS The (single) base class.

See also:

Type Information and Casting

#define PFSHAREDTYPEDEFBASE(THISCLASS)

Additional definitions for the base class of a polymorphic type that needs run time type information, and that will be stored in shared pointers (reference counted, like TTextureRef).

Putting this macro in a class will declare and define IsKindOf(), GetCast() and ClassId() for that class.

Parameters:

THISCLASS The current class to declare.

#define PFSHAREDTYPEDEFBASE_DC(THISCLASS)

Additional definitions for the base class of a polymorphic type that needs run time type information and dynamic creation, and that will be stored in shared pointers (reference counted, like TTextureRef).

Putting this macro in a class will declare and define IsKindOf(), GetCast() and ClassId() for that class.

#define PFTYPEDEF(THISCLASS, BASECLASS)

Additional definitions for a class that needs run time type information.

Putting this macro in a class will declare and define IsKindOf(), GetCast() and ClassId() for that class.

See also:

Type Information and Casting

#define PFTYPEDEF_DC(THISCLASS, BASECLASS)

Additional definitions for a class that needs run time type information and dynamic creation.

Putting this macro in a class will declare and define DynamicCreate(), IsKindOf(), GetCast() and ClassId() for that class.

See also:

Type Information and Casting

#define PFTYPEDEFBASE()

Additional definitions for the base class of a polymorphic type that needs run time type information.

Putting this macro in a class will declare and define IsKindOf(), GetCast() and ClassId() for that class.

#define PFTYPEDEFBASE_DC(THISCLASS)

Additional definitions for the base class of a polymorphic type that needs run time type information and dynamic creation.

Putting this macro in a class definition file will declare and define IsKindOf(), GetCast() and ClassId() for that class.

Parameters:

THISCLASS The current class to declare.

#define PFTYPEIMPL(THISCLASS)

Implementation for run time type information.

Use this macro in the implementation file.

#define PFTYPEIMPL_DC(THISCLASS)

Implementation for run time type information for a class with dynamic creation.

Use this macro in the implementation file.

#define PFTYPEIMPL_DCA(THISCLASS)

Implementation for run time type information for an abstract class, descendents of which will require dynamic creation.

Use this macro in the implementation file.

15.88 pflibcore.h File Reference

15.88.1 Detailed Description

Include file that PFLIB requires you load before anything else.

Functions

- void PlaygroundInit ()
  A user-defined function that will be called by Playground before TPlatform is initialized.

15.88.2 Function Documentation

void PlaygroundInit ()

A user-defined function that will be called by Playground *before TPlatform is initialized*.

No non-static calls on TPlatform are allowed.

In order for this function to be called, you must define PLAYGROUND_INIT in the file where you're defining PLAYFIRST_MAIN, before you include pflibcore.h.

This function is really designed only for setting up values with TPlatform::SetConfig() that TPlatform will need during initialization; most application initialization still belongs in Main().

Part III

Appendix

Appendix A

Forward Declarations

A.1 forward.h File Reference

A.1.1 Detailed Description

Forward declarations for Playground.

Typedefs

- typedef shared_ptr< TTexture > TTextureRef
  A reference to a TTexture.
- typedef shared_ptr< TAnimatedTexture > TAnimatedTextureRef
  A reference to a TAnimatedTexture.
- typedef shared_ptr< TSprite > TSpriteRef
  A reference to a TSprite.
- typedef shared_ptr< TTextSprite > TTextSpriteRef
  A reference to a TTextSprite.
- typedef shared_ptr< TAnimatedSprite > TAnimatedSpriteRef
  A reference to a TAnimatedSprite.
- typedef shared_ptr< TScriptCode > TScriptCodeRef
  A reference to a TScriptCode object.
- typedef shared_ptr< TSound > TSoundRef
  A reference to a TSound.
- typedef shared_ptr< TSoundInstance > TSoundInstanceRef
  A reference to a TSoundInstance.

- typedef shared_ptr< TModel > TModelRef

  A reference to a TModel.

- typedef shared_ptr< TAsset > TAssetRef

  A reference to a TAsset.

Appendix B

Change History

B.1 Playground SDK Change Log

B.1.1 New Features in Playground 4.0.22

- Cleaned up and added documentation. In particular, added documentation for how flat file support works in Playground.
- FluidFX and FirstStage gain the ability to switch resolutions.
- Added support for command line to FirstStage.
- Added FirstStage to the build package of Playground.
- Added compile warning to Mac that will fire when a float is unintentionally converted to an integer; on Windows, abs() is defined overloaded for ints and floats, while on Mac it's defined for ints only, so on Mac it would lose precision.

B.1.2 Fixes in Playground 4.0.22

- Fixed Mac build to use kPublisherName and kGameName in the creation of paths. You need to use TPlatform::SetConfig("mac_fix_common_and_user_dirs", "1"); to enable this new behavior.
- Fix a problem found on the Dell D610 where game would hang on Alt-Tab away and back to the game.
- Fixed TTextEdit to not crash when a game creates one and never calls TTextEdit::Create().
- Fix Flash-playback when game is installed to a path with extended characters.
- Fix Vista-only Web game bug with render targets on some video cards.
- Added a missing license clause.
- Added a define pfDebugBreak() in non-debug builds, so you can still call pfDebugBreak() from a release build and it simply won't do anything.

- Fixed a crash bug in sidewalk.
- Removed references to things were were "going to do" in 2007 from the docs.
- Fixed firstpeek.pfmod to properly quote an value that contains commas, even if the value starts with a number.
- Fixed project to create three distinct .pdb files, one for each of the build types. This should allow profiling and debugging to include more specific information.

B.1.3 New Features in Playground 4.0.21

- sidewalk now has an improved rectangle packing algorithm that is more intelligent about packing dissimilar rectangles into a combined atlas texture. sidewalk also has a vastly improved image comparison algorithm, and will optimize the JPG and JP2 compression of each file individually until it reaches a given quality threshold. The new algorithm has the advantage of being much faster (typically) than the old algorithm.
- FluidFX now has a graph-editor for modifying particle fade sequences.
- FilmStrip now gives you better information on a failure to load or save to help you know what might be wrong.
- FirstStage now has a tutorial section of the documentation.

B.1.4 Fixes in Playground 4.0.21

- Fixed a bug where one DisplaySplash() followed immediately by another could trigger a bad state if you attempted to exit the game during the first movie. **Note that your game may need to update its kClose handling to match the current skeleton's in order to avoid other variations of this bug.**
- Removed obsolete THiscore::OnDirtyRect from Playground Skeleton app.
- Fixed TTextEdit tab functionality when you have more than two TTextEdit objects in a window.
- Fixed TTextEdit tab functionality when one (or more) of your TTextEdit windows is disabled.
- Fixed word wrapping bug where sometimes a character in the middle of a string would break the word, keeping a fragment (up to that character) on the previous line, but still copying it onto the next line. This typically only appeared in translations, since it was only triggered by a non-Latin character being missing from a fancy font and replaced by an alternate character in the default font.
- Fixed Flash playback on Mac if the user navigates away from the movie.
- FilmStrip now correctly triggers a "save" on Ctrl-S instead of "save as".

B.1.5 New Features in Playground 4.0.20

- TImage can now take a new lua flag - 'autoscale'. If true, it will scale the TImage to screen size. It Will NOT change aspect ratio of image if it is different from the screen. DisplaySplash now sets autoscale to true, this is so that splash screens will auto scale to the right screen resolution.

- In the Playground Skeleton, change the medals screen behavior so that the screen shows a "resubmit" button if the user has previously submitted medals, but no new medals.
- In the Playground Skeleton, change the high scores screen behavior so that the screen shows a "resubmit" button if the user has previously submitted high score.
- Added TPfHiscores::ClearPlayerData. Updated Playground Skeleton so that TSettings::DeleteUser calls ClearPlayerData(). This is a new requirement in the PlayFirst Production Guidelines.
- Added TPlatform::kLargeVRAMUsage config flag. If set, this will force certain graphics cards to run in 16-bit mode to avoid driver issues that occur when large amounts of VRAM are used.
- New sidewalk texture packing algorithm packs textures far more efficiently, usually resulting in smaller textures.
- pfhs.lib for non-playground developers so that they can submit medals and scores to pf.com

B.1.6 Fixes in Playground 4.0.20

- Properly save medal state after a medal submission with TPfHiscores.
- Fixed auto-merge again; it somehow broke between a combination of changes.
- Update the xsell.lua file in the xsell addon so that it can be used in games that run in different resolutions. In games larger than 800x600, it will center the xsell window in the screen.
- xsell/xsell.xml is not an xsellwindow config parameter instead of being hard-coded.
- Updated xsell window button and text positioning
- Fix for QueryMuteGame() to work properly with MSN/Zylom
- Fixed flash playback bug on Windows where alt-tabbing out of a Flash movie didn't always restore the movie to the right size if the desktop was set to a resolution smaller than the resolution of the game.
- Added another card to the list of Intel cards with bad drivers: The Intel 950, detected as the 82945.
- Made FluidFX work with the fDebug() command, to allow for particle system debug printing.
- Fixed a couple of bugs with the way hotkeys worked in the skeleton, and one bug with how hotkeys work within Playground.
- Fixed mac flash crash bug.
- Fix for hotkey and OnChar messages both being sent by keypresses.
- Work on Jasper color-space handling
- Software texture-culling bug fix
- Fix to allow unicode in mac window titles

B.1.7 New Features in Playground 4.0.19

- Optional support for improved locale-aware string sorting. See str documentation.
- Added TPlatform::kSoftwareTextureCulling, which enables Playground software culling to help performance on graphics cards that don't do a fast job of culling.
- Added TLuaParticleSystem::GetRenderer() so that the built-in renderer could be accessed.

B.1.8 Fixes in Playground 4.0.19

- Fixed issue on some Intel graphics cards that can't handle alternating between DrawIndexedPrimitive and DrawPrimitive by forcing all calls on these cards through DrawIndexedPrimitive.
- Fixed issue in full screen mode when hardware cursors are used where game could hang.
- Fixed issue where calling TFile::AddFileMask would cause the loss of the default cursor image.
- TGameState is instantiated before the game main loop is called, so that the Lua function will be registered, and it won't crash if the first call to SetState() is in Lua.
- File access on Windows now can handle international characters in all path names as well as in the product name.
- Fixed a crash on really short application window names on Windows.
- Added missing TAnimatedSprite::GetScriptCode() implementation.
- Added SetConfig() option TPlatform::kFixButtonMaskScaling that fixes the mask scaling behavior in TButton.
- Fix a crash if you have a texture larger than the actual screen surface; add a warning in the debug log in that case.
- Fix a bug in merged render targets that prevented the merge from copying past the screen height in width.
- Fix skeleton to properly respond to escape during the games.
- Fixed a bug in the font rendering where a tag, if it preceeded a word that was line wrapped to the next line, would not end correctly at the .
- Calling TPfHiscores::SubmitData with submitMode == kSubmitMedal will now submit medals even if they have all been previously submitted.
- Fixed a bug where TDialog::NewParent() wasn't calling its base class, and therefore wasn't always initializing itself correctly.
- Fixed a Mac-only bug where the font cache wasn't working as expected.
- Fixed a Mac bug where TSound::KillInstance() could occasionally crash when called with the sound in an active update in the background thread.
- Fix for SiS Mirage 3, which has a bug in its DrawIndexedPrimitive() call in the DirectX 7 drivers.
- Fix in skeleton of TFxSprite, preventing it from crashing if it doesn't have a parent.
- Fix in sidewalk of animated gif reading.
- Fix skeleton application dialogs that contain "Privacy Notice" links; the text boxes were too tall and not releasing mouse capture to the buttons below.

B.1.9 New Features in Playground 4.0.18

- There's a new SetConfig() option, TPlatform::kKeepOGGUncompressed, that enables complete decompression of ogg files at file load, rather than decompressing them as needed during playback.

B.1.10 Fixes in Playground 4.0.18

- Deleting multiple mouse listeners in a mouse listener callback is now supported (and won't crash!).
- Deleting a thread off of the main script will no longer crash.
- Fixed some const issues in TWindow.
- Fixed a false memory leak when using TPrefsDB.
- Fixed TTexture::Save() and BeginRenderTarget() to correctly work on a TAnimatedTexture.
- Fixed a bug in TLuaParticleSystem that limited the number of parameters you can use.
- Major changes to the audio implementation on the mac. Migrated to OpenAL and a multithreaded, double-buffered solution. OpenAl is included in the playground distribution and so it does not need to be linked with any game using it. Games will, however, need to link with CoreAudio.framework, AudioUnit.framework and AudioToolbox.framework.
- General documentation improvements.
- The TPfHiscores system now sends data to the server using POST.

B.1.11 New Features in Playground 4.0.17

- Playground now optionally supports Speex compression. To enable Speex compression in sounds, include pf/speex.h, and then call TSpeex::Register(). Behind the scenes, the code that reads data from a sound file has been cleaned up a lot.
- TRenderer::SetColorWrite() enables or disables the writing of color values (as opposed to z-depth values).
- There's a new SetConfig() option, TPlatform::kFpuPrecision, that enables high-precision floating point support.
- For Web games, you can now query GetConfig(TPlatform::kDownloadURL) to get the download URL already parsed for you from the command line.
- TLuaParticleSystem: You can now index a value, including a return result from a function. For example, you can now do this: pPosition:Anim(dLocus + Vec2(0,f2dRotation(pAge/100)[2]∗100));
- TPrefsDB: A new database-based preferences system. Works as a drop-in replacement for TPrefs, but can update a portion of the database instead of the entire file.
- A new command line option, –stringtest, will turn all the non-space characters extracted from the string table to "x" chararacters. This allows QA to go through a game and verify that all strings used in the game are in the string table–any strings that aren't entirely "x" aren't being translated.
- A new option, TPlatform::kSpriteAlwaysTestChildren, that changes TSprite::HitTest() to work as it might be expected: By default it silently fails if the TSprite has no texture attached, even if the children do have textures. With this option set, it will always HitTest() children regardless of whether it has a texture.

- TSlider has a new "value" tag that can be used from Lua to set the initial value of the slider.
- Added new config setting, TPlatform::kAdjustAnimatedParticles. If this is set, then the particle renderer will correctly position animation frames when a TAnimatedTexture is used as the texture in a particle system.
- sidewalk now leaves the extension off of the image in .xml files it creates, so that sidewalk's crunch mode can change the image types in-place without having to modify the .xml animation files.
- Lots of new documentation for the Playground tools, and lots of fixes and clarifications in the main docs.
- Lots of new features and fixes for Filmstrip, FluidFX, and sidewalk. Play with the tools to find the new features!
- Support for reading JPEG2000 images (with extension .jp2); sidewalk crunch mode support for writing .jp2 images.
- Added Lua command, "toparams(t)", that takes table "t" and returns it as a parameter list.

B.1.12 Fixes for Playground 4.0.17

- Fixed a lot of accumulated bugs in the PDF version of the docs.
- Fix an occasional spurious failure of BeginRenderTarget().
- Fix key events to be signed, so they behave consistently between OnKeyDown() and GetEvent().
- sidewalk: Fix –scale for animations where the registration point is outside of **all** frames of the animation.
- Fix TFlashHost so that if you start a movie with the mouse button down, it will still receive events.
- Fixed problem where a 1024x768 game in windowed mode would end up off-screen on Vista at 1280x800.
- Fixed changelog for 4.0.16 below; added missing FilmStrip and FluidFX features, and documented the patch.
- When submitting medals, the high scores system will now re-transmit all medal info to the server, even if medals have previously been marked as submitted.
- Fix of Mac font calibration.
- Fix of the skeleton yesno.lua to allow nested yesno.lua dialog boxes.
- Fix of TFxSprite to properly clean up after the last TFxSprite has been destroyed.
- Fix a spurious ASSERT on Mac.
- Fix of a bug in new TPrefsDB that caused the data to sometimes not be saved in release builds.
- Fix of alt-tab-away-from-fullscreen bug.
- Fix new TGameState MSN/Zylom hooks and improved documenation to make the usage more clear.
- Added a default icon to Mac builds to make it easier to replace it with a custom icon.
- Fix a crash-on-exit if you forget to release a TSprite or TTexture.

B.1.13 Migration to Playground 4.0.16

- If you were previously counting on values OnKeyDown() reported, but that weren't defined by Playground (i.e., constants other than 'A'-'Z', '0'-'9', and those defined in TEvent), many of them are going to change in this release. In particular, keys like ',', ';', and numeric pad keys will now all report as their unshifted variant.

B.1.14 New Features in Playground 4.0.16

- A new option that can be enabled with TPlatform::SetConfig(): kAutoMergeMask, which allows you to specify the name (or root) of a .jpg file, and it will automatically merge a .mask.png file with the same root name as a combined alpha file. This allows you to use maximum compression for the image layer, while keeping the mask layer lossless.
- More public accessors: TClock::GetPaused(), TTextEdit::GetEditable(), TTextEdit::GetIgnoreChars().
- Sidewalk can now scale an existing animation using the new command-line parameter, –scale=#.
- TTexture::GetSimple() now has an "alpha" parameter to allow it to create textures with alpha.
- TTextEdit now has a UTF-8 mode which allows the user to properly input international characters. As an additional feature, TTextEdit will restrict the UTF-8 characters accepted to those that exist in the included font.
- TAnimatedSprite now sets a path that includes the sprite's folder so that a sprite can include other Lua files.
- FilmStrip now has a "properties viewer" that allows you to tweak the numerical properties of individual frames, the ability to export .anm files directly, and an onion-skin view that allows you to see several frames superimposed on top of one another. The look of FilmStrip has also been improved.
- FluidFX has a new image browser that restricts you to the current game/project, so that you don't accidentally choose an image that Playground (which is restricted to the assets folder) can't display.

B.1.15 Fixes in Playground 4.0.16

- PATCH - fixed a bug in OnKeyUp() where it didn't translate keys the same as OnKeyDown(), so several of the constants, in particular the arrow keys, didn't work correctly.
- Fixed an uninitialized variable in the particle renderer that caused random behavior if you failed to specify a blend mode for a particle system.
- Fix registration of TTextEdit when created from C++ code.
- Made it so that TSoundInstance::Kill() could be called within an OnComplete() callback without crashing.
- Fixed OnKeyDown() event to work with more "normal" keys, including numeric pad keys.
- Fixed a bug in TSprite where, if the TSprite::Draw() call actually removed the sprite from the parent's child-list, it would crash. This happened when TFxSprite::Draw() encountered a "done" condition.
- Prevent slow textures from being counted as "video RAM" allocations for purposes of debug tracking.
- Added missing include (pf/prefs.h) to pflib.h.

- Fixed Vista-Aero-desktop-related problems: When Ctrl-Alt-Del is pressed on Vista, and the Aero desktop is enabled, DirectX is forcefully disabled in your application instantly–and so texture creation would suddenly start failing unexpectedly. Now Playground will always hand you a valid texture reference.
- More Vista Ctrl-Alt-Del fixes, including a problem when hitting Ctrl-Alt-Del right before a Flash movie plays, and another problem where the mouse thread would cause a threading crash within DirectX during startup.
- Changed flat-file reading to guarantee alphabetical order in reading the .pfp files.
- Fixed skeleton to properly set the clock on the TFxSprites in the swarm demo.
- Fixed spurious DirectX warning printout when setting a software cursor.
- Fixed a bug in sidewalk's new "Delete duplicate frame" feature.
- Fixed a Flash memory leak.
- DoLuaString() now properly sets the LUA_PATH to the current path before running its string.

B.1.16 Migration to Playground 4.0.15

- A new set of ASSERTs has been added to enforce the proper order of calling Begin2d/Begin3d and End2d/End3d. All calls to the above functions are required to be between BeginRenderTarget and EndRenderTarget, or BeginDraw and EndDraw (which are typically called by Playground). There's a common–but incorrect–pattern that's easy to write that looks like:

```C++
if (renderer->BeginRenderTarget(...))
{
   TBegin2d begin2d;
   // draw something
   renderer->EndRenderTarget();
}
```

This is wrong, because begin2d will be destroyed *after* EndRenderTarget() is called. Instead you should write:

```C++
if (renderer->BeginRenderTarget(...))
{
   TBegin2d begin2d;
   // draw something

   begin2d.Done();
   renderer->EndRenderTarget();
}
```

Or alternately, add an extra scope:

```C++
if (renderer->BeginRenderTarget(...))
{
   {
      TBegin2d begin2d;
      // draw something
   }

   renderer->EndRenderTarget();
}
```

B.1.17 New Features in Playground 4.0.15

- Hot-key support! By adding the new code found in the Playground Skeleton to your project, you can automatically add hot keys (keyboard accelerators) to your project buttons.
- Cancel button support: A button in a dialog can be marked as the button to be pressed when a user hits escape.
- Better protection against calling Begin2d/Begin3d/End2d/End3d at incorrect times. Sometimes you can end up with bugs on other platforms if you don't follow the correct pattern of calling Begin2d/Begin3d **after** a BeginRenderTarget, and calling End2d/End3d **before** you call EndRenderTarget.
- Ignore .svn folders when scanning assets folder; this disables "bad filename" warnings for those folders.
- Added missing accessors to TClock and TTextEdit.
- Added the ability to restart a counter in FirstPeek metrics, so that the same counter can be reused.
- Added a TPlatform::SetConfig() flag, TPlatform::kOverlayWindowMouseEvents, that enables the routing of mouse events to overlay windows. See TWindowManager::AdoptOverlayWindow().

B.1.18 Fixes in Playground 4.0.15

- Fixed a text-wrap bug where if the last word on the line was partially surrounded by a tag, and that part of the word fit on the line but the remainder didn't, it would get repeated on the next line. For example, "<i>test<i>." at the end of a line where "test" fit but the period didn't, the word "test" would end up being repeated on the next line.
- pfpack.exe now can take an absolute path to the folder to compress, includes several added tests to verify files are being read and saved correctly, and will delete the destination pfp file if it detects any errors.
- Fixed TAnimatedTexture::GetRect and TAnimatedTexture::HitTest to properly set the current frame before doing their business.
- Added missing implementation for TTextSprite::HitTest.
- Default high-score name length has been extended to 20 chars.
- In skeleton, prevent user names that are all spaces, and crop spaces from start and end of name.
- Better windowed-mode display synchonization.
- In the Playground Skeleton, in TServerSubmit, if the user submits a score in anonymous mode, ignore whether or not the remember button is checked.
- Fixed a memory leak in TTexture::Save() for .png files.
- Fixed TPfHiscores::KeepMedal to correctly save the gameData parameter between sessions.
- Fixed ActiveX cursor scaling.
- Fixed occasional false "memory leak" warning that indicated that a TMessage wasn't deleted. Now the window script forces Lua to run its garbage collector immediately prior to quitting.
- Fixed ShowCursor(false) bug that was introduced in 4.0.14.
- Fixed TAssetMap to register files by the same internal handle as when you use the native class Get() function; this means that TTexture::Get(), e.g., will get the same instance as a file added with TAssetMap::AddAsset(), which is the intended behavior.

B.1.19 Migration to Playground 4.0.14

- The default behaviour of DoLuaString() has changed. Previously, TScript::DoLuaString(str luaCommand, int32_t length=-1) left a return value on the lua stack. The documentation did not describe this behaviour, so this return was rarely popped from the stack, and frequent use in a persistent context lead to unbounded growth. As of 4.0.14, TScript::DoLuaString(str luaCommand, int32_t length=-1, int nresults=0) has parameter nresults with the same intent as RunFunction(int32_t nargs=0,int32_t nresults=1). If code containing DoLuaString() migrated to 4.0.14 depended on this return value, it should be changed to specify 1 for the nresults parameter.
- The new metrics behavior has been changed since 4.0.13. Lines in the resulting file are now terminated by \r\n instead of just \n, and time is now recorded in seconds instead of milliseconds. Finally, the file is now named metrics.txt, and is saved to the individual user folder rather than the common data folder.

B.1.20 New Features in Playground 4.0.14

- Added several new functions to TLuaParticleSystem: fAlpha(), fDistance(), fDebug(), fOpenCycle(), and fClosedCycle(). Thanks to Ken at Zemnott for the prodding to support these.
- Added documentation to TLuaParticleSystem for all functions.
- Added new "crunch" mode to sidewalk that allows you to auto-compress your assets folder.
- **New FluidFX 2.0 included with SDK**! Not only has the Playground SDK particle editor been revamped, but it also comes with some new features, including built-in support for a variable emitter location (the skeleton application's dLocus), and a new dynamic variable that allows you to feed an orientation into the particle system. Also supports sets of particle systems that can be loaded with a single command.
- **New TFxSprite source code**, a TSprite-derived class that allows you to trivially attach a particle emitter to a sprite. TFxSprite directly supports the new features available in FluidFX 2.0.
- Cleaned up and improved new debug output information.
- New sub-pixel-accurate TTextGraphic::Draw() call that replaces the older call. The older call is still available, but now is deprecated.
- Added TTextGraphic::SetFont and TTextGraphic::SetFlags, to allow for deferred initialization of TTextGraphic.
- Added TTextSprite class which renders text as a sprite; text alignment is relative to position of sprite, and sprite rotation is supported.
- Added a const version of TSprite::GetDrawSpec(), to support const-propagation.
- The Playground Skeleton hiscore submission screen has been refactored so that it can be called from different locations (i.e. a high score screen and a medals screen). hiscoresubmit.lua has been replaced by serversubmit.lua, and THiscoreSubmit has been renamed to TServerSubmit and has been put in its own .h/.cpp files.
- The TPfHiscores API has been changed to allow easier storing and submission of medals. LogScore(), SubmitScore(), and SubmitMedals() have been deprecated, and the new API is KeepScore(), KeepMedal() and SubmitData(). The Playground Skeleton has been updated to use this new API as well.
- An example medals screen has been added to the Playground Skeleton to demonstrate the the storing and submitting of medals. This medals screen also introduces an example TRolloverWindow class to demonstrate rollover text.

- When using the pfservlet_stub.dll for testing of hiscore services, submitted medals will now be recorded in the serverdata.txt output file. Developers can verify that their medal submissions are accepted by the test servlet by checking that file.
- The Playground Skeleton app now has a "game over" screen. The game over screen lets the user choose whether or not to log a high score and also log medals. This should better demonstrate the high score and medals API.
- Added a new TPlatform::GetConfig() constant: TPlatform::kMonitorFrequency, to allow the game to query the current screen refresh rate. Can change when the computer switches between full screen and windowed mode.
- Changed the behavior of the "hardware cursor" on Windows to actually render as a software cursor, except in ActiveX mode.
- Sidewalk now, by default, sets completely transparent pixels in the output file to the color of an adjacent pixel. This new functionality can be disabled with the new command line option –noclear.
- Changed Sidewalk zlib to increase PNG compression based on recommendation here: http://www.cs.toronto.edu/~cosmin/pngtech/too_far.html
- New TButton::Press() member that encapsulates "Press the button" functionality.
- Added ASSERTs to DrawVertices() that fire if neither Begin2d() or Begin3d() is active.
- Added new kUTF8Mode configuration flag to enable TWindow::OnUTF8Char() virtual function support to allow reading of UTF-8 characters from the keyboard.

B.1.21 Fixes in Playground 4.0.14

- Fixed Mac xxxA rendering to not be broken after the first render in that mode. This fixes Mac TSprite::HitTest() as well.
- TLuaParticleSystem was causing unbounded growth of a lua stack due to a bug in DoLuaString() which was used by TLuaParticleSystem::Update().
- Fixed a function in the TLuaParticleSystem core (TParticleMachineState::GetOffset()) that is used internally by fPick() and fFade(). Previously it calculated the offset incorrectly if there was another value on the particle stack, so a statement like this would fail: pColor:Anim(fAlpha(pColor, fFade(pAge, 1, 1500, 0))); The generated commands would add pColor to the stack, and then fFade() would get incorrect fading values.
- Made TSlider auto-register in Playground, and removed the registration from the sample application.
- Adjusted layout of high score submittal screen so that mouse clicks on the Submit button are not intercepted by the legal policy text.
- On Windows, added a limiter to windowed mode updating that may improve the smoothness of animation.
- Changed TRandom::RandRange to officially support negative values, and fixed an edge case bug (at INT_MAX funny things happen to doubles).
- Fixed TLuaParticleSystem, TParticleRenderer, and T2dParticleRenderer interfaces to fully support a pluggable renderer.
- Fixed minor TLuaParticleSystem leak.
- Made the skeleton "more correct" by having it unregister script functions as the game classes exit.

- Prevent particle system from using indexed vertices when Playground detects a broken driver (specifically Intel 82845 or 82810 cards).
- Added missing kBlendMultiplicative constant to Lua particle system files.
- Fixed FilmStrip to be able to correctly load .lua files with "requires" statements.
- Fix FilmStrip to save an identity transform in the frame following a non-identity transform; the loading code assumes a null transform should be copied to subsequent frames.

B.1.22 New Features in Playground 4.0.13

- Added TRenderer::GetZBufferTest(), TRenderer::GetZBufferWrite(), and TRenderer::ClearZBuffer().
- Added new TGameState abstraction that wraps both a new FirstPeek API, and the MSN/Zylom web game API. Operates with optional plug-in modules; the FirstPeek data collection only happens when the first-peek.pfmod file is present in the assets folder, for instance.
- Added new debug command: Hit F3 during the game to see a list of all currently allocated assets.
- Added TAssetMap demonstration code to the skeleton.
- Added Alt-F1 HUD support to the Mac.
- Rounded out the new TLuaTable modifiers with the addition of:
  - TLuaTable::Assign(str,TLuaObjectWrapper*)
  - TLuaTable::Assign(lua_Number,TLuaObjectWrapper*)
  - TLuaTable::Erase(str)
  - TLuaTable::Erase(lua_Number)
- When loading an asset, it will now write an error to the debug log when the asset fails to load.
- To the TLuaParticleSystem fFade{} command, added the ability of the "time" parameter to go negative, holding the first fade value until it goes positive, allowing the fade values to start after a delay.
- TTextGraphic has a new optional "sprite" render mode. The default mode can be changed using TPlatform::SetConfig(TPlatform::kTextGraphicSpriteRender, 1).
- TTextGraphic::GetTextBounds() can take a scale to determine the bounds at a particular scale of the text.

B.1.23 Fixes in Playground 4.0.13

- Upgraded TAssetMap to support all new TAsset types.
- Improved the documentation of the Playground F-key debugging facilities, and made them all work on the Mac.
- Fixed TLuaParticleSystem to be able to index a data-source register in the particle virtual machine.
- On Windows, fixed a deadlock in the Sound thread when early returns in the main thread leaves the sound mutex locked.

- On OS X, reduced memory footprint of TTexture primarily by postponing the allocation of the buffer containing the texture's pixels returned by TTexture::Lock() until the first call to Lock(), instead of allocating it when the texture is created.
- On Windows, fixed a crash when we fail to allocate a cursor.
- Fix sibling-window-search-logic to properly find children of sibling windows when the first window found doesn't want the mouse input.
- Conditionally fix alpha-blend-to-render-target bug on Windows: Existing programs won't change their behavior, but if you call TRenderer::GetInstance()->SetOption("new_render_blend", "1"), it will now behave correctly.

B.1.24 New Features in Playground 4.0.12

- Updated xml2anm to support new animation features.
- Fixed FilmStrip to properly update and save animation script.
- Removed a lot of useless debugging statements to clean up debug output.
- TRenderer::SetCaptureTexture()/TRendererIsCapturePending(): Set a texture to receive the results of the next screen update, and look for the capture to have succeeded.
- Added SetConfig(TPlatform::kFlashMajorVersion), which sets a minimum Flash version that your game requires. If the client system doesn't support this version of Flash, then TFlashHost will fail when attempting to start a movie, and that case still needs to be handled by the game.
- On Windows, set the global Windows timer resolution to be 1ms, which will improve the frame rate on systems that have a much coarser default timer.
- Add TLuaTable::Assign() to allow the assignment of simple values to tables.
- "Do you want to quit?" dialog added to skeleton, since it's now part of the PlayFirst design guideline.

B.1.25 Fixes in Playground 4.0.12

- Fixed str::getFormatted to support strings of unlimited length.
- Fixed image load to be more exact on Windows: Previously images lost about 1/255 color value per-color on load because of the way DirectX was filling the textures. The colors should be better now.
- Fixed const-correctness of WriteDbg(), and improved its handling of long debug output messages.
- Fixed ActiveX-mode (when rendering is scaled) render target bug(s), and a related FillRect() bug.
- Fixed Mac to properly render after the projection matrix is set.
- Fixed a crash if you assign a null texture to a particle system.
- Fixed Mac sound pausing, so that music triggered when the game is in the background won't accidentally play. Also prevented Pause(false) from restarting sounds on the Mac if the sounds are not currently paused.
- Fixed bug where, if you have a Flash movie playing that is non-abortable, and you attempt to exit the game with the close box, it could lock the script (depending on game client code).

- Fixed Mac FillRect() of slow textures, as well as adding a work-around for a 10.4-only bug in NVIDIA GeForce 8600M GT drivers.
- Fixed Mac sound updating, so that mouse events can't interfere with sound updates.
- Fixed ActiveX issue where it would attempt to scan user: and common: for ActiveX games, even though those folders are not defined for ActiveX games. This prevented override .pfp files from working correctly.
- Fixed Mac close-down issue where close requests were not being properly dispatched to GetEvent(), as well as a few other shut-down related issues.
- Re-fixed Mac render target support, as verified by our new unit tests.

B.1.26 New Features in Playground 4.0.11

- Added a new rendering option that enables a more cross-platform style of subtractive render that we recommend all games activate: TRenderer::GetInstance()->SetOption("new_subtractive","1");
- Implemented hard limit on number of vertices that can be rendered using DrawVertices, and hard limit on vertices and indices for DrawIndexedVertices. Will ASSERT in debug build if you exceed either limit.
- Added new kVsyncWindowedMode option that instructs Playground to wait for a vertical blanking period before copying to the screen (on Windows, in windowed mode).
- Added TSlider::GetState() to expose the current state of the slider.
- In order to avoid many of the common bugs that are happening with the use of TPfHiscores, more information regarding the hiscore system has been added to key.h. All new PlayFirst games MUST use this version of key.h, and will be sent updated versions of key.h specific to their game. In addition, code has been added to the Playground Skeleton to show how to properly use this data in PlaygroundInit(). Again, all Playfirst games MUST use this initialization procedure. Finally, the need to use serverdef.txt to configure pfservlet_stub.dll has been removed. The pfservlet_stub.dll will now automatically configure itself based on the settings in key.h.
- Added a new tool to the distribution, xml2anm, that will allow you to create binary versions of xml animation files that will load more quickly than their xml versions.
- New version of FilmStrip–2.0! Many new and cool features!
- In skeleton application, created new file, version.rch, that contains the version information for the application. This prevents a common error where someone edits the .rc file with the IDE and overwrites the VERSION macros.

B.1.27 Fixes in Playground 4.0.11

- Fixed issue in the xsellkit addon where xsell.lua assumed that "label" was defined for Button (like the Playground Skeleton defines it). This assumption is now removed.
- Fixed TLuaTable::Create to be static.
- Plenty of documentation bugs are now fixed.
- Fix "mouse cursor doesn't display when game window isn't top window" bug.

- Fixed SetClippingRectangle() to work correctly on the Mac when using render targets; this allows you to draw to render targets using TWindow-derived classes.
- Fixed a render-target-related system-specific failure case where the primary and fallback copy-to-texture routines were both failing; added a secondary fallback copy routine that's slower but that always works.
- Fixed skeleton application Enter Name screen based on comments in forum.
- Fixed Mac sound looping bug.
- Fixed Mac flat-file bug.

B.1.28 New Features in Playground 4.0.10

- TAnimatedTexture can now support animated alpha.
- New TTextGraphic::SetAlphaBlend() enables a more sophisticated render-text-to-texture mode.
- Support for Decoda Lua debugger.

B.1.29 Fixes in Playground 4.0.10

- Correctly send full-screen-toggle event on Mac.
- Fix of particle register member indexing (pVelocity[2]).
- Fixed a problem with flickering on high-end video cards.
- Fixed a potential leak/behavior problem that could occur when calling BeginRenderTarget() right when the DirectX state fails.
- Fixed a bug in TSprite::HitTest() where BeginRenderTarget() was called, and the return value was ignored.
- Prevent EndRenderTarget() from crashing in release build if it's mistakenly called (it should only be called after a successful BeginRenderTarget()).
- Fix Mac command line to be constructed more like the Windows command line.
- Fix Mac bug that caused render-targets, TTexture::Lock(), and TTexture::CopyPixels() from working deterministically on some video cards.
- Fix recover-from-sleep-in-full-screen issue with semi-private TRenderer::BeginDraw() function when its parameter is true.
- Fixed a bug in TTextGraphic that would improperly crop text that was centered or bottom-justified if the text included a tag that reduced its size.
- Fix TTextGraphic rendering of outlined fonts to a texture.
- Prevent TWindowStyle from being copied.
- Prevent a crash in TAnimatedSprite if you call Pause() before calling Play().
- Fix TAnimatedTexture to properly use the transform values in the XML file if present.
- In TSprite::HitTest(), properly fail if internal BeginRenderTarget() fails (such as if DirectX has failed).

- Improved Playground support for Max OS X 10.5 (Leopard) and Microsoft Windows Vista (several fixes for each).
- Fixed an ActiveX issue where mouse would leave trails when MSN requested the game to pause.
- Fixed a problem in the skeleton where the name of the application didn't match the name in serverdef.txt.

B.1.30 Migration to Playground 4.0.9

- Your Mac project will need a new NIB file if it was created using previous versions of the Playground SDK. Open up the Playground Skeleton project, and open up English.lproj. If you're in a shell, it's just a folder, but if you're using Finder, you need to right click (control-click for you single-button users) and tell it to "Show Package Contents" to open it. Copy the FlashWindow.nib folder into your project's English.lproj. Then, in Xcode, select the project window, and drag (from a finder window) FlashWindow.nib into the "resources" group in the project. Tell it to "copy if necessary", "Recursively create group", and then click the "Add" button. Now you should be able to play Flash files using the new version of Playground.
- If you use the "align" property in TText or TTextEdit objects, *and* you use the little-known "negative offset means measure from the right/bottom edge" feature in those same objects, then you'll need to decide to use one or the other: "align" disables the negative-offset feature now, and negative positions will just position the window off the screen to left/above.
- To increase the security of an ActiveX build, we added another GUID that's used as the encryption key (GUID3) to the activex.bat configuration file.

B.1.31 New Features in Playground 4.0.9

- Hardware cursor support for Windows XP and Vista; Mac support is not in yet, nor will this ever be supported on older platforms, so supplying a software cursor is still required for full compatibility.
- An optional user-defined function PlaygroundInit() can be created to set the name of the application user and common folder names, as well as to change the name of the publisher for non-PlayFirst-published games. See the new skeleton code for an example, and documentation on the constant TPlatform::kPublisherName.
- New TWindow virtual function: TWindow::OnParentModalPopped(), which is called when a window's parent (or ancestor) modal is popped from the modal stack.
- Added new accessors to TModel, to allow access to model triangles.
- Added TFile::DeleteFile().
- Gave TPlatform::GetConfig() a new, optional default parameter.

B.1.32 Fixes in Playground 4.0.9

- 4.0.9.6 patch: Fixed more sleep- and resolution-change-related crashes in Vista and XP.
- Fixed several sleep and Ctrl-Alt-Del related crashes in Vista.

- Fixed word-wrapping for words longer than a single line of text (previously it would lose a character when it wrapped). Also will now display a cropped character if a single character is wider than an entire line of text.
- Fixed a Mac chained-sound problem where killing the sound didn't work correctly.
- Fixed two sidewalk bugs: First, images with opacity are now always cropped to opaque pixels, even if the opaque pixels go right out to the edge. In other words, if the entire source image is opaque, it won't crop the image at all. Second, there was a bug where if you give sidewalk a single image with an exact power-of-two width (or if your existing image exactly fit in a current row), it would bump the image to the next line. In the single-image case, this caused an invalid state and crash of sidewalk.
- Fixed some Flash playback issues on the Mac.
- Fixed a problem with TFile::AtEOF() for flat files, where AtEOF() would report EOF when you're read the last byte of the file, while traditional feof() (called by the normal TFile::AtEOF) returns EOF only after a failed read past end of file. Changed the behavior to match.
- Fixed an issue in axtool.bat that caused long game names to fail.
- Fixed a Mac render-to-texture bug in XXXA mode, where it wasn't properly clearing the backbuffer before rendering.
- Fixed a bug in TSlider that caused it to render more "blurry" than it should have on some platforms.
- Fixed declaration of CreateVertsFromRect() to be accessible from client applications.
- Fixed an uninitialized member in TTextGraphic that could cause it to report bad information about its contents before a call to RenderText() (which is called implicitly in SetText() and a few other places).
- Fixed a problem on the Mac where the cursor wouldn't update on static screens when the mouse was being dragged (the mouse button was down).

B.1.33 New Features in Playground 4.0.8

- Force simple and slow textures to NOT be valid drawing sources; now they correctly fail (and ASSERT) when attempting to DrawSprite() or SetTexture() with them. Note TImage uses DrawSprite(), so slow textures are also forbidden in Bitmap{} calls.
- Added an "antialiasing" hint to Playground.
- Added TTextEdit::GetCursor and TTextEdit::SetCursor, to enable TTextEdit overriding.
- In the skeleton application, add a Text{} field to the credits screen that shows the game version.
- The function TPlatform::Rand() is now fully documented and supported. Rand() is seeded using the system time, and returns a stream of pseudo-random 32-bit numbers using TRandom internally.
- TRenderer::SetOption('fps', '0') will disable FPS output.
- Added TWindow::GetWindowPos()
- Added TImage::GetAlpha()
- Playground Skeleton now has a shrink-to-fit hiscore name feature: When presented with really long PlayFirst user names, the Skeleton will shrink the name until it fits.
- TTextGraphic::SetLineHeight() can now change the font size after construction.

- TText::GetTextGraphic() can now access the TTextGraphic hidden in a TText.
- Added str::sizeof_utf8_char() so that it's possible to iterate through UTF8 characters in a translated string.
- Fixed Lua-binding of functions returning void with 4 or more parameters.

B.1.34 Fixes in Playground 4.0.8

- Fixed sound chaining on Mac.
- Fixed CopyPixels() bug on nVidia cards on Mac.
- Fixed package size on Mac (was accidentally including packages twice).
- Added a hack to work-around a bug in Vista that caused it to think a Playground game had stopped responding, despite the fact that it was actively pumping messages.
- Fixed 4+ parameter Lua function binding for functions with a void return type.
- Fixed crash bug in FluidFX.
- Fix Windows Playground to not allow drawing from "slow" textures.
- Fix DrawSprite() clipping on Mac.
- Fix SetFullscreen() to properly return false when attempting to set windowed mode on a system that has been flagged as having insufficient video memory for 32-bit mode.
- Reduce the aggressiveness of the low-video-memory flag, so that it doesn't get triggered when the initial display is being initialized, as sometimes happens on a resume from sleep.
- Fix problem with render targets in 16-bit display depth (in windowed mode) on some hardware.
- Expand some fixed path lengths internally, and convert others to use str, to attempt to address a very-long-path-length issue.
- Fix toggle-style TButtons to work correctly when the toggle has a command that brings up a dialog. Also fix bug where pressing mouse down on one button and then moving it over a toggle would cause the toggle to flip states but not call the command.
- Fixed crash bug in FluidFX
- Prevented DisplaySplash() from accidentally eating up close messages from other windows.
- Fixed a small TSound memory leak.
- Fixed TStringTable to read empty rows as strings that translate to "", rather than not including them in the table at all.
- Clean up an fix a minor bug in chooseplayer.lua in the skeleton.
- Fix Mac bug where right- and middle-button-up messages were being sent as left-button-up messages.
- Fixed a z-render-depth issue on Mac.

B.1.35 New Features in Playground 4.0.7

- Added the ability to mask out files from the local file system (TFile::AddFileMask).
- Added a new key in settings.xml that selects one of two file masks: when the <firstpeek> tag is set to 0, the file system looks for final.txt next to the executable, and if found, passes the contents to AddFileMask(). Alternately, if <firstpeek> is set to 1, it reads firstpeek.txt.
- New Flash translation technique that also works correctly on Mac.
- TTextEdit can now delay its registration with its parent modal window, and you can call TTextEdit::Unregister() to cause it to release its connection with the modal window. This allows a TTextEdit to exist in a detached window hierarchy that is periodically attached when needed.

B.1.36 Fixes in Playground 4.0.7

- Fixed TLuaParticleSystem to properly load and display animated textures.
- Fixed Mac TColor32 implementation to work correctly with locked textures.
- Fixed Mac text calibration bug.
- Fixed TSlider to create correctly from C++.
- Fixed TTextGraphic to not corrupt the screen when rendering to a texture during a Draw() phase.
- Fixed Mac TTexture::Lock() to return the correct pitch on non-power-of-two texture surfaces.
- Fixed Mac 3d cull order.
- Fixed Mac render-target blending to match Windows behavior.
- Removed some hard-coded values to allow Playground to scale larger than 800x600 on Windows.
- Added a missing TRenderer::SetShadeMode() implementation.
- Properly test to see if a file exists in user: or common: when opening it for read.
- Fix two crash bugs when shutting down the app while playing an SWF.
- Fix a memory leak that occurs if someone forgets to call TTexture::Unlock() after TTexture::Lock().
- Correctly set TTextGraphic to be noncopyable.
- Improved comments and removed cruft from skeleton style.lua.
- Fix ActiveX version to never write a log file.
- Prevent right-justified text from being cut off by one pixel.
- Prevent right-clicking on SWF file from bringing up Flash menu when "allow input" is enabled.
- Fixed a bug where the cheat-enabling application didn't work with Together.
- Added a few missing items to the 4.0.6 changelog below.

B.1.37 New Features in Playground 4.0.6

- Added TLuaParticleSystem::AdoptFunctionInstance().
- TRect::Contains(), which deprecates TRect::IsInside().
- Added a line to the skeleton credits file that gives proper credit to the Playground SDK.

B.1.38 Changes in Playground 4.0.6

- Changed GetTypeName() to be available in debug and release builds.
- Documentation typo fixes and updates, including lots of additional docs on TDrawSpec usage.
- Removed include of Carbon headers in debug.h–they shouldn't have been there to begin with.
- TPrefs: Handle bad encryption key or corrupt file more gracefully.
- On Mac, fill screen with black when starting up.

B.1.39 Fixes in Playground 4.0.6

- Fix pfservlet_stub to behave like actual hiscore server when SubmitMedals() is used immediately after SubmitScore() without first waiting for a server response. It is recommended that you use the new pfservlet_stub.dll when testing your hiscore functionality.
- Fix BeginRenderTarget() on Windows to reset internal state if it's called when DirectX is disabled (e.g., when the window is minimized).
- Fix bug in hiscore system where ranking values returned in TPfHiscores::GetScore() were not always correct when scores were logged using the replaceExisting flag in TPfHiscores::LogScore().
- Fix Mac sound bug.
- Fix big-endian reading of new TPrefs format.
- Fix Crash/ASSERT in simplexml.
- Fix const-correctness of TDrawSpec::GetRelative()
- Fix test for kCenter in TAnimatedSprite::GetRect to actually test the right mFlags.
- Remove some long-unused debug info.
- Fix Lua hex conversion to use unsigned int internally (GCC was truncating a negative signed value to 0).
- Fixed IsForeground() return result when running under Together.

B.1.40 New Features in Playground 4.0.5

- Added "desktop:" file prefix, for saving files to the desktop
- Added TTexture::Save for saving textures to a .jpg file.

B.1.41 Migration to Playground 4.0.5

- Mac developers must now link against WebKit.framework.

B.1.42 Fixes in Playground 4.0.5

- Fixed an intermittent bug in the optimized TXmlNode.
- Fixed Mac Flash playback to not rely on Apple's broken QuickTime Flash player.
- Fixed TFile::Exists() for user: and common: files.

B.1.43 Changes in Playground 4.0.4

Be sure you do a complete rebuild and replace pflibDebug.dll when you get 4.0.4!

- str, TXmlNode (simplexml), and TPrefs have been internally optimized.
  - TPrefs in particular now runs much faster with larger data sets–there were a few unintentional $O(n^2)$ algorithms in the load and save data paths. Also, TPrefs binary data is now saved directly in the file (encrypted), and doesn't need to be uuencoded–so it should also be a lot faster.
  - Parts of str were inlined.
- Several str APIs now take or return size_t parameters to closer match std::string.

B.1.44 New Features in Playground 4.0.4

- Include Visual Studio 2005 plugin that displays the contents of str variables in the debugger. See bin/p-faddinReadme.txt.
- TEncrypt::GetLastSize() gets the actual size of the last encrypted or decrypted data.
- TFile::AddMemoryFile() allows you to add a virtual file to the file system.
- New str APIs were added to bring str closer to std::string.

B.1.45 Fixes in Playground 4.0.4

- TSound::Get() now takes a str as its first parameter instead of a char*, which makes the interface more consistent.
- Fixed clipping rectangle issues: Prevent upper-left corner of clipping rectangle from causing an offset in rendering, and allow clipping rectangle to be updated dynamically.
- Fix ScriptRegisterDirect() to support functions with more than three parameters.
- Fix bug in TTextGraphic::GetTextBounds() in handling of zero length strings.

- Fix Flash rendering bug when TFlashHost::Start called in BeginDraw/EndDraw.
- Fix swf2mvec parameter processing bugs.
- Const-correctness fix in str::find.
- Cursor delta mode won't let the cursor escape on PC.
- An OnKeyDown() problem with key flags was fixed.
- TFile now reads subfolders in user: and common: correctly.

B.1.46 New Features in Playground 4.0.3

- TFlashHost::Start() now has an optional bAllowInput parameter to allow interactive flash movies.
- The TScriptCode::Get() function has a new, optional parameter that retrieves any error message related to the loading of the Lua file.
- Support for delete key in TTextEdit.
- Sidewalk now puts a 1-pixel border between images to prevent bleed in rendering.
- Sidewalk now has a –reg= option to allow a fixed registration point to be specified.
- Changed Begin2d() to reset 3d matrices on entry and exit, but respect them for doing 2d transforms. Perspective matrix is coerced into being a 2d perspective transform, and cannot be changed.
- Optimized DrawVertices() to not set the matrices as often.

B.1.47 Fixes in Playground 4.0.3

- Fixes for anonymous hiscore demonstration in the Playground Skeleton application.
- Previously DrawVertices() stomped on the matrices in some cases accidentally; now changed the design so that Begin2d() stomps on matrices intentionally, and DrawVertices() never does.
- Fixed back button bug in the xsellkit addon.
- TScript::RunScript() now correctly prints the error message generated when parsing and compiling a Lua file.
- Scroll-wheel messages now work as advertised.
- SetTextureMapMode() link error fixed.
- Fixed some internal sound resource lifetime issues.
- Fixed memory leak in sound notifications.
- On Mac, fixed TPlatform::Exit to not immediately exit application.
- On Mac, fixed render-target clipping.
- On Mac, include default soft-cursor.
- On PC, forced default cursor to only be initialized once.

B.1.48 New Features in Playground 4.0.2

- When creating new files in user: or common:, any folders specified will be auto-created if they don't already exist.

B.1.49 Fixes in Playground 4.0.2

- Fix TLuaTable(TLuaObjectWrapper) constructor.
- Fix Windows fullscreen Flash bug.
- Fix documentation re custom TMessage creation.
- Fix QuickTime/Flash enable detection on the Mac, so that when Flash is enabled in QuickTime on Intel-based Macs it will display them again. Still pending is a switch to WebKit that will allow Flash to work on ALL Macs.
- Fix to TLuaParticleSystem that allows complex expressions in Vec∗() and Color() definitions.
- Fix TMessage delivery so that OnMessage() gets an unwrapped message pointer.
- Fix TLuaMessageWrapper to properly be set PFLIB_API, so that client code can call TLuaMessageWrapper::ClassId().
- Fix TLuaMessageWrapper::IsKindOf() to properly take a PFClassId type.
- Fix cursor-delta mode to work in ActiveX mode.
- Fix TTextGraphic::GetTextBounds() to correctly read bounds of justified text.

B.1.50 Migration Notes for Playground 4.0.1

- The signature for the sound notification callback OnComplete() has changed.
- Setting a viewport with zero size will now FAIL–so don't do that any more.

B.1.51 New Features in Playground 4.0.1

- TPlatform::SetCursorMode() can set the mouse to be in a "delta" gathering mode. OnMouseMove() then only provides mouse-deltas, the mouse is hidden, and the mouse is prevented from leaving the application window. Note that you need to make your window a mouse listener if you want to get the mouse events.
- Some minor optimizations of TAnimatedSprite(). More to come.
- Documentation updates, including the animatedsprite.lua file documentation that includes documentation on Lua calls available in TAnimatedSprite scripts.

B.1.52 Fixes in Playground 4.0.1.4

- Updated axtool to 4.0 source

B.1.53 Fixes in Playground 4.0.1

- Compatibility bugs in the render-target support on some systems have been fixed.
- A bug in sound-complete notification has been fixed.
- Fix a bug in the particle system that caused particle systems to interfere with each other.
- Fix a compatibility bug in FillRect() that caused problems on some DirectX cards.
- Fix crash-when-closing-during-splash-movie bug.
- Fix a Mac compatibility problem with TColor32 accessors.
- Make kInstallKey always return the same value no matter how the game is launched.
- A render-target problem with textures larger than 600x600 was fixed.
- A bug in the TTexture::Create() call that caused some systems to report the wrong width and height has been fixed.
- Mac render-target support bugs fixed.
- Mac CopyPixels() bug when drawing texture-to-texture fixed.
- Fixed skeleton Release DLL linkage.

B.1.54 New Features in Playground 4.0.0 Beta 1

- All new features up to 3.5.0.6 are included in this 4.0 beta.
- A TSoundInstanceRef is returned for each sound instance that's played.

B.1.55 Fixes in Playground 4.0.0 Beta 1

- Screen now refreshes right away when swapping screen resolutions.
- Writable is now set to a default ("Playground Application") if the resource info isn't found.
- Render target support works correctly on the Macintosh.

B.1.56 Migration Notes for Playground 4.0.0 Final

- Several TRenderer enums have been standardized as singular:
  - EBlendMode
  - ECullMode
  - EFilteringMode
  - ERenderTargetMode
  - EShadeMode
  - ETextureMapMode

B.1.57 New features in Playground 4.0.0 Final

- Major documentation rework.

B.1.58 Migration Notes for Playground 4.0.0 Beta

- TSound::Play() now returns a TSoundInstanceRef instead of an int. You now need to use the TSoundInstanceRef to modify an individual sound once it's playing.
- TRenderer::BeginRenderTarget() now takes a mode parameter that you must specify to indicate how you will use it. The safest (though slow) mode to use is kMergeRenderRGB1, which will all you to render onto an existing texture and will make the alpha of the entire image opaque. See TRenderer::BeginRenderTarget() for more information.

B.1.59 Migration Notes for Playground 4.0.0 Alpha 3

- TWindow::OnKeyDown now takes a uint32_t, which means you need to update the signature of any classes that override it if you want it to be called.
- TLuaParticleSystem::Draw2d() was changed to TLuaParticleSystem::Draw().
- Folders have changed location. In your project file you should make the following substitution:
  - utilities\bin -> bin

B.1.60 Fixes in Playground 4.0.0 Alpha 3

- TMatrix::Rotate() now properly rotates the 2x2 transformation portion of the matrix without changing the position, and it uses a pre-transform so that scaling is also properly rotate.

B.1.61 Migration to Playground 4.0.0 Alpha from Playground 3.5.X

The list below is long because of the many aspects of Playground that have been improved or cleaned up between 3.5 and 4.0. A number of the changes listed below will not show up as compile errors, and will likely manifest as parts of the program not working. The good news is that most of the changes tend to involve the removal of code or tweaking of calling conventions or include paths.

APIs that draw to the screen are now respecting the screen's viewport settings. This makes the library more consistent, in that some functions already respected the viewport (TTexture::DrawSprite, for instance). However, if your game is on the 3.5 branch and uses any of the modified APIs, you should check to make sure that you're really using TWindow client coordinates, since the TWindow::Draw() function is called with the viewport set to be the size of the TWindow.

If you're working on a development contract with PlayFirst, be sure that you talk with your PlayFirst producers about any schedule impact that converting to 4.0 might entail–and don't even try if you've already submitted a beta candidate, because PlayFirst has already done too much testing on your current build to wind back the clock and start testing on 4.0.

As always, direct questions or issues to the forums! PlayFirst employees monitor the forums, and Playground developers frequently are able and willing to provide help with issues they've already encountered.

- *All* Playground library files are now in a "pf" subfolder. Loading include Playground include files should look like:

```C++
#include <pf/pfconfig.h>
```

- Add an SDK.txt to your project along with related support, including copying the Pre-Build step from the skeleton application.
- TTransformedLitVert type has changed its behavior: Now when drawing using TTransformedLitVert vertices, the coordinates specified will be relative to the current viewport.
- TTextGraphic objects will also be drawn relative to the current viewport. Note that the standard credits.cpp will need to be changed to match the new skeleton credits.cpp in order for credits to continue to function correctly. The scripts/credits.lua file may also need to be updated.
- The blend mode kBlendAdditive has been eliminated in favor of kBlendAdditiveAlpha. Subtle differences between DirectX and OpenGL encouraged us to simplify and support only the alpha variant of additive blending.
- TWindow initialization has changed:
  - The semantics of TWindow::Init() has changed. It is only called during Lua window construction, and is now called after window position and size have been initialized. Its signature has changed to return void and take a TWindowStyle& parameter. Search your code for Init() overrides and update the signatures or your Init() function won't be called because of the new signature! If you need behavior similar to the previous TWindow::Init(), change your call to TWindow::OnNewParent() (see below).
  - TWindow::PostChildrenInit() also takes a TWindowStyle& parameter, so you need to change its signature in client code as well.
  - You **must** call the base class of TWindow::Init() and TWindow::PostChildrenInit() in the new initialization model.
  - Windows no longer automatically resize to fit their children. In Lua, you can add the tag "fit=true" to a window definition to let it know you want it to grow to encompass its children. Or you can add that tag to your default style if you want all windows to grow in a way similar to the 3.X behavior. In C++ code, you can call TWindow::FitToChildren to request that a window resize itself to encompass its children. If suddenly no mouse messages are going to window children, hit F2 and look at your window hierarchy: Probably one of your custom windows isn't getting a size. That one needs fit=true.

  - TWindow::PostInit() has been removed. You can safely replace it with TWindow::Init() if you're building windows with Lua; otherwise you'll need to use PostChildrenInit or create your own post-init call.
  - TWindow::OnNewParent() is a new API that's called at the same time as the previous TWindow::Init(), though it no longer has the style information available to it that it had in 3.X, so most initialization should be moved to the new TWindow::Init virtual function.

- TWindow::OnDirtyRect(), TWindow::InvalidateRect(), and all dirty-rectangle management functions have been removed. To cause a screen refresh, you can call TWindowManager::InvalidateScreen().

- TPlatform::GetConfig() now returns the game's version with GetConfig(kGameVersion). This was previously handled by TPlatform::GetVersion().

- A new singleton class, TRenderer, is now the container for all screen rendering related functions, so any reference to any of the following functions or their associated types as part of TPlatform will need to be changed to refer to TRenderer::GetInstance():

C++
```
bool TRenderer::Begin2d();
bool TRenderer::Begin3d();
bool TRenderer::BeginDraw(bool needRefresh);
bool TRenderer::BeginRenderTarget( TTextureRef texture,
                                   bool fullCoverage=false );
void TRenderer::DrawIndexedVertices( EDrawType type,
                                     const TVertexSet & vertices,
                                     uint16_t * indices,
                                     uint32_t indexCount );
void TRenderer::DrawVertices( EDrawType type, const TVertexSet & vertices );
void TRenderer::End2d();
void TRenderer::End3d();
void TRenderer::EndDraw(bool flip=true);
void TRenderer::EndRenderTarget();
void TRenderer::FillRect( uint32_t x1,
                          uint32_t y1,
                          uint32_t x2,
                          uint32_t y2,
                          const TColor & color,
                          TTextureRef dst=TTextureRef());
void TRenderer::GetProjectionMatrix(TMat4* m);
bool TRenderer::GetTextureSquareFlag();
void TRenderer::GetViewMatrix(TMat4* m);
void TRenderer::GetViewport( TScreenRect * viewport );
void TRenderer::GetWorldMatrix(TMat4* m);
bool TRenderer::In2d() const ;
bool TRenderer::InDraw() const ;
void TRenderer::PopViewport();
void TRenderer::PushViewport( const TScreenRect & viewport );
bool TRenderer::RenderTargetIsScreen();
void TRenderer::SetAmbientColor(const TColor & color);
void TRenderer::SetBlendMode( EBlendModes blendMode );
void TRenderer::SetCullMode(ECullModes cullMode);
void TRenderer::SetFilteringMode( EFilteringModes filteringMode );
void TRenderer::SetLight( uint32_t index, TLight * light );
void TRenderer::SetMaterial(TMaterial* pMat);
void TRenderer::SetOrthogonalProjection(TReal nearPlane, TReal farPlane);
void TRenderer::SetPerspectiveProjection( TReal nearPlane,
                                          TReal farPlane,
                                          TReal fov=PI/4.0f,
                                          TReal aspect=0);
void TRenderer::SetProjectionMatrix(TMat4* pMatrix);
void TRenderer::SetShadeMode( EShadeModes shadeMode);
void TRenderer::SetTexture(TTextureRef pTexture=TTextureRef());
void TRenderer::SetTextureMapMode( ETextureMapMode umap, ETextureMapMode vmap );
void TRenderer::SetView( const TVec3 & eye, const TVec3 & at, const TVec3 & up );
void TRenderer::SetViewMatrix(TMat4* pMatrix);
void TRenderer::SetWorldMatrix(TMat4* pMatrix);
```

```
void TRenderer::SetZBufferTest( bool testZbuffer );
void TRenderer::SetZBufferWrite( bool writeToZbuffer );
```

- TWindowManager::SetCapture() and TWindowManager::ReleaseCapture() have been renamed to TWindowManager::AddMouseListener() and TWindowManager::RemoveMouseListener(), respectively. These names are more indicative of their actual behavior, since you're not *capturing* the mouse–you're just becoming one of many listeners.
- PopModal has a new behavior: You need to give it the ID or name of the window you're popping. If you were previously using it to return a value, instead call ModalReturn(value) and then PopModal(id) or PopModal(name). PopModal is also more aggressive, in that it *will* pop the window you name, along with any modal windows that have been pushed on top of it.
- Construction of TPfHiscores has changed (again); game name and version are now extracted from TPlatform::GetConfig(). If you need to change the game name, you should edit the resource file (.rc) *with a text editor* to change the ProductName. If you edit the .rc file with the Developer Studio resource editor, it will break the version string macro, which must stay in place.
- Button{} has changed: The default Lua Button{} function only creates a very basic TButton. The skeleton includes a replacement Button{} that behaves similarly to how the 3.X Button{} worked; include that definition in your style.lua in order to keep the same behavior. If your buttons are showing up with missing text, this is what you need to add.
- The TSlider class has become a part of the library. If you are using TSlider in your game from 3.3.X, then if it's heavily modified, you should rename the files and class to not conflict with the library version. If it's not modified and you want the new functionality, then you can use the new TSlider in your game. The new TSlider needs different graphics: The two end caps and a scalable mid-section. To upgrade, delete slider.cpp and slider.h from your project, modify the slider assets so that you have top, middle, and bottom images, and then change the slider Lua instantiations to match the ones given in the anitest sample application. Be sure to get the style from playgroundskeleton/assets/scripts/styles.lua and select that style before you create new sliders (see playgroundskeleton/assets/scripts/options.lua for an example).
- TPlatform::HideCursor() was renamed to TPlatform::ShowCursor(), and its parameter meaning was flipped.
- TTexture::DrawFast() has been renamed to TTexture::CopyPixels(), and is now explicitly forbidden during a Draw(). This prevents problems on some video cards related to interleaving 2d copy methods with 3d render calls (like TTexture::DrawSprite()).
- TLuaTable::PushValue() now pushes **nothing** and returns false instead of pushing nil if the key isn't found, so be sure to update any uses of TLuaTable::PushValue in game code.
- TLight has been broken out into its own new header. Any files that use it must include the new header:

```
#include <pf/light.h>                                                    C++
```

- TRenderer::FillRect now takes a TURect parameter rather than four separate parameters. Additionally, when using TRenderer::FillRect to draw to the screen, the current viewport offset is taken into account–so coordinates will be relative to the upper-left of the current window.
- When a TWindow is marked as TWindow::kOpaque and it fills the entire screen, no modal windows behind its parent modal window will render. A TImage with 100% opacity will assume it's opaque. If you have any full screen TImages that have translucent or transparent on a modal window that is supposed to layer on top of another modal window, you may need to explicitly mark the TImage as non-opaque. You can do this from Lua by specifying alpha=true explicitly.
- Many integer types throughout Playground have been changed to unsigned in cases where negative values would have been illegal. Client code may need to be updated to reflect the new integer types (to eliminate new warnings).

B.1.62 New Functionality in Playground 4.0.0 Alpha 3

- TMat3 now has a 2x2 matrix multiply operator (%, or Multiply2x2) that ignores the translation component.
- TAnimatedSprite now has a Stop() function that will stop the animation but not kill its script. This way you can add functions or set persistent data within the script and it won't be killed every time you play it.
- An embedded cursor image will be used as the default cursor on Windows to avoid bugs with the hardware cursor.
- Debug logs will not be limited by size in debug builds.

B.1.63 New Functionality in Playground 4.0.0 Alpha 2

- TFile has two new append modes: kAppendBinary and kAppendText.
- Round out TAnimatedSprite and TAnimatedTexture.
- Fixed build and packaging script to properly include docs and two more utilities.

B.1.64 New Functionality in Playground 4.0.0 Alpha 1

- The TRandom class allows you to get high quality and very fast random number streams with configurable state and seed. You can instantiate a TRandom class for each independent random number stream you need. The generator has a period of 2^19937-1, and distributes its pseudo-random numbers evenly across 623 dimensions. And it's faster than rand(), if you're worried about speed.
- Completely hidden TModalWindows are no longer drawn. "Completely hidden" is defined as being behind a TModalWindow that has a child that covers the screen and has the kOpaque flag set. The flag is set automatically by the TImage class.
- The TSlider class gives you scalable slider (like a volume slider or scroll bar) functionality–it's an improved version of the slider in the sample game. Specifically, you give it a height or width, and it scales to the given size. A height means it's drawn vertically, and a width indicates it's to be drawn horizontally. It's drawn using top, stretchable middle, and bottom images.
- TTexture::CreateSimple() can create a "slow" (system RAM) texture with an optional third parameter now. This allows you to create large or oddly shaped textures which you can use as a source for TTexture::CopyPixels().
- A new pftypes.h that includes C99-style types for use in library functions as well as Playground-specific types.
- TWindow::FitToChildren will grow a window's boundaries to encompass its children.
- TWindow::PostDraw can be used to draw on top of a window's children.
- A new TRenderer class that encapsulates all rendering-related functions (except those on TTexture).

B.1.65 Major Changes to Playground 4.0.0

- Much cleaner TWindow initialization.
- The magic behind Button{} is exposed in client Lua code now, making it easier to create Buttons with custom behaviors.
- Screen saver functionality has been removed from Playground.
- Support for paletted textures has been removed from Playground due to compatibility issues.
- The call TTexture::DrawFast() has been changed to TTexture::CopyPixels(), which more clearly describes its semantics.
- Window-style specific functions have been removed from TScript: All of the Get∗() (though not GetGlobal∗) functions that extracted information from a window table or style have all been moved to TWindowStyle.
- Dirty rectangle support has been removed.

B.1.66 Minor Changes to Playground 4.0.0

- TLuaTable::PushValue() now pushes **nothing** and returns false instead of pushing nil if the key isn't found.

B.1.67 New Features in 3.5.0.6

- TSound::GetSoundLength()

B.1.68 Fixes in 3.5.0.6

- Fix a TTexture::DrawSprite clipping problem.
- Fix TAnimTask to correctly only call its animation once per frame.

B.1.69 New Features in 3.5.0.5

- Set the cursor position with TPlatform::SetCursorPos. This enables relative mouse addressing.

B.1.70 Fixes in 3.5.0.5

- Change TImage to always use TTexture::Draw, which fixes flickering on some really annoying video cards with terrible drivers.

B.1.71 New Features in 3.5.0.4

- New TFile open modes:
  - kAppendText
  - kAppendBinary

B.1.72 Fixes in 3.5.0.4

- Patch to fix problem with flat file system in Windows 98.

B.1.73 Fixes in 3.5.0.3

- Updated anitest sample to have the newer animated sprite and animated texture code.
- Fixed a bug in text scaling so that it now scales more smoothly.

B.1.74 New features in 3.5.0.2

- str::downcase()
- str::find_first_of()
- str::find_first_not_of()

B.1.75 Fixes in 3.5.0.2

- Fixed release build for anitest sample.
- Handle encryption keys of any length.
- Fix a bug where PopModal() during draw could cause a crash.
- Add a missing TVec2 operator implementations.
- Removed empty TVec* destructors.
- Fixed TSoundManager::KillAllSounds() crash bug.
- Fixed anitest sample code to properly position animated image that changes size.
- Fixed TFile bug where attempting to open a file that didn't exist would add an entry to the cache with that file name, improperly indicating when asked again that it did exist.
- Fixed bug in TTexture::GetInternalSize() where it would sometimes return the wrong size.

B.1.76 Migration to Playground 3.5.0 from Playground 3.3.X

- Be SURE to delete your old Playground distribution before you install the new one! Files have been deleted that, if you don't remove them, can hinder your efforts at becoming compliant.
- Remove the STLport includes from debug and release builds of your application.
- Release builds of your game now need to link to iphlpapi.lib.
- ENCRYPTION_KEY definition needs to be moved to file key.h in your src folder. This is to allow automated creation of the new cheat-enabler application. See the sample application for an example of the new key.h file. Typically this file is included in your settings.cpp file.
- Construction of TPrefs and TPfHiscores has changed, with the encryption key now in a TPlatform configuration setting. If you are using settings.cpp from the sample, there should now be a line that reads:

```C++
TPlatform::GetInstance()->SetConfig(TPlatform::kEncryptionKey,ENCRYPTION_KEY);
```

  And this line should appear before TPrefs or TPfHiscores is created. The ENCRYPTION_KEY parameter has been removed from each of those constructors.
- In Release builds, your game needs to get its cheat mode setting from TPlatform::IsEnabled(TPlatform::kCheatMode). You will be supplied with an application (cheat.exe) that has your game's encryption key burned into it for enabling cheat mode. This call will only succeed after the application encryption key is set, as per above instructions.
- Any code that uses TranslateResource to test for a file's existence must be changed to use the new TFile::Exists() API.
- Any code that uses GetDataDirectory() can be trivially changed by removing all path mangling and instead use the "user:file.ext" or "common:file.exe" URI format in any file name given to TFile::Open() or any library resource request.
- Paths in your game need to be modified to assume that they are accessing a file relative to the assets folder. Including "assets" in the path is no longer allowed.
- There is a new required parameter in TPfHiscore::LogScores() - replaceExisting. This parameter tells the hiscore system whether or not to replace the existing hiscore (for use in story/career mode games) or create a new one (for use in arcade mode games). If you are in doubt about what to use for this parameter, please contact your PlayFirst producer.
- If you're not already using Visual Studio 2005, then there are changes you need to make:
  - In Release Build, on the Linker properties page, select Optimization, "Don't Remove Redundant COMDATs". That optimization, as implemented in VS2005, is not compatible with Playground.
  - In all builds turn on C++ Exception Handling in C++ code generation.
  - The statically linked Playground release build needs to be linked against DirectX 7 libraries. If you don't have the DirectX 7 libraries (e.g., from MSDN subscription CDs), then you'll need to switch your project over to use the Release DLL build: 1 Remove PFLIB_STATIC_LINK from your Preprocessor Definitions. 1 Change the Linker Input from pflibStatic.lib to pflib.lib. 1 Copy the pflib.dll from pflib/lib into the folder of your executable.
- If you're using the old Playground Skeleton sample as your base, the credits.cpp file used fopen() style file access. Please update your credits.cpp to the new version.
- Be sure to read the new Coding Standards on the site at https://developer.playfirst.com/standards and update your code to the new standards. Important things to note:
  - Your code must build with warning level 4 with no warnings in release build.
  - You must remove FILE- and fopen-based code from release builds.

B.1.77 New Functionality in Playground 3.5.0

- Transition to Visual Studio 2005: support for Visual Studio 2003 has been dropped. You'll need to upgrade to 2005 to get official support on Playground.
- New TFile file abstraction. *All direct file access in 3.5 must be done using this file abstraction.* See the TFile documentation for details. Internally all file accesses are handled using this new abstraction. A set of functions is available as a drop-in replacement for fopen/fclose style usage, as well as a C++ style interface.
- Transparent support for collapsing the assets tree into a single flat file; the flat files are created as part of the build process. XML and Lua source files are also compressed/obfuscated in packaged builds.
- Added TPlatform::SetTextureMapMode to allow the client to switch between WRAP, CLAMP, and MIRROR texture mapping modes.
- New parameter (replaceExisting) added to TPfHiscores::LogScore()
- Lots of new documentation

B.1.78 Major Changes to Playground 3.5.0

- Removed STLport support and libraries. Moving forward we are only going to support the containers supplied in both the VS2005 STL and the GCC STL.
- Removed all APIs dealing with paths: TranslateResource(), GetDataDirectory(), TDirectorySearch, and everything in fileio.h.

B.1.79 Fixes and Minor Changes in Playground 3.5.0

- Added a copy constructor to TVertexSet to work around a bug in the C++ spec.
- Changed undocumented str::insert to the more understandable str::overlay, since the function allows you to overlay or extend a string with an existing const char *. Also added documentation.

B.1.80 What happened to 3.4?

Playground version 3.4 was a feature-freeze of the main development trunk for use in Diner Dash Flo on the Go. The Macintosh port was also finalized on 3.4 internally, and several Mac titles shipped on 3.4. However, it never became fully productized–this documentation, for instance, languished–so 3.3 continued as the public release, sometimes getting features and patches that didn't make it to 3.4.

Now that we have GM products on 3.4, we don't want to change it–it's completely frozen except for bug fixes. But we wanted to get the new file abstraction and other new features out to our users soon, so we just bumped the library version to 3.5, rolled in the 3.4 updates, and added the new features listed above.

B.1.81 New Functionality in Playground 3.4.0

- Constants for Page-Up and Page-Down keys added to event.h
- TPFHiScores::SetRememberedUserInfo to save persistent user data.

- TPrefs::SetUserStr and TPrefs::GetUserStr to get encapsulated user data.
- TTextGraphic::SetBoldOverride, to set up a font that can be used as the font for bold text.

B.1.82 Fixes and Minor Changes in Playground 3.4.0

- TWindowNotify gets a proper virtual destructor

B.1.83 New Functionality in Playground 3.3.0.4

- Add an ActiveX test command line parameter: –axtest brings up your game small for testing.
- Added fixbyteorder.h to include folder to allow client code to support cross-platform save and networking.
- Add new samples folder with anitest and TSprite/TAnimatedSprite/TAnimatedTexture/TDrawSpec source.
- Updated documentation.
- Added 3x3 matrix to mat.h.
- Added sidewalk.exe animation creation tool.
- Added dirSync.exe, SDK synchronization tool.
- Changed TVec3(TVec2(),z) constructor to have a default z parameter.

B.1.84 Fixes for Playground 3.3.0.4

- Fixed an ActiveX scaling bug when rendering transformed lit vertices.
- Fixed some MSN ActiveX integration problems.

B.1.85 New Functionality in Playground 3.3.0.3

- Alt-F1 brings up Playground version # and frames-per-second

B.1.86 New Functionality in Playground 3.3.0.2

- Debug output that identifies the video card for improved customer issue tracking.

B.1.87 Fixes and Minor Changes in Playground 3.3.0.2

- Fixed problem where forcing alpha wasn't working if you loaded a PNG with no alpha.
- Made message passing to Lua GUI thread more aggressive, which improves GUI performance in some circumstances.
- Fix problem that allowed a button sound to trigger after the button had been disabled.
- DrawVertices lighting fix.
- Fixes to ActiveX to support DrawVertices calls.
- Eliminate spurious warnings for a call to FillRect with an empty rect.
- Fix software mouse rendering in cases where screen is not entirely updated and normal Playground render path isn't used.

B.1.88 New Functionality in Playground 3.3.0

- New file: mat.h - defines TMat4 class (replaces old matrix.h file and matrixstack.h file)
- New file: vec.h - defines TVec2, TVec3, TVec4

B.1.89 Fixes and Minor Changes in Playground 3.3.0

- Text calibration fix where text got cut off by 1 pixel
- Fixed bug where child windows bigger than their parents caused a drawing issue
- Fixed bug where SetWindowSize() would not always properly clip the window.

B.2 Changes to Playground 3.2.1 from Playground 3.2.0

1. New Functionality in Playground 3.2.1
2. Fixes and Minor Changes in Playground 3.2.1

B.2.1 New Functionality in Playground 3.2.1

- Added xsellkit addon for creating Cross Sell screens in games.

B.2.2 Fixes and Minor Changes in Playground 3.2.1

- Prevent some non-critical warning messages from flooding the log file in certain circumstances.
- TSimpleHttp threading fixes.
- TWindow close button bug fixed.
- Bug in Lua Pause() fixed.
- ActiveX fixes for FillRect and TTexture::Draw.
- Allow ActiveX games to run in software rendering mode when there is no fallback option.
- ActiveX fix where right mouse button could crash Internet Explorer.
-

B.3 Changes to Playground 3.2.0 from Playground 3.1.5

1. New Functionality in Playground 3.2
2. Fixes and Minor Changes in Playground 3.2

B.3.1 New Functionality in Playground 3.2

- Behavior of TWindowManager::SetCapture changed to search current list of capture windows for Set...() and Release...() so that you can't have windows fighting for capture and ending up on the stack A-B-A-B- etc. It then broadcasts all mouse events to all windows in the capture chain, i.e., make SetCapture() a request to be a mouse-message-listener.
- kBlendLit is now marked as deprecated, and is a synonym for kBlendNormal. This is to ensure that alpha blending works on all graphics cards. This means that vertex colors must be specified for all vertices drawn with DrawVertices().
- Removed - unused TTexture::DrawSprite flags eBottomHalf and eTopHalf.
- Multiple app instances are allowed if "–multiple" is passed on the command line.

B.3.2 Fixes and Minor Changes in Playground 3.2

- typo fixed TSimpleXml::GetNextAttriubte renamed to TSimpleXml::GetNextAttribute
- Fixed potential crash when modal window stack is empty
- Fixed italic/underline text combination bug
- Fixed really long text underlining bug.
- Fix for "face" font tag.
- Fixed FlushMessages() Lua command

- Fixed crash in script, if the script fails to run.
- FillRect clipping fixed.
- Fixed assert in TTexture::DrawFast that was firing incorrectly.
- Fixed crash that occurs when font is missing.
- Fix for DirectX filtering initialization in Begin2D()
- Fix to TText::SetScroll to work better with last line of text.
- Fix for characters in TTextEdit fields that are not low ASCII.
- Fix for handling a NULL script file without crashing
- Fix fallback method in TTexture::DrawFast()
- Fix cropping of rectangle in TTexture::DrawFast()
- Fix TImage to correctly choose DrawFast() when it's possible to use in lieu of DrawSprite().
- Fix out-of-memory 16-bit fall back condition
- Bring up message box to let users know when shifting to 16 bit mode.
- Record when app has shifted to 16 bit mode, so app does not try to shift each time the app runs.
- Fix for inappropriately picking 8/3/3/2 A/R/G/B modes
- Better documentation for TStringTable
- Fix for rarely used code path in TTexture::Draw()
- Fix case where mouse would not update on second monitor.
- Fix case where mouse cache was failing.

B.4 Changes to Playground 3.1.5 from Playground 3.1.4

- Bug fixes for DisplaySplash.

B.5 Changes to Playground 3.1.4 from Playground 3.1.3

- Added TTextGraphic::SetNoBlend to resolve rendering text to texture bugs.

B.6 Changes to Playground 3.1.3 from Playground 3.1.2

- Clipping of bad viewports to prevent DirectX errors
- Fix sound looping bug when looping sounds that are in memory

B.7 Changes to Playground 3.1.2 from Playground 3.1.1

- More aggressive message dispatch strategy to speed up Lua responses to events.

B.8 Changes to Playground 3.1.1 from Playground 3.1.0

- Disable log file for ActiveX mode
- MSNZone fixes, documentation
- Fix sub-pixel rendering when calling DrawSprite()
- ActiveX window size fix for running on machines with 800x600 resolution
- Allow negative viewports
- ActiveX text scaling fix
- Fix Unlock() bug for 16 bit textures that did not have alpha
- Fix Unlock() crash for 16 bit textures with 1x1 dimensions

B.9 Changes to Playground 3.1 from Playground 3.0.x

1. Changes in Playground 3.1
2. Compatibility fixes in Playground 3.1

B.9.1 Changes in Playground 3.1

- Major text rendering optimizations made
- Added TPlatform::SetFilteringMode() for texture filtering.
- Added TPlatform::HideCursor()
- Added TSound::SetPostion()
- TTextGraphic::Draw can now take in an optional parameter to specify a target texture to render the text into.
- Added optional parameter to TTexture::DrawSprite for a source rect.
- Added TTexture::HasAlpha
- File loading optimizations made.
- Fix memory leak involving lua tables
- Fix text layout problem when text uses outlines
- TXmlNode destructor made virtual

- Various mouse cursor bugs fixed, including dual monitor support and invalid mouse cache issues.
- Fix issue where default buttons were still considered defaults even if they were not enabled
- Fix crash that would occur if two button messages were triggered before window stack was updated.
- on parameter added to button creation via Lua
- Fixes made to mouse tracking/hovering
- Added Text::GetMaxScroll
- Added TTextGraphic::GetStartLine
- Added support for tabs in TTextGraphic tagging
- Fixes to TTextGrahpic::GetMaxScroll
- Made \ an escape character for TTextGraphics.
- Fix to text picking where it was ignoring the line padding parameter.
- Fix viewport clipping issue in ActiveX mode.
- TPrefs and TPfHiscores now have an optional parameter to disable file saving.
- Fixed issue where windows would flicker when a window went outside the 800x600 screen.
- MSNZone support added.
- Added axtool utility for support in developing web games.
- Added TImage::GetScale()
- Fix for crash when SetCursor is called during application shutdown.
- Fix for TTextGraphic::GetTextBounds () in case where the string is empty
- Fix crash for TPlatform::OpenFile when file name is empty.

B.9.2 Compatibility fixes in Playground 3.1

- Fix various software cursor compatibility issues, including disabling software cursor support on machines that do not report allowing color key capabilities.
- Fix for full screen conflict where Playground would try to keep app on top of a window that wasn't visible, causing unpredictable performance.
- Fix for full screen performance issue where Playground was running slower than it should have been.
- Fix to screen saver installation on Windows 98.
- Playground now requires that a hardware renderer be found in order to run.

B.10 Changes to Playground 3.0 from Playground 2.3.x

1. Changes in Playground 3.0
2. Compatibility fixes in Playground 3.0

B.10.1 Changes in Playground 3.0

- Text drawing fixes for ActiveX mode, text rotation, and some text optimization.
- New Window{} functionality for specifying non-functional windows in LUA
- Added mask parameter to Bitmaps in lua for specifying alpha masks.
- Fix problem with looping sounds that are shorter than 1 second long.
- Fix scaling the center location in DrawSprite.

B.10.2 Compatibility fixes in Playground 3.0

- Require that video cards allow 1024x1024 textures
- Display restore fixes for low end cards
- Sound callback crash fixes

Appendix C

Annotated Class Listing

C.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

ParticleMember (A particle member value) . . . 325
str (Reference-counted string class) . . . 163
T2dParticle (Basic Particle Values) . . . 174
T2dParticleRenderer (A particle renderer that expects 2d particles) . . . 175
TAnimatedSprite (A TSprite with an attached TScript) . . . 178
TAnimatedTexture (This class encapsulates the concept of an animated texture) . . . 185
TAnimTask (The TAnimTask interface) . . . 194
TAsset (The interface class for game assets) . . . 197
TAssetMap (A collection of assets that simplifies asset reference-holding) . . . 198
TBegin2d (Helper class to wrap 2d rendering) . . . 202
TBegin3d (Helper class to wrap 3d rendering) . . . 203
TButton (Encapsulation for button functionality) . . . 204
TButton::Action (An abstract action class for button actions) . . . 213
TButton::LuaAction (A class that wraps a Lua command in an action) . . . 214
TClock (Encapsulates timer functionality) . . . 216
TColor (An RGBA color value) . . . 218
TColor32 (A 32-bit platform native color value) . . . 221
TDialog (A generic modal dialog) . . . 223
TDrawSpec (2d drawing parameters for use in DrawSprite) . . . 225
TEncrypt (A class that encapsulates an encryption engine) . . . 229
TEvent (System event encapsulation) . . . 231
TFile (A file reading abstraction) . . . 234
TFlashHost (An embedded Flash-playback routine) . . . 241
TGameState (An object that allows the game to communicate state information to the system) . . . 244
TGameStateHandler (A handler for game state actions) . . . 250
TImage (TWindow that contains and draws a TTexture) . . . 252
TLayeredWindow (A TLayeredWindow is a TWindow with multiple layers which can be switched between) . . . 256
TLight (A 3d light) . . . 259
TLitVert (3d untransformed, lit vertex) . . . 261
TLuaFunction (A wrapper for a Lua function) . . . 262
TLuaObjectWrapper (Wrap a Lua object for use within C++ code) . . . 264
TLuaParticleSystem (A particle system driven by Lua scripts) . . . 267
TLuaTable (A wrapper for Lua table access in C++) . . . 273
TMat3 (2d Matrix with 2x2 rotation component and TVec2 offset component) . . . 282

TMat4 (3d Matrix with 3x3 rotation component and TVec3 offset component) . . . 290
TMaterial (A rendering material) . . . 299
TMessage (Application message base class) . . . 300
TMessageListener (A message listener–a class that you override and register with the TWindowManager if you want to listen for broadcast messages) . . . 303
TModalWindow (Base class for any window that can be a modal window) . . . 304
TModel (A 3d model) . . . 310
TParamSet (A set of parameters or return values, depending on context) . . . 314
TParticleFunction (A user data source) . . . 318
TParticleMachineState (The internal state of a TLuaParticleSystem) . . . 321
TParticleRenderer (The abstract particle renderer class: This class is used by TLuaParticleSystem to wrap an actual particle renderer) . . . 326
TParticleState (A particle state) . . . 328
TPfHiscores (TPfHiScores - class that manages local and global hiscore saving and viewing) . . . 331
TPlatform (The platform-specific functionality encapsulation class) . . . 342
TPoint (2d integer point representation) . . . 357
TPrefs (Designed to help with the saving of preferences for a game) . . . 358
TPrefsDB (Designed to help with the saving of preferences for a game) . . . 363
TRandom (A deterministic random number generator) . . . 367
TRect (A rectangle) . . . 370
TRenderer (The interface to the rendering subsystem) . . . 376
TScreen (The base level modal window) . . . 395
TScript (An encapsulation for a Lua script context) . . . 397
TScriptCode (An encapsulation of a compiled Lua source file) . . . 406
TSimpleHttp (Implements a basic HTTP connection) . . . 408
TSlider (Slider class) . . . 412
TSound (Object that can play a sound asset) . . . 417
TSoundCallBack (TSoundCallBack –a class that you override and attach to a TSound if you want to know when the sound has finished playing) . . . 420
TSoundInstance (An instance of a sound) . . . 421
TSoundManager (Controls access to the sound subsystem) . . . 424
TSpeex (Interface for class TSpeex) . . . 426
TSprite (A 2d sprite object) . . . 427
TStringTable (The interface class for a string table) . . . 435
TTask (The task interface) . . . 437
TTaskList (A list of TTask-derived objects) . . . 439
TText (A text window) . . . 441
TTextEdit (Editable text TWindow) . . . 448
TTextGraphic (Formatted text class) . . . 455
TTextSprite (A 2d text sprite object) . . . 464
TTexture (This class encapsulates the concept of a texture) . . . 473
TTransformedLitVert (2d Transformed and lit vertex) . . . 482
TURect (A TRect that's forced to be unsigned at all times) . . . 483
TVec2 (A 2d vector class) . . . 485
TVec3 (A 3d vector class) . . . 491
TVec4 (A 4d vector class) . . . 498
TVert (3d untransformed, unlit vertex) . . . 505
TVertexSet (A helper/wrapper for the Vertex Types which allows TRenderer::DrawVertices to identify the vertex type being passed in without making the vertex types polymorphic) . . . 506
TWindow (Base class of any object that needs to draw to the screen) . . . 509
TWindowHoverHandler (A callback that receives notification that a window has had the mouse hover over it) . . . 531
TWindowManager (Manages, controls, and delegates messages to the window system) . . . 532
TWindowSpider (A class used with TWindow::ForEachChild to iterate over the children of a window with a single "callback" function) . . . 541

TWindowStyle (An encapsulation of a Lua window style) 542
TXmlNode (Limited XML parser) 546

About the Author

Tim Mensch has honed his library-design skills over more than 20 years in the games industry, working with companies such as Lucasfilm Games, Disney Interactive, Sega, Maxis, Velocity, Hasbro Interactive, Digital Eclipse, 3d6 Games, and Activision, and also running his own development house. Now he is the lead architect of the Playground SDK™. Tim has a degree in cognitive science from the University of California at San Diego, and now lives in Boulder, Colorado, with his wife and their daughter. In his free time he enjoys his family, seeks technological solutions to the world's problems, works on his digital photography skills, and plays badminton and ultimate frisbee.