PopCap Game Framework

Proprietary and Confidential
Revision 2
January 26, 2005

Overview

The PopCap game framework, named SexyApp Framework, is a flexible high-level library that provides commonly required functions and reusable components. The framework is designed to facilitate rapid development of high-quality games by allowing game programmers to concentrate their efforts on expressing game concepts while minimizing the work required to create a rich visual and audio presentation. This framework and its predecessors have been used for all of PopCap's Deluxe games, which account for over 100 million framework-derived game units downloaded.

Technical Basis

The framework is written in C++, specifically for compatibility with Visual C++ 6.0 and Visual C++ .NET. The framework targets DirectX 6 and above on Windows 95 through XP, and uses the third-party library BASS for music and additional sound support. The rendering architecture supports two modes, a hardware accelerated mode for modern effects and a 2d non-accelerated compatibility mode for older computers. Both modes use the same basic interface for drawing, but the underling hardware accelerated mode uses Direct3D7 and the non-accelerated mode uses custom DirectDraw rendering code that internally writes directly to the bits of a system memory off-screen surface. Both modes support alpha source, alpha destination, alpha modulation, color modulation, additive blending, rotation, line drawing, filled polygons, and both bilinear and nearest-point scaling. Hardware accelerated mode additionally supports textured polygons, which are too slow to be included in non-accelerated mode.

Coding Philosophy

The framework differs from many other APIs in that some class properties are not wrapped in accessor methods, but rather are made to be accessed directly through public member data. The window caption of your application, for example, is set by assigning a value to the std::string mTitle in the application object before the application's window is created. We felt that in many cases this reduced the code required to implement a class. Also of note is the prefix notation used on variables: "m" denotes a class member, "the" denotes a parameter passed to a method or function, and "a" denotes a local variable.

Framework Structure

Applications using the framework will derive from the core class, SexyAppBase. SexyAppBase contains many helpful methods and objects and can be seen as the core which your game is built around. One of the primary objects used to control the display and game logic is SexyAppBase::mWidgetManager, which is responsible for maintaining a list of Widgets, similar in concept to a window in a windowing system. A Widget could represent a button on a dialog or even the entire game. SexyAppBase also includes other manager classes for controlling multimedia functions such a sound effects and music.

Widget Details

All widgets are derived from the base Widget class. This Widget class contains virtual methods than can be overridden for handling, among other things, drawing, game logic, keyboard and mouse handling.

Location

Widgets can be positioned anywhere within the main game window and in any order that makes sense. The Z order is implicitly created when new Widgets are added to the WidgetManager using WidgetManager::AddWidget, but there are a number of calls to move the Widgets around on the X Y and Z axis.

Widget Hierarchy

The framework doesn't formally support Widgets parenting other Widgets. This functionality can be mimicked, however, by having a parent Widget that constructs other Widgets and moves them around in response to its own movement and such.

Updating

The Update method of all the widgets gets called by the WidgetManager at a fixed rate. The default rate is 100 times per second. This will remain constant regardless of the computer speed or video refresh rate. All game logic that progresses at a fixed rate should take place in the Update method. For example, if you have a box that moves across the screen at a rate of 100 pixels per second, you should increment its position by 1 pixel each time your Widget's Update method gets called.

Drawing to the Screen

When the visual state of your Widget has changed (in response to the player moving or an animation updating, for example), the Widget should be marked as

dirty by calling Widget::MarkDirty to indicate that it should be redrawn. When MarkDirty is called, it may trigger other widgets to redraw depending on the position of the Widget. If other Widgets appear over the Widget being redrawn, for example, they will need to be redrawn since the drawing of the first Widget will write over pixels owned by the Widget on top.

The redrawing will occur after all the widgets have been brought up-to-date by having their Update methods called. The Widgets will have their Draw methods called in back-to-front order, with a Graphics object being passed in. The Graphics object represents a drawing destination, and contains all the drawing methods available to the application. All drawing coordinates will be relative to the Widget's coordinates and will be clipped to the Widget's extents.

Update / Draw Independence

Ideally, you should imagine that you app will always have it's Updates called 100 times per second but may NEVER have it's Draw methods called. Regardless, the game state should proceed EXACTLY as it would as if Draw was getting called 60 or 80 times per second. Always think to yourself, "if I had missed Draw for the last 5 seconds and then got one right now, would I be drawing exactly the same thing to the screen as if I had gotten one Draw for every Update up until now?" If your answer is no, you should think about redesigning your code path as you could be introducing timing issues that could be hard to track down and fix later. Furthermore, it could break the helpful demo recording capability built into the framework.

Smooth Updating

Smooth updating is an advanced optional feature that is unused in nearly all of our games, but can help achieve smoother effects in some cases, particularly for side-scrolling games. The core problem that smooth updating overcomes is the temporal aliasing between updating game logic at 100 HZ and a monitor refresh rate that can run anywhere between 60HZ to 85HZ (and sometimes even higher). The goal is to sync the movement of objects in the game to the monitor refresh rate while still allowing us the simplicity of having a set update rate for most of the game logic.

The solution consists of having two separate update calls for a widget, Update (which is called at 100HZ), and UpdateF (which is called at the monitor's refresh rate). UpdateF has a floating point number passed in, which represents how many 100HZ units that one call represents, which will always be between 1 and 1.67. Because the monitor refresh rate will vary on different computers you will have to take that foating point number into account in every time-based calculation that occurs in UpdateF, such as motion. While that makes the implementation of the UpdateF call somewhat less convenient than the fixed-rate Update call, you may mix logic amongst the two to get the best of both words. To ensure good

behavior, the framework promises that there will always be either one or two Update calls between each UpdateF call – there will never be two UpdateF calls without an Update call between them. That means that you can update critical object and scrolling positions in UpdateF but still rely on Update for collision detection, animation, and other game logic. Smooth updating mode is enabled by setting SexyAppBase::mVSyncUpdates to true.

Deleting Widgets

A deleting issue is often encountered with widgets where you may want a widget to delete itself, either directly or indirectly. Take the example of a dialog box: if you create a DialogBox widget that you want to remove when you click on it, you'd override DialogBox::MouseDown, first removing the widget from the widget manager and then needing to delete it, but directly deleting the widget at that point could cause a crash. In order to get around that, call SexyAppBase::SafeDeleteWidget, which will insert the widget into a deferred list that will be deleted at a safe time.

Images

The base image representation in the framework is the abstract class Image, but in practice all images will be a MemoryImage or a DDImage. A DDImage is derived from a MemoryImage but differs in the respect that it can point to a DirectDraw surface whereas a MemoryImage contains a raw copy of the pixels in either 32-bit ARGB or palletized format.

Loading Images

Image loading can be done at any point, but should normally occur at program startup, either directly through SexyApp::GetImage or indirectly through the ResourceManager (described later). SexyApp::GetImage load a DDImage for you from a PNG, GIF, or JPEG file. While PNG is the only format that directly supports alpha channels, the framework will look for a black-and-white "alpha channel image" with the same name except with an underscore prepended or postpended to it.

In this example, the color channel is stored in "swapper.jpg" but the alpha channel is stored in "_swapper.gif". The combined image can be loaded with a single call to SexyApp::GetImage("swapper")

_swapper.gif

swapper.jpg

Modifying Bits

MemoryImages are set up to be easily modifiable to create programmatic effects. GetBits() can be called to get an unsigned long pointer to the raw image data in 32-bit ARGB format. Simply modify the bits and call BitsChanged to commit the new data to the image.

Memory Consumption

Loading lots of large images can quickly add up in terms of memory usage. Pressing the F11 key while in debug mode (enabled with ctrl-alt-d) will create a "dump" directory containing PNG copies of all images and a HTML index with a list of how much memory each image takes up. To get an idea of how much memory an image can take, a 640x480x32 image will take nearly 1.2MB of memory for just the raw bits, plus more for native pixel tables and run-length alpha encoding tables (performance-enhancements that are taken care of behind-the-scenes). The per-image memory cost can be reduced if it can be internally palletized (if it contains 256 colors or less). Use the MemoryImage::Palletize method to convert a memory into its palletized representation.

Drawing

All drawing to the screen is done through widgets added to the widget manager in SexyAppBase::mWidgetManager. The widget manager calls the widgets' draw methods, passing in a Graphics context that can be used to draw to the screen.

Clipping Regions

The widget manager sets the clipping region on the graphics context before passing it into a widget's Draw method. Initially the clipping region is the rectangular extents of the widget, but can be further reduced by calling Graphics::ClipRect. If Widget::mClip to false, the widget's Draw method will be called without a clipping region set.

Drawing Performance

Performance should be monitored in non-accelerated mode to ensure that the final product will achieve reasonable frame rates. Things that may affect performance are: large areas of alpha or additive drawing, tons of tiny images (as with every graphics platform there is some overhead to drawing), too much overdraw, or overuse of rotating images.

In Bejeweled 2 we have a board with a large alpha area behind the gems that was being drawn over a planetary backdrop but we were able to avoid the alpha and overdraw penalty by composing a single fullscreen image that combined the backdrop and the board into a single image that is drawn under the gems every

frame. You can draw to an image by constructing your own Graphics context pointing to your image.

Image Scaling

There are two scaling modes available, set by calling Graphics::SetFastStetch. Fast stretching using nearest-point sampling and is acceptable for real-time non-accelerated use. Setting fast stretching to false enables bilinear stretching, which is only appropriate for accelerated drawing or for caching a resized version of an image in a new image at load time (or some other time where some slowdown is acceptable).

Smooth Image Movement

The framework supports sub-pixel movement through Graphics::DrawImageF, which accepts floating point coordinates. Unless used only on a small quantity of small images, DrawImageF should only be used in hardware accelerated mode.

Textured Polygons, Advanced Hardware Accelerated Support

In hardware accelerated mode you can get more direct access to the Direct3D layer through SexyAppBase::mDDInterface->mD3DInterface. The D3DInterface exposes calls like DrawTriangleTex, which can be used for drawing arbitrary textured triangles, using an image for the texture. You can also directly access the D3D, D3DDevice, and D3DViewport objects through the D3DInterface, but you must keep in mind that internally an image doesn't map one-to-one to a texture since an image may be broken into multiple textures to account for power-of-two limitations on texture sizes. The algorithm for breaking an image into textures is optimized for memory and texture swapping, but you can set the D3DImageFlags_MinimizeNumSubdivisions flag in MemoryImage::mD3DFlags to break the image into fewer textures, optimizing for speed in essence.

Managing Hardware Acceleration

By default, hardware acceleration is disabled in the framework. To turn it on, set SexyAppBase::mAutoEnable3D to true in your application's constructor. This will cause a 3D test to occur when the user first runs your application, categorizing the user's computer into three acceleration categories: recommended, supported, and unsupported. Hardware accelerated mode will be automatically enabled if for 'recommended', the user will have to manually turn it on for 'supported', and 'unsupported' will not allow the user to enable it at all. To turn on hardware acceleration for 'supported' computers, call SexyAppBase::Set3DAccelerated.

Often you'll want to add some enhanced effects for computers that support hardware acceleration. Often in those cases you can just branch your widget's

drawing logic depending on the result returned from SexyAppBase::Is3DAccelerated.

Sounds

Sounds are loaded via the SexyApp::mSoundManager interface. SoundManager::LoadSound will load a sound into one of the source sound channels. SexyApp provides a helper method for playing any loaded sound through SexyApp::PlaySample. If you want more control over a sound instance, you can call SoundManager::GetSound instance to get access to a SoundInstance. You can then access the SoundInstance directly to perform such actions as playing it, releasing it, checking its state, or changing its volume or pan.

MP3 and OGG Decoding

MP3 and OGG support are included in the framework. Whenever a MP3 or OGG file is loaded, the uncompressed version will be cached in a "cached-?.wav" file for faster loading the next time. As with the image loader, no extension needs to be specified for the sound file name.

Playing a repeating sound

Sounds can be played in a fire-and-forget fashion by simply calling SexyApp::PlaySample(), passing in the sound id. You'll need to deal directly with sound instances, however, if you want more control over a sound such as being able to loop sounds, stop them in progress, or change their panning and volume settings while they are playing. Allocate a sound instance by calling SexyAppBase::mSoundManager->GetSoundInstance. Sound instances should never be deleted, only released. You release a sound instance by either directly calling SoundInstance::Release or by allowing the sound to auto release after playing by calling SoundInsatnce::Play with the autoRelease parameter set to true.

Music

Music support is provided through both the BASS interface, which offers support for MOD, XM, IT, and MO3 "tracker" files. MO3 is the most advanced format and produces the smallest files due to MP3 compression of samples.

Loading Music and Playing

Music is organized by channels, where each channel contains its own instance of a song file, has its own volume setting, and can be played and stopped independently from the other music channels. To load and play a music file in

channel 0, for instance, call mMusicInterface.LoadMusic(0, "music.s3m") then mMusicInterface::PlayMusic(0).

Cross-fading Between Songs

In order to cross-fade between songs, you must have at least two channels that have music loaded into them since you will need one channel to fade out while the other one fades in. If all of your music tracks are contained in one music file that means you will have to load the same music file twice (or more). To fade out the old song, call MusicInterface::FadeOut. There is a "stopSong" parameter passed into this call, which will determine whether the music actually stops when it fades all the way our or silently continues so you can fade it back in from it's current position later on. If you stop the song, it will start from the beginning next time you fade it in.

Fonts

The framework supports its own format of fonts, defined by a font descriptor. The font descriptor is a human-readable modifiable text file describing the characters included in the font, the image information about each character, and other font stuff like kerning information.

Creating Fonts

FontBuilder is a program that allows you to convert any TrueType font installed in Windows to a framework font file. Along with the standard font controls there are settings for padding, which allows you to build a couple of blank pixels around the edges so you can load the font image in Photoshop and use image manipulation that may make the character become larger than normal.

Initialization and Resource Loading

The SexyApp::Init method is called upon application initialization, allowing for loading of resources required before the loading progress screen is displayed. After initialization, a resource loading thread is started, which calls SexyApp.LoadingThreadProc. Typically, a game will load only the resources required to show the loading screen in SexyApp::Init and load everything else in the LoadingThreadProc. The LoadingThreadProc should keep track of roughly what its completed percentage is so the title screen can display a progress bar.

Multithreading Considerations

Some caution should be used when making calls in the loading thread to ensure that there are no threading conflicts. While loading images and sounds is safe,

you wouldn't want to be creating Widgets and adding them to the WidgetManager or anything crazy. Also, caution should be used if you create your own meta-resource manager that the title screen or Widgets that appear pre-load access while the loading thread is trying to add newly loaded resources to it. Even having a simple image pointer vector where images are pushed onto when they are loaded could cause problems if the title screen is accessing the vector at the same instant the LoadingThreadProc is adding to it.

ResourceManager

In addition to directly loading sounds and images in the framework, there is a ResourceManager that can load in resources such as images, sounds, and fonts based on information in an XML file. The ResourceManager system relies on the ResourceGen.exe program to parse the XML file to create C++ support code to make the included resources visible to the program. See the document "Using PopCap Resource Manifests.doc" for more information.

Dialogs

In order to create arbitrarily-sized dialogs, we use a "skinning" technique where the source dialog image is logically split into 3x3 sections. Each of the 4 corners of the 3x3 section is drawn in the appropriate corners of the destination dialog area, and then the areas between each of corners are repeated over the remaining area of the dialog. Dialog buttons work in a similar fashion, except they only tile horizontally so they are split up into logical 3x1 sections.

General Program Stuff

In addition to the multimedia capabilities, the framework provides much of the basic 'glue' necessary to build a game. Here are some of the basic issues.

Saving / Loading Settings

The framework automatically saves some settings to the registry such as volume levels and window position. To save your own settings to the registry you can override SexyAppBase::WriteToRegistry and SexyAppBase::ReadFromRegistry. Look at the Registry* calls in SexyAppBase.h to see the complete list of registry calls.

Handling Command Line Parameters

Override SexyAppBase::HandleCmdLineParam to handle custom command line parameters. Both a name and a value are passed in, allowing you to specify parameters in the format of "name=value".

Saving and Loading Files

Files can be easily saved and loaded through the Buffer interface. To write data, construct a Buffer and write to it using any of the Write* calls, then save the Buffer to a file calling SexyAppBase::WriteBufferToFile. To read data, call SexyAppBase::ReadBufferFromFile to put the file data into a Buffer and then use any of the Buffer's Read* calls to extract it.

Windows Message Boxes

SexyAppBase::MsgBox can be called to show standard windows message boxes. These should be used for debug uses only.

Debugging

Often times, programs are not perfect the first time and require some amount of debugging, both before and after release. The framework provides some assistance in that area. When a crash occurs it will be caught by the Structured Exception Handler, where a dialog will be shown to the user containing details of the crash such as a stack trace, program-configurable output, and build information. The crash information can be loaded into MapLookup along with the appropriate map file to look up function names from the stack trace. It's recommended that you create map files with line information and that you set "Omit Frame Pointers" to "No" in your projects C/C++ Optimization settings, as frame pointer omission interferes with the ability to generate a stack trace.

Debugging Mode

Debugging keys can be enabled and disabled with ctrl-alt-d, which toggles the SexyAppBase::mDebugKeysEnabled flag. The currently supported debug keys are:

F2 - Start/Stop Perf Timing

F3 - toggle framerate display
Shift F3 - toggle framerate/mouse coord display

F8 - Show current Video Stats (mostly used to see if 3d is currently on)
Shift F8 - Toggle 3d mode

F10 - Single Step Program (show one frame at a time)
Shift F10 - Stop single steping
Ctrl F10 - Toggle Slow Motion

F11 - Take Screenshot (goes into the _screenshots) directory

Shift F11 - Dump all program images in memory to the _dump directory

A common technique at PopCap is to extend those debug keys by overriding SexyAppBase::DebugKeyDown, providing pre-release cheat keys to help QA in testing.

Performance Profiling

Performance monitoring code is contained in PerfTimer.h. Code you want to profile should generally be surrounded by SEXY_PERF_BEGIN/SEXY_PERF_END macros, and you must set the SEXY_PERF_ENABLED preprocessor define before including PerfTimer.h. Press F2 once to start profiling and press it again to view the results.

Demo Recording

Demo recording writes timing information and program input to a repayable demo file that, if the program is set up properly, will result in the same program state progression as the original session when played back. This requires adhering to a set of rules to ensure deterministic behavior. While even small deviation from this set of rules could create demo playback problems that are hard to track down, the process has gone fairly smoothly in practice. To record a demo, pass "-record" into the program or set SexyAppBase::`mRecordingDemoBuffer` to true. Play back the demo by passing "-play" into the program. Demo support is optional and can be ignored.

Common Problems

P: The background isn't drawing properly under my widget, or alpha/anti-aliased regions are drawing darker than they should.

S: When you call MarkDirty, the framework will try to redraw as few widgets as possible. If your widget is completely opaque there will be no need to redraw widgets directly under it, but if your widget contains alpha portions then the widget under it must be redrawn first. Setting Widget::mHasAlpha on widgets with alpha will fix the problem.

P: After deleting a widget my program is crashing.

S: You must make sure you remove the widget from the SexyAppBase::mWidgetManager with WidgetManager::RemoveWidget before deleting it. Also, you may need to call SexyAppBase::SafeDeleteWidget instead of deleting the widget directly if the widget is still in the call stack (as in the case of a widget deleting itself when you click on it, for example).

P: My program takes too much memory.

S: Lots of image data can add up. You can easily see what's loaded by enabling debug keys and using F11 to dump image information to a "_dump" directory. Reducing appropriate images to a 256-color palletized format and keeping them in memory only when needed is also advised.