PopCap Games Framework Version 1.2 Changes

Overview

There have been a number of changes to the framework since version 1.0, much of which has been in the area of widgets and graphics. The big change in widgets is the formalization of widget hierarchies, which allows any number of child widgets to be parented by any widget. This makes it easier to create widget-based user interfaces than before. The big changes in graphics include a non-hardware-accelerated textured triangle rasterizer and an overall decrease in the memory that images consume.

There have been a number of changes in just about all of the major framework sections, most of which are small, but hopefully add up to the framework being even friendlier than before.

Backwards Compatibility

Version 1.2 is backward-compatible, except DDImage::OnlyKeepDDSurface has been replaced by a more generalized MemoryImage::PurgeBits. Example:

Version 1.0:

```
if (mApp->Is3DAccelerated())
	mNextBackdrop->mOnlyKeepDDSurface = true;
else
	mNextBackdrop->OnlyKeepDDSurface();
```

Version 1.2:

```
mNextBackdrop->mWantDDSurface = true;
mNextBackdrop->PurgeBits();
```

App Changes

Many of the image-based SexyAppBase methods that construct images (such as SexyAppBase::GetImage) have been modified to directly return DDImages rather than Images. This should avoid some confusion and allow you to avoid feeling uncomfortable making “dangerous” upcasts.

Initialization and shutdown has changed slightly, in that there is now a ShutdownHook and an InitHook. Overriding ShutdownHook takes the place of overriding Shutdown, except you don’t have to check mShutdown and call the base implementation, and InitHook gets called by SexyAppBase::Init at the end. In addition, it is now safe to call

Shutdown during the loading thread; doing so will only cause mLoadingFailed to be set to true, which is the standard way to cause program termination from that thread.

Blocking operations are now formally and generally supported by the framework. The only blocking operation allowed before was Dialog::WaitForResult, which returned the dialog result once the dialog was closed. Blocking operation support has been generalized to simplify some types of time-based tasks by allowing state to be maintained within local variables (and by nature of code execution location) rather than forcing widgets to explicitly store that state information across calls to Update. The obvious use is for dialogs, but it can also be used for transitions or other animations (for example, calling a RollDice() method that creates a dice object, animates the rolling, and returns the result). One thing to be careful of is that the program can be shut down at any time, so you must be able to exit gracefully from your blocking calls. SexyApp::UpdateApp returns false if the application is shutting down, and you can detect that the blocking call was terminated prematurely by checking SexyAppBase::mExitToTop. The core of a blocking call should be formatted as such:

```
while ((gSexyAppBase->UpdateApp()) && (mStillPerformingTask))
{
	// This code gets executed once per update
}
return mTaskResult;
```

Or, if you need the termination condition to be evaluated with more granularity because you want to avoid processing update or any other input events if the termination condition is met by a previous input event (such as clicking a button), you can use the following:

```
bool updated;
while ((gSexyAppBase->UpdateApp(&updated)) && (mStillPerformingTask))
{
	// This code gets executed after every input event and after
	//  every update

	if (updated)
	{
		// This code gets executed once per update
	}
}
return mTaskResult;
```

Widget Changes

Full widget hierarchies are now support, which allows any widget to be a parent to any number of child widgets. The child widgets inherit flags from the parent widget, draw relative to the parent widget, and are added and removed from the widget manager along with the parent widget. This means that in a widget's constructor you can allocate a child widget, immediately position it and add it as your child by calling the parent widget's AddWidget method instead of the WidgetManager's AddWidget, and then delete it in the

destructor without having to remove it first – no other child management is required. To facilitate this generalized parenting, both Widget and WidgetManager now share a base class: WidgetContainer. Despite the enhanced functionality, widgets are still fairly light-weight and no changes are required for old widgets to work as they always have.

A "flag system" has been added to Widgets to help simplify some common-usage cases such as always marking widgets dirty (avoiding explicit calls to MarkDirty) and pausing the game when it loses focus. Widget behavior is described by the following flags:

```
WIDGETFLAGS_UPDATE      - Set to receive Update calls
WIDGETFLAGS_MARK_DIRTY  - Set to automatically mark the widget dirty
WIDGETFLAGS_DRAW        - Set to allow drawing
WIDGETFLAGS_CLIP        - Set to clip Graphics passed into Draw
WIDGETFLAGS_ALLOW_MOUSE - Set to allow mouse interaction
WIDGETFLAGS_ALLOW_FOCUS - Set to allow focus to be maintained
```

The only flag not set by default is WIDGETFLAGS_MARK_DIRTY. A top-level widget's flags are based on the initial WidgetManager::mWidgetFlags value, but modified by that widget's parent WidgetContainer::mWidgetFlagsMod value. mWidgetFlagsMod allows you to override flags by specifying which ones to turn off and which ones to turn on. The default setting of mWidgetFlagsMod neither adds or removes flags, so adding WIDGETFLAGS_MARK_DIRTY to WidgetManager::mWidgetFlags will cause **every** widget to automatically mark itself dirty, or you could set one particular Widget's mWidgetFlagsMod.mAddFlags to WIDGETFLAGS_MARK_DIRTY to enable it just for that one Widget. To use more than one flag, use the bitwise OR operator to combine them together.

The flags system now extends to modal dialogs, which are more flexible and well-behaved now. In order for modal dialogs to be modal, focus must be taken away from any widgets under the modal dialog and mouse interaction must not be allowed. This behavior is defined by mDefaultBelowModalFlagsMod.mRemoveFlags, which is initially set to WIDGETFLAGS_ALLOW_MOUSE | WIDGETFLAGS_ALLOW_FOCUS. Any widget can act as a modal widget, however, with WidgetManager::SetBaseModal, and the modal behavior can be extended to not updating or even not drawing widgets beneath the base modal widget by passing in the appropriate FlagsMod to the function. The modal system has been further extended by allowing modal widgets to stack, so a dialog can open another dialog but have its previous modal settings remembered when the second dialog closes. Focus is automatically returned to the formerly-focused widget when a modal widget closes, as well.

Overlay drawing has now been incorporated as a basic widget feature. Most applications have a need for some type of overlay drawing, generally where the game board needs to draw some graphic effects over some other widgets such as button widgets, but it cannot due so because the board is, in fact, beneath the other widgets. Previously, developers would generally create an OverlayWidget that was positioned at the top level which was responsible for drawing the special effects, often by just a simple dispatch to mBoard->DrawOverlay. These overlays can be a pain, however, since they have to be managed separately from the widget that actually wants to draw the overlaid graphics, and the

developer typically has to do extra work to make sure that the OverlayWidget stays properly placed at the top of the widget list (but under dialogs and sometimes transitions, etc).

The new integrated solution is exposed as Widget::DeferOverlay. DeferOverlay schedules a call to Widget::DrawOverlay later on, after other widgets have had a chance to draw. The overlay scheme is based on priorities, where you pass in an overlay priority to DeferOverlay, and every Widget has an mPriority that determines its Draw() priority. When a widget is reached in the drawing cycle that has a higher priority than a deferred overlay, the overlay is drawn before that widget's Draw is called. In the simple normal case, every Widget has a priority of zero except Dialogs, which have a priority of one (these are default). The board will call DeferOverlay() (with the default priority of zero), where the overlay will not get drawn until either a dialog (with the priority of one) is encountered or all widgets have been drawn. Multiple overlays can be scheduled for a single Widget. DeferOverlay should be called in the widget's normal Draw method and not in Update.

Another change is that lazy programmers that don't want to pass around Graphics to things like sprites can get the current graphics context from WidgetManager::mCurG, although it's only valid during widget drawing.

Dialog Changes

Previously, dialog results had to be decoded through button ids passed into SexyAppBase::ButtonDepress, but now Dialogs work with a generic DialogListener. The dialog's DialogListener is specified in Dialog::mDialogListener which defaults to gSexyAppBase. DialogListener includes DialogButtonPress and DialogButtonDepress, both of which receive the dialog id and the applicable dialog's button id. Button ids are 1 for yes/ok, and 0 for no/cancel (or for one-button dialogs). In contrast, by using ButtonPress/Depress to catch the clicks, the ids passed in were the id of the dialog box plus 2000 for yes/ok and plus 3000 for no/cancel..

Dialogs also now include default drawing implementation so you can include dialogs in quick mockups without requiring extra graphics or overriding SexyAppBase::NewDialog.

Graphics Changes

There are several new methods in the graphics class that are made possible by the inclusion of a software triangle rasterizer in the framework. The calls are:

DrawImageMatrix
DrawImageTransform
DrawTriangleTex

DrawImageMatrix draws an image with an arbitrary matrix transformation (for instance a scale and a rotation concatenated together).

DrawImageTransform is similar to DrawImageMatrix but it uses the Transform class which keeps track of whether or not the transformation is a simple one which can be accomplished with other graphics calls. So, for instance, if you simply do a rotation transformation, DrawImageTransform will delegate the call to DrawImageRotated. If the transformation can't be accomplished with a more specific graphics call then DrawImageMatrix is used to accomplish the drawing.

Both DrawImageMatrix and DrawImageTransform draw the image relative to its center rather than its upper-left corner.

DrawTriangleTex will draw a textured triangle using the image passed to it as the texture.

Please note that the software triangle rasterizer doesn't have support for additive drawing or linear blending right now, but we plan to add these in the future.

Other graphics changes include a reduction in memory consumption by allowing for some redundant data to be removed from images. In 2d mode, images typically end up containing both the raw 32-bit pixel data plus "native data", which has been converted to the format of the display for faster drawing. In 3d mode, images contain the 32-bit pixel data plus the memory occupied by the textures comprising the image in the Direct3D texture manager. In both cases, the 32-bit raw pixel data can be often considered extraneous. To help reduce overall memory usage, MemoryImage::PurgeBits has been added. This call indicates to the image that you do not need the raw bits around, only the native data. PurgeBits can be accessed through the resource manager by adding "nobits", "nobits2d", and "nobits3d" tags to images in your resource XML file. "nobits3d" means that you want to get rid of the bits only in 3d mode, "nobits2d" means you only want to get rid of them in 2d mode, and "nobits" means to get rid of them regardless of mode. Note that, while purging the raw image bits does reduce overall memory usage, it's unlikely to actually produce any frame rate improvement on low-memory machines, as those unused bits would be paged out by the Windows virtual memory manager anyway. An important thing to note is that you shouldn't purge an image if you are going to be calling GetBits on it, or doing any other operation that as a side effect calls GetBits. Since the bits would have been deleted, the resulting call to GetBits will have to rebuild the data, causing a performance hit. Functions that implicitly call GetBits are as follows:

- Rotational drawing functions (such as BltRotated)
- Stretch blit functions
- Palletize
- And any other function that makes a call to GetBits().

Thus for images that are stretched and rotated in real-time, you should not purge their bits.

In other news, maintaining the graphics state such as the current clipping area, color, drawing mode, scaling mode, and translation is now easier through the inclusion of Graphics::PushState and Graphics::PopState. This can be automated by the GraphicsAutoState object, whereby a GraphicsAutoState can be constructed on the stack of a graphics-related method where the state is pushed in the GraphicsAutoState constructor and popped in the destructor. This allows you to do the following:

```
void Board::DrawObject(Graphics* g, Object* theObject)
{
	GraphicsAutoState anAutoState;

	// Do whatever you want to the state of graphics here, because it
	//  will be restored by anAutoState's destructor when we exit
}
```

Another new graphics method is Graphics::SetClipRect and Graphics::ClearClipRect, which give you more control over the graphics clipping region.

Sound Changes

A few small sound changes have occurred, as well. First of all, oggs and wavs are no longer cached out as wavs in the same directory as the original sounds, but rather are created in a "cached" directory off of the main game directory. This should make some things less annoying such as making it easier to not include the cached sounds when packaging builds.

Another sound change is allowing for dynamically allocated sounds. Dynamic allocation is accomplished through the addition of DSoundManager::LoadSound(theFileName). This will allocate the next available channel, counting down from MAX_SOURCE_SOUNDS. By allocating sounds from the highest channel down, we avoid conflicting with sound channels that may be allocated by the resource manager. Any allocated sound can be freed with DSoundManager::ReleaseSound.

Other Small Fixes / Changes

- Fixed mVSyncUpdates problem with monitors running at over 100HZ
- Fixed crash bug in EditWidget for small text fields (< 10 characters visible)
- During crash in debug mode added "Debug" button to crash dialog so the crash can be tranferred to a debugger (if not already running in a debugger)
- Cursor widgets and transient widgets removed
- The update backlog is limited to 200ms instead of 1 second