

Serenity Code Reference

LView Class Overview

Compatibility: BeOS Dano, yellowTAB Zeta, PhOS, (no R5 support)

Introduction:

The LView class is derived from BeOS's BView. The purpose of the class is to provide a drawing surface for LControl-derived classes, and forwarding system messages, input events, drawing notifications, automatic lazy-clipping, and much more.

LViews, when implemented, will normally not have any children, instead the class will normally be used to display content determined by an LControl. Because LView is a BView-derived class, it overrides many BView virtual function calls (hook-functions), meaning that any classes being derived from LView should ALWAYS issue an inherited::Callback() for each LView function overridden that LView overrides from BView.

Example: (proper header exclusions removed)

```
// LView derivation example:
/*      Header      */
#include "LView.h"

class myView      :      public LView
{
    public:
        myView(BRect);

        virtual void AttachedToWindow();

        ...
};
/* end of header */
```

// Source Code file (myView.cpp)

#include "myView.h"

```
myView    ::    myView(BRect r)
              :LView(r, "myVieworSomething")
              {    // Constructor
              }
```

void

```
myView    ::    AttachedToWindow()
{
    LView::AttachedToWindow();
    // Your Code *AFTER*
}
```

The easiest way to determine if you need to issue a call-back is to look in the **LView.h** header file. Any function you are going to implement that is contained in **LView.h** needs the call-back. You should only override functions that have been declared as **virtual**.

In almost all cases, you will want to issue the callback before any of your own code is executed. This will allow the drawing and LControl control facilities to be utilized properly.

Function-by-Function:

Consutructor:

```
LView(BRect frame,
      const char* name,
      uint32 follow_mask, = B_FOLLOW_ALL
      uint32 flags = B_WILL_DRAW | B_PULSE_NEEDED,
      int32 ramAllocIncrement = 1);
```

frame - Determines exterior bounds of **LView**

name - Name for **LView**, useful for searching

follow_mask - Automatic-resizing to container (Window or view)

flags - determines events to be passed and actions allowed

ramAllocIncrement - how much room to allocate for LControls each time the list will grow when needed by this number to allow more room

Destructor:

```
virtual ~LView();    // frees list, and all members
```

Hook Functions: (from **BView**, see BeBook)

```
virtual void DetachedFromWindow();
virtual void Pulse();
virtual void Draw(BRect);
virtual void MouseMoved(BPoint, uint32, const BMessage*);
virtual voidMouseDown(BPoint);
virtual voidMouseUp(BPoint);
virtual voidKeyDown(const char*, int32);
virtual voidKeyUp(const char*, int32);
virtual voidMessageReceived(BMessage*);
```

Locking:

Locking is generally handled 100% without your knowledge or say so. You may desire to lock this **LView** on your own if you are trying to add and remove controls asynchronously.

The locking only stops operations on the LControl members list.

DO NOT LOCK UNLESS YOU HAVE TO, AND DO IT QUICKLY!

LView will "spin" on the lock every 1000ns until it is released!

```
void Lock();
bool IsLocked();
void Unlock();
void SetFuncLockout(bool = true);
```

If you pass false to SetFuncLockout, all locking is disabled. There are only a few rare cases where the highly-granular locking mechanisms might cause issues. Normally the case will be when and if you decide to control or manipulate **LView**'s members list directly.

Member Handling Functions:

```
void AddMember(LControl*);
int32 CountMembers();// returns 0 if locked
bool RemoveMember(LControl*);// false if locked
bool RemoveMemberAt(int32);
LControl*MemberAt(int32);
BList* Members(); // returns NULL when locked
```