

THE OsRAY++ LIBRARIES

Application Programming Overview

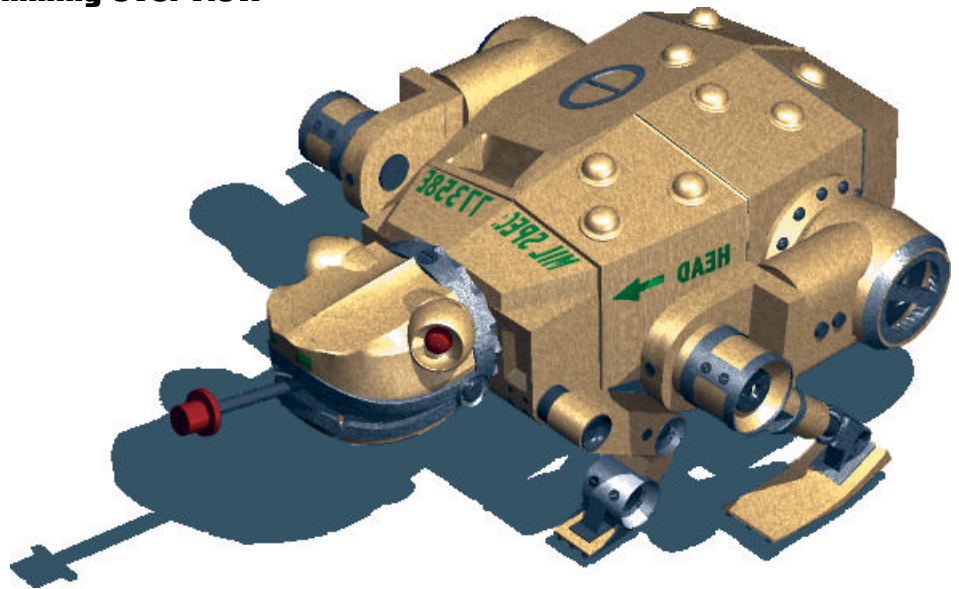
By Ray Tobey

Document revisions:

November 1998

July 2000

August 2001



Contents

OsRay Core Library	42
Visual-C Library	48
Pixel Library	49
Resource Compiler	50
Glazier Editor	51
Glaze Library	52
Pane Library	54
Codec Library	55
Slang Library	56
Storage Library	57

Ray Tobey

709 Old County Road #K
Belmont, CA 94002-2672
(650) 591-0208
rtobey@bionictoad.com



Introduction

OsRay++ is a collection of libraries and tools that provide building blocks for 32-bit Windows programs. In its third generation, this toolset can work with or replace the standard C, C++ and MFC libraries. The libraries consist of the following:

- OsRay Core including strings, containers, file access, threads, timing, etc
- Visual-C Program startup code for use with Microsoft compilers
- Pixel 2D graphics including bitmap blitters
- Scene 3D rendering with a scene graph
- Glaze GUI windows and controls
- Pane GUI sub-components for animation
- Codec Lossless & lossy image and data compression
- Slang Scripting language compiler
- Storage Simplified persistent database

The goal for the tools is to reduce complexity in the libraries by preprocessing data. Therefore, while the libraries represent more than 350 source files, the object code is very small and efficient. The tool programs include:

- Animax Image editor for icons and graphics
- Glazier Window and dialog editor
- Formax File format converter
- Res-Comp Resource compiler to convert editor files to program data

This document describes the basics of many of the classes, but not all. Nor does it cover all the procedures and details of those classes, instead simplifying for clarity.

Coding Style

Types (structs, classes, typedefs, etc.) are in UPPERCASE to stand out. Though preprocessor macros and definitions are very rare, they are also in uppercase. I have avoided Hungarian notation whenever possible, as I believe this defeats the purpose of type checking.

In order to avoid compiler and processor dependent code, Compiler.H defines the following:

```
typedef signed    int    BOOL;                /* boolean (use TRUE or FALSE) */

typedef signed    char    SBYTE;              /* signed 8-bit number */
typedef unsigned  char    UBYTE;              /* unsigned 8-bit number */
typedef signed    short   SWORD;              /* signed 16-bit number */
typedef unsigned  short   UWORD;              /* unsigned 16-bit number */
typedef signed    long    SLONG;              /* signed 32-bit number */
typedef unsigned  long    ULONG;              /* unsigned 32-bit number */
typedef signed    int64    SHUGE;             /* signed 64-bit number */
typedef unsigned  int64    UHUGE;             /* unsigned 64-bit number */
typedef signed    int      INTS;              /* signed number (>= 16 bits) */
typedef unsigned  int      INTU;              /* unsigned number (>= 16 bits) */

typedef float      FLT32;                     /* Lo precision floating point */
typedef double     FLT64;                     /* Hi precision floating point */

typedef            char    ICHAR;             /* International character type */
typedef const      ICHAR    CONCH;            /* Constant char for string ptr */
```

If the libraries are ported to other processors, these allow the basic types to be redefined. The *ICHAR* and *CONCH* types are for future expansion to Unicode.

I do not include header files within other header files, as I like to look at the top of a source module to see what it depends on. However, headers should ideally comment their prerequisites. I almost never use C++ operator overloading, which I find confusing. Most classes have casting functions instead of casting operators and function names like *Compare ()* instead of operator overloading. In almost all cases, the data members of classes are *private* or *protected* to promote encapsulation, even if not so declared in this document.



Application Class

An application using OsRay++ must include one and only one application object, derived from the base class *APP*, defined in *OsRay.H*.

```
class APP : public THREAD
{
virtual void Main (void);
static void VAR Message (MSGID msg, ...);
};
```

There are two classes derived from this base:

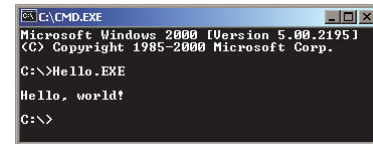
```
class APPCON : public APP // Console application
{
};

class APPWIN : public APP // Graphical user interface
{
    BOOL Dispatch (BOOL wait);
    void SetFocus (HWND focus, HACCEL accel);
};
```

In some cases, a program will declare a custom class derived from one of these, but it's not necessary:

```
static APPCON myapp; // Instance of console application

int main (int argc, char *argv[])
{
    myapp.Message (HELLO); // Print a message resource
    return 0;
};
```



Message () replaces *printf ()*, either printing to the console or displaying a dialog box depending on the derived *APP* class.

```
static APPWIN myapp; // Instance of windowed application

int WinMain (HANDLE inst, HANDLE prev, LPSTR cmdline, int show)
{
    myapp.Message (HELLO); // Display a message resource
    return 0;
};
```



The constructor for *APP* initializes the global pointer [*app*] so code throughout an application and the libraries can get access to the *APP* functions. *APPWIN*'s *Dispatch ()* provides the Windows message pump. *SetFocus ()* directs input to a particular window and specifies the key commands it recognizes.

Multi-Threading

The application object is the initial thread for a program. You can spawn additional threads by deriving classes from the *THREAD* base.

```
class THREAD
{
virtual void Main (void);
virtual void Terminate (void);
};

class APP : public THREAD
{
static void Initialize (THREAD *thd);
static void MultiThread (void);
};
```

Initialize () prepares the operating system for an additional thread, and then *MultiThread ()* begins execution of all the initialized threads with their *Main ()* procedures.

Strings

OsRay++ encapsulates text in *STRING* objects and *MESSAGE* resources. The *STRING* class simplifies dynamic text:

```
class STRING
{
    ICHAR *    data;                // Zero terminated character array
    BOOL       alloc;               // Indicates memory must be freed
    INTU       len;                 // Number of characters in data

    inline BOOL Exists      (void) const;
    inline INTU Length     (void) const;
    inline CONCH * String   (void) const;

    void Clear      (void);         // Additional assignment procedures
    BOOL Assign     (CONCH *str);   // exist, see the include files
    BOOL Assume     (CONCH *str);
    BOOL Set        (CONCH *src);
    void VAR Format  (MSGID fmt, ...);

    void LowerCase  (void);         // Additional manipulation and
    void UpperCase  (void);         // comparison functions also exist
    void Replace    (ICHAR src, ICHAR dst);
    BOOL CompareCS  (const STRING &targ) const;
    BOOL CompareCI  (const STRING &targ) const;

    void ReadIFF    (FILERD *in);   // Yes, there's more file stuff too
    void WriteIFF   (FILEWR *out, ULONG id);
};
```

Constant text is often compiled into a program. Instead of quoted strings in the source code, message resources should be placed in separate script files, named, and compiled by the resource compiler (ResComp.EXE). Although this takes longer, proper formatting (tabs, quotes and returns, etc) and international translation are both much easier. The resource compiler can also check for errors and compress or obfuscate the text.

```
Message:  HELLO
Hello, world!%0
```

STRING's Format () procedure replaces *sprintf ()* by generating a formatted *STRING* object from a message resource. In addition to fixing some of the bugs in *sprintf ()*, message resources have different field specifications:

```
Specification: %N[JT]
               N   Argument number (1-15)
               J   Optional justification & padding
               T   Field type

Justification: lN Left justified, field width N
               rN right justified, field width N
               jN Centered, field width N
               pX Pad with character X (default is space)

Field types:   s   Zero terminated ICHAR string (CONCH *)
               c   Single ICHAR character
               i   Ascii identifier value
               d   Signed decimal integer
               u   Unsigned decimal integer
               o   Unsigned octal integer
               h   Unsigned hexadecimal integer
               b   Unsigned binary integer
               m   Compiled message identifier
               fN Fixed point w/ N digits after decimal point

Terminator:    %0
Percent char:  %%
```

File names can be stored in *STRING* objects, but to manipulate them use the specialized *FNAME* class. It is similar to *STRING* (though not a derivative) but breaks the name into its component parts. It is compatible with Microsoft's long (LFN) and 8.3 file names, and will eventually handle Unix as well. Do not use fixed length char arrays since LFN requires these to be 260 bytes or larger.



```
class FNAME
{
    void      Clear      (void);
    void      Set        (CONCH *fname);
    void      Set        (CONCH *path, CONCH *file);
    void      SetPath     (CONCH *path);
    void      SetFile     (CONCH *file);
    void      SetExt      (CONCH *ext, BOOL replace);
    void      DosFix      (void);

    operator CONCH *      ()      const;
    inline  CONCH *      String  (void) const;
    inline  CONCH *      Path    (void) const;
    inline  CONCH *      File    (void) const;
    inline  CONCH *      Ext     (void) const;
    inline  CONCH *      Root    (void) const;
    inline  CONCH *      Base    (void) const;
    inline  BOOL         Exists  (void) const;
    inline  INTU         Length  (void) const;
};
```

Error Handling

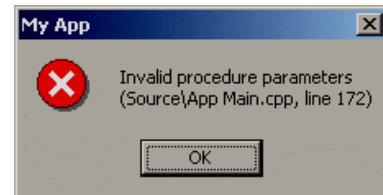
Each *THREAD* has its own error status variables, but the *APP* class contains the interface:

```
class APP
{
    static void      Clear      (MSGID caption);    // Clears any recorded errors

    static void      Beep      (STATUS severity);
    static void VAR   Debug    (MSGID msg, ...);    // There are additional recording
    static void VAR   Warning  (MSGID msg, ...);    // functions for system errors
    static void VAR   Error    (MSGID msg, ...);

    inline  BOOL      Check    (void) const;        // Error checking and reporting
    inline  BOOL      Abort    (void) const;
    static  BOOL      Report   (void);
};
```

Before starting an operation, such as when the user selects a menu command, use *Clear ()* to set the caption for any future errors. If an error occurs, record it using *Warning ()* or *Error ()*. They operate like *printf ()*, but store the output string for later display. *Debug ()* immediately prints to the debugger's trace log without recording the error. Use *Check ()* to continue processing unless an error has already occurred and *Abort ()* to detect an error before abandoning an operation. Then, after completing the operation, call *Report ()* to display all the error messages.



```
class APP
{
    static void      Attach    (REDACTOR *red);
    static void      Detach    (REDACTOR *red);
};

class REDACTOR
{
    virtual void      Redact    (CONCH *src, STRING& dst) = 0;
};
```

To provide the user additional information, error *REDACTOR* objects can edit messages. For example, a file object, described below, is a redactor. Just before reading a file, you can attach the file to the *APP*, then the redactor will append the file name to any error encountered until the file is detached.

Notice that this does not deal with recoverable errors, nor does it use C++'s exception handling, which I find cumbersome.

Math

OsRay++ has some basic math functions, including vector and matrix math for 3D graphics. *FLT3D* and *TMP3D* are currently defined as float and double, respectively.

```
class VECTOR
{
    FLT3D    x,y,z;                                // W value assumed to be 1

    inline   FLT3D    Axis      (INTU index)      const;
            TMP3D    Length    (void)            const;
            BOOL     Compare   (const VECTOR& targ) const;

    void     Clear      (void);
    void     Normalize  (void);
    void     Add         (const VECTOR& src1, const VECTOR& src2);
    void     Subtract    (const VECTOR& src1, const VECTOR& src2);
    void     Multiply    (const VECTOR& src,   const TMP3D scalar);
    TMP3D    Dot         (const VECTOR& src2) const;
    void     Cross       (const VECTOR& src1, const VECTOR& src2);
    void     Transform   (const VECTOR& src,   const MATRIX& mtx );
};
```

Matrices are assumed to be affine, i.e. the last column is 0,0,0,1, and the memory format is the same as in OpenGL.

```
class MATRIX
{
    TMP3D    val [4][4];                            // In column major order

    void     Clear      (void);
    void     Rotate     (FIX32 x, FIX32 y, FIX32 z);
    void     Translate   (FLT3D x, FLT3D y, FLT3D z);
    void     Scale       (FLT3D x, FLT3D y, FLT3D z);

    BOOL     Transpose   (const MATRIX& src);
    BOOL     Invert      (const MATRIX& src);
    void     Multiply3   (const MATRIX& src1, const MATRIX& src2);
    void     Multiply4   (const MATRIX& src1, const MATRIX& src2);
};
```

The vector and matrix operations are implemented in optimized assembly language (without MMX). I store angles in 16.16 bit fixed point, so *Rotate ()* uses these to rotate a matrix. Quaternions are also included:

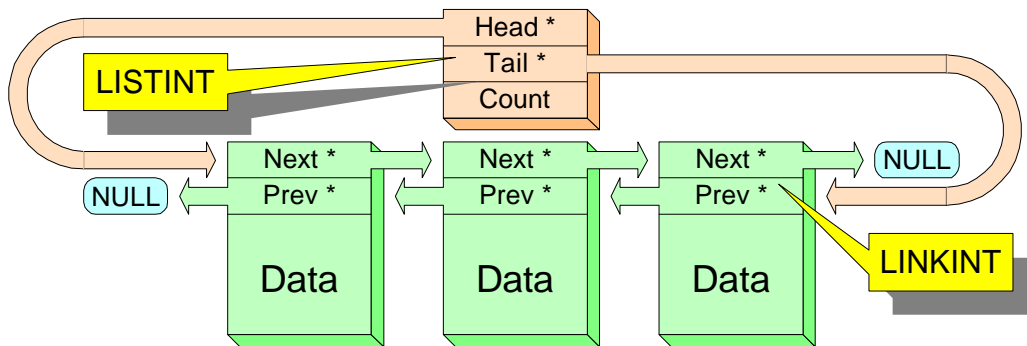
```
class QUAT
{
    FLT3D    x,y,z,w;                                // Real scalar and imaginary vector

    void     Clear      (void);
    void     FromAxis    (const VECTOR *axis, FIX32 angle);
    void     FromMatrix  (const MATRIX *mtx);
    FIX32    ToAxis      (VECTOR *axis)   const;
    void     ToMatrix    (MATRIX *mtx)    const;

    void     Multiply    (const QUAT *src1, const QUAT *src2);
    void     Add         (const QUAT *src1, const QUAT *src2);
    void     Subtract    (const QUAT *src1, const QUAT *src2);
    void     Normalize   (void);
};
```

Linked Lists

I use doubly linked lists extensively, but usually intrusive lists. To store an object in the intrusive *LISTINT* <> (a template), its class must be derived from a *LINKINT* <>. This means that the link pointers are a part of the object, not allocated separately. It is both more efficient and simpler than Microsoft's *CList*, but an object can only be a member of one *LISTINT*.



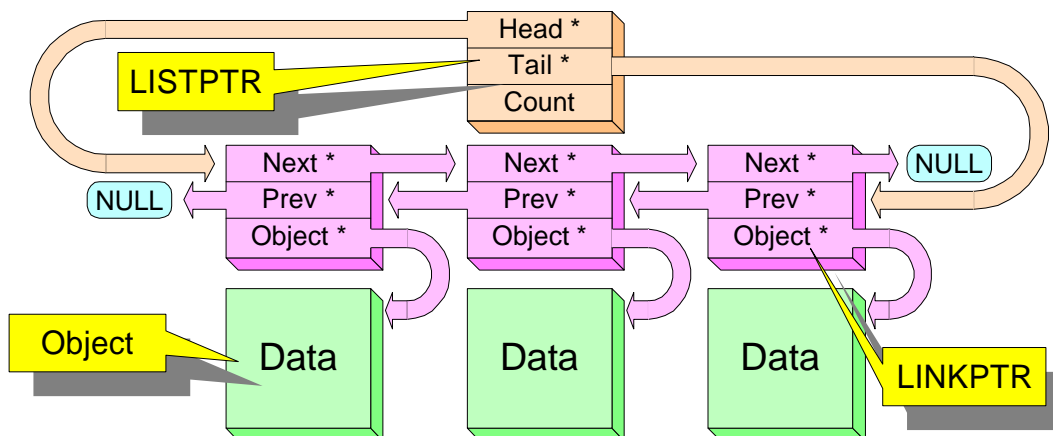
The linked list interface is below. Note that indexes start at 1. Attempting to get the index of a node not in a list returns 0.

```
template <class TYPE> class LINKINT
{
public:
    TYPE *    Prev      (void)      const;    /* Return the previous entry */
    TYPE *    Next      (void)      const;    /* Return the next entry */
    INTU      Index     (void)      const;    /* The number of prior entries */
};

template <class TYPE> class LISTINT
{
public:
    INTU      Count      (void)      const;    /* Return number of entries */
    TYPE *    Head       (void)      const;    /* Return the first entry */
    TYPE *    Tail       (void)      const;    /* Return the last entry */
    TYPE *    Entry      (INTU index) const;    /* Return the nummbered entry */

    void      Append     (TYPE *);           /* Add an entry to the end */
    void      Insert     (TYPE *prev, TYPE *); /* Insert entry in the middle */
    void      Remove     (TYPE *);           /* Remove a specific entry */
    TYPE *    Extract    (void);             /* Remove & return first entry */
};
```

Some objects need to be in multiple lists, calling for non-intrusive linked lists. This is a linked list of pointers to objects, similar to Microsoft's *CList*.



Before adding an object to this type of list, you need a *LINKPTR* instance. It can be separately allocated or a member of an object. If you know how many lists the object can be a member of, you can put the *LINKPTR*'s in the original object itself. The interface is the same as for the intrusive list, except each *LINKPTR* must be assigned to its referenced object and all the list changes are done with *LINKPTR*'s instead of their referenced objects:


```

template <class TYPE> class LINKPTR
{
public:    void      Init          (TYPE *object);          /* Assign to referenced object */
};

template <class TYPE> class LISTPTR
{
public:    void      Append        (LINKPTR <TYPE> *);
          void      Insert        (LINKPTR <TYPE> *prev, LINKPTR <TYPE> *);
          void      Remove        (LINKPTR <TYPE> *);
};

```

File Access

File access is encapsulated in the following disk classes, which process sequential files effectively, but do not simultaneously read and write to the same file. Word and long operations write according to the endian flag: big or small.

```

class FILERD : public FILEBASE
{
    BOOL      Open          (CONCH *fname);
    void      Close         (void);
    void      SetFlags      (BOOL endian, BOOL align);

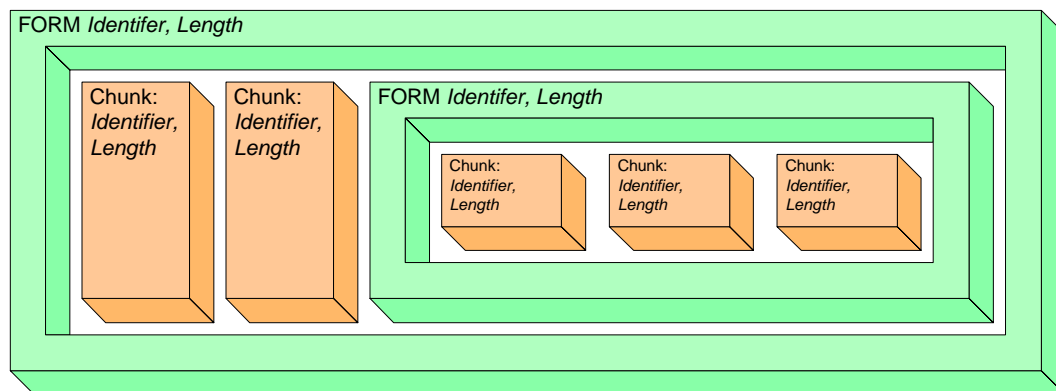
    UBYTE     ReadByte      (void);
    UWORD     ReadWord      (void);
    ULONG     ReadLong      (void);
    FLT32     ReadFLT32     (void);
    void      ReadData      (void *dst, INTU length);
};

class FILEWR : public FILEBASE
{
    BOOL      Open          (CONCH *fname);
    void      Close         (void);
    void      SetFlags      (BOOL endian, BOOL align);

    void      WriteByte     (UBYTE value);
    void      WriteWord     (UWORD value);
    void      WriteLong     (ULONG value);
    void      WriteFLT32    (FLT32 value);
    void      WriteData     (void *dst, INTU length);
    void VAR  Message       (MSGID msg, ...);
};

```

Many file formats are chunky, meaning they contain a series of chunk identifiers with lengths. An unknown or unwanted chunk can be skipped by advancing past it's length. The *FILERD* and *FILEWR* classes handle most of these formats, including IFF's such as Dpaint, Lightwave and Maya, RIFF's such as Windows WAV's and AVI's, and 3D Studio's 3DS.



The chunk parser *ReadChunks* () takes an array of *CHNKDEF*s (identifiers with procedure pointers) and calls the appropriate reader for each chunk that it encounters, skipping over unknowns. The chunk writing functions track the starting and ending points and fill in chunk lengths. *SetChunks* () specifies the byte sizes for a chunk's identifier and length values, and, along with *SetFlags* (), enables all the different file formats.



```

struct CHNKDEF /* Chunk definition entry */
{
    ULONG      id; /* 4 character identifier */
    ULONG      subid; /* 4 character identifier */
    void      (*Reader) (FILERD *); /* Called when chunk found */
};

class FILERD
{
    void      SetChunks (INTU id, INTU len, BOOL hdr);
    void      ReadChunks (void *owner, CHNKDEF *list);
};

class FILEWR
{
    void      SetChunks (INTU id, INTU len, BOOL hdr);
    ULONG      StartChunk (ULONG id, ULONG subid);
    void      EndChunk (ULONG pos);
};

```

Chunky files created by Animax and 3D editors are often much more complex than LBM's or WAV's. IFF forms are often contained within other forms, and later forms can have chunks that reference the previous ones. To support this, pointers to objects stored in chunks are cached in an ID reference indexing system available in both *FILERD* and *FILEWR* classes. These functions convert pointers to chunk indexes and vice versa while both reading and writing.

```

class CORRELATED
{
    void *      Current (ULONG id);
    void *      Entry (ULONG id, INTU index);
    INTU      Index (ULONG id, CVOID *obj);

    INTS      Correlate (ULONG id, void *obj);
    void      Select (ULONG id, void *obj);
    void      Select (ULONG id, INTU index);
};

```

FILERD also handles some unusual chunk types:

- A flag chunk is a zero length chunk that indicates *TRUE* if it exists, *FALSE* if it does not. Though it takes up as many as 8 bytes, it is more portable and durable than a bit field entry in a structure.
- A *NAME* chunk is often included within a sub-FORM (a FORM contained inside another FORM). It indicates the object's filename if it were saved by itself.
- *FORM XTRN* is a reference to another file. It contains a *NAME* chunk and an optional *PATH* chunk, and means 'read this file as if it appeared within the current one'.

VISUAL-C LIBRARY

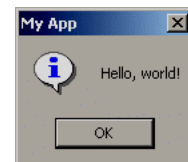
The standard C libraries include startup code that executes first and then calls *main* (). For many programs, some or all of this code is unnecessary. The Visual-C library provides a minimal replacement allowing OsRay++ programs to be compiled and linked without any of the standard libraries.

```

static APPWIN myapp; // Instance of windowed application

void APP::Main (void)
{
    Message (HELLO); // Display a message resource
};

```



The startup code calls static object constructors and destructors, so declaring an application specifies the initial thread of execution. If you need command line information like *argc* & *argv*, see the *CMDLINE* class in File.H of the OsRay core library.

The Visual-C library also provides global *new* and *delete* operators that currently call the operating system's heap routines. Until I replace them with a local allocator, these are considerably slower than the standard library versions.

PIXEL LIBRARY

The OsRay core library includes the graphics sources, while the Pixel library contains drawing primitives for simple 2D drawing. The primitives do not provide the more complex painting functions of a painting program or a 3D renderer but can cooperate with OpenGL.

```
typedef    ULONG    COLOR;                                /* Drawing color 8, 16 or 32 bits */

struct RGBA                                                /* Color definition with transparency */
{
    UBYTE    red;
    UBYTE    grn;
    UBYTE    blu;
    UBYTE    alf;
};

struct BOX                                                /* Rectangle, position and dimensions */
{
    INTS     x, y;
    INTU     w, h;
};
```

The *PIXMAP* class defines bitmap images of various color depths. They can be dynamically allocated to any dimensions, loaded from IFF files or compiled into a program with Res-Comp.EXE.

```
class PIXMAP
{
    UBYTE    lgres;                                        /* Log (2) of bits / pixel per plane */
    INTS     dx;                                          /* Blit handle offset from top,left */
    INTS     dy;
    INTU     width;                                       /* Image dimensions in pixels */
    INTU     hght;
    UBYTE *   pixels;                                    /* Pointer to pixel data by rows */
    INTU     dsplst;                                     /* OpenGL display list identifier */

    void      SetRes      (UBYTE lgres);
    BOOL      Alloc      (INTU width, INTU hght);
};
```

The *PIXTARG* class, in the Pixel library, represents the destination for drawing operations using the above source types. It is an abstract base, which currently has two derived classes: *PIXSOFT* and *PIXOGL*.

```
class PIXTARG
{
    void      SetOrigin   (INTS x, y);
    void      SetClip     (BOX *box);

    void      SetPixel    (INTS x, y, COLOR color);
    COLOR     ReadPixel   (INTS x, y);
    void      HorzLine    (INTS x, y, INTU w, COLOR color);
    void      VertLine    (INTS x, y, INTU h, COLOR color);
    void      Bevel       (BOX *box, INTS w, COLOR *shades);
    void      FillBox     (BOX *box, COLOR color);
    void      Blit        (PIXMAP *bmp, INTS x, y, PIXCNVT *cvt, POINT2 *mag);

    COLOR     Convert     (PIXCNVT *cvt, INTU index);
    PIXCNVT * Convert     (RGBA *pal, INTU ncolors, UBYTE *indexes, UBYTE lgres);
};
```

The *PIXCNVT* object is the key to the primitives. It is a pre-configured translation table that the destination's *Blit* () procedure uses to convert from a particular source pixel format (*lgres*). For example, it can draw 8 bit indexed into 16 bit RGB. It can also remap indexed colors as it draws, so the source and destination palettes do not have to match. *Convert* () allocates and builds these according to the source and destination pixel types. The single color *Convert* () function changes an *RGBA* color definition into a *COLOR* value through a *PIXCNVT*.

The *PIXOGL* graphics destination encapsulates calls to OpenGL, using the *dsplst* member of a *PIXMAP* to load images into video memory. It allows OpenGL 3D software to use the same 2D and window management code as other OsRay++ programs.

The *PIXSOFT* graphics destination is a 2D software renderer, primarily written in assembly language for speed. The destination pixel format is specified by a *PIXRES* structure, which provides custom assembly language called by the primitives in the *PIXSOFT*. It currently supports resolutions of 8, 16 and 32 bits per pixel. While *PIXSOFT* can draw into a DirectDraw surface, a DirectX hardware version should be added.

```
class TILESET
{
    PIXMAP    image;                // A single, relocatable tile
    PIXMAP    tiles;               // Large image containing all tiles

    INTU      nhorz;               // Number of tiles in the container
    INTU      nvert;
    INTU      tilew;               // Single tile dimensions
    INTU      tileh;

    BOOL      Alloc      (INTU wdt, INTU hgt, UBYTE lgres);
    BOOL      Allot      (INTU wdt, INTU hgt, INTU ntiles);
    PIXMAP *   Location   (INTU entry);
};
```

The *PIXTARG* class also contains a blitter for tiled images where each pixel in the source *PIXMAP* is an index into a *TILESET*. A *TILESET* is a single bitmap divided into tiles; it's *Alloc ()* routine specifies the dimensions of the container, while *Allot ()* specifies the dimensions of the tiles. *Location ()* returns a *PIXMAP* describing an individual tile within the container. Res-Comp.EXE can also create tiled *PIXMAP*s and *TILESET*s from Animax files.

RESOURCE COMPILER

Res-Comp.EXE converts both text scripts and IFF binary files into data for OsRay++ based programs. However, it generates structures to be linked in as standard data instead of resources that need to be explicitly loaded.

```
Message:  HELLO
Hello, world!%0
```

Compiling an input file produces two output files: an assembly language data file (.ASM) and a C++ header file (.MH). Since most text input to Res-Comp consists of message resources, I use the extension .MSG. The sample above produces the following header:

```
extern struct MESSAGE  HELLO [];
```

To use a resource in your code, simply put the source file in your make script and #include its .MH header before calling an OsRay++ function with a pointer to the structure. There are no integer identifiers to hassle with.

IFF resources:

- Color palettes
- Bitmap images
- Image folders
- Tile catalogs
- Tiled images
- Film animations
- Glazier windows
- Glazier dialogs

Text resources:

- Message strings
- Error maps
- Lexer patterns
- Parser tokens
- Parser grammars
- Native procedure declarations
- Scripting language code

Interchange format is an older standard used by Dpaint and Amiga programs. Animax uses IFF exclusively, but Photoshop version 5, Debabelizer and other programs also read & write IFF images.

```
C:\CMD.EXE
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\>Res-Comp.EXE

Resource Compiler. Copyright (c) 2001 by Raymond E. Tobey
Converts resources into data for use with OsRay++

USAGE: ResComp [-options] <infile> [outfile]
-i IFF image input
-s Text script input
-t Borland Turbo Assembler output
-n Netwide Assembler output

-d Debug output (faster compression)
-r Release output (better compression)
-p Predictive Lempel-Ziv image compression
-e Exhaustive Lempel-Ziv image compression
-h Haar wavelet image compression
-w Daubechies wavelet image compression

-v Verbose log output

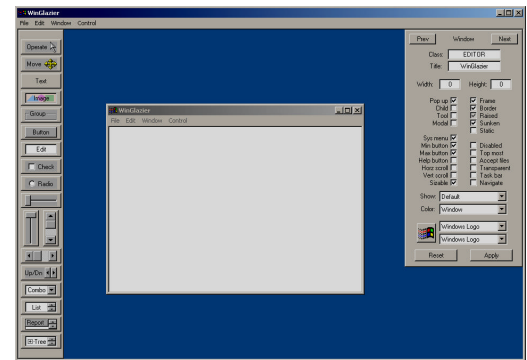
An optional output filename may follow the input filename
C:\>
```

GLAZIER EDITOR

The dictionary defines 'glazier' as a person who cuts and fits glass to windows. Glazier.EXE is a window and dialog editor designed to create resource data for use with OsRay++.

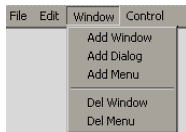
Using Glazier, you can create a window or dialog, set its display styles, and then add menus, control gadgets and keystroke assignments. The saved file is compiled by the resource compiler and linked into a program as an external structure.

While its purpose is the same as the dialog editor in Microsoft's Visual Studio, it produces simpler and more complete output. Since it generates linked data structures instead of .RC resources, integer identifiers are not necessary.



Windows

Use the window menu to create a blank window or dialog. This will put a panel of parameters on the right side of Glazier describing the new window. Use the check boxes to set the display style, which match the normal and extended flag bits for Microsoft's *CreateWindowEx* (). The 'Apply' button will display the window using the changed settings, while the 'Reset' button restores the settings to match the displayed window.



Combinations of these style settings produce varied results, for example, 'sys menu', 'frame' and 'border' must all be checked to get a close box at the right of a title bar. Since Glazier displays its window using the same code that an application would, you can experiment to find the right settings.



In OsRay++, dialogs are very similar to other windows except that they can become modal, meaning that the operating system restricts input to other windows while displaying a modal dialog. Microsoft's API uses different data structures and dispatches messages for dialogs differently than for other windows, so you must choose a window's type on creation.

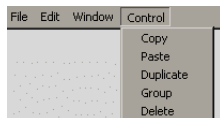
Controls

The palette of tools on Glazier's left side enables you to 'draw' control gadgets on a window or dialog. By clicking and dragging, you can make a rectangle on the window that Glazier will fill with a chosen control. Selecting one of these controls changes the parameter panel on the right to one specific to that kind of control.

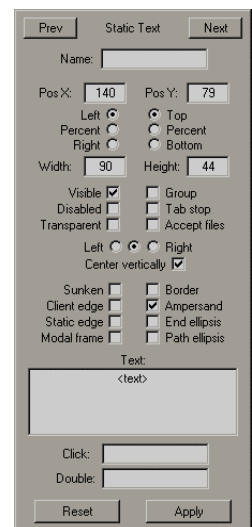
Many parameters are common to all controls. Each can be assigned a name for interfacing with its parent application. Position and dimensions are specified in pixels, not dialog units. The 'left' & 'right' options specify which side of a control's parent window its x position is relative to. The 'percent' option changes the position to be a percentage of the parent window's width.

Most controls have optional event handler parameters. For these, enter the name of a function to call when the event occurs. This function should be a member of the window's class unless you specify a scope as part of the name.

The default editing mode is . It allows you to select and interact with the controls on your subject window. While the primary selected control is outlined in red, you can select secondary controls that are outlined in orange. The  editing mode lets you move a control or group of selected controls with the mouse.

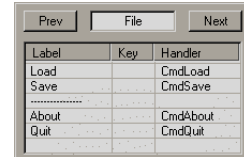


The control menu lets you copy a selected control or group of controls from one window and paste them into another window. The duplicate command makes copies of controls including their parameters. Automatic radio buttons use their group style bit to determine which other radios to deselect when they are pressed, so the group command puts all the selected controls together in for this purpose.

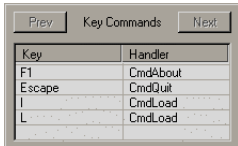


Commands

You can add menus to a window or dialog, but be sure to set the window's style so menus can be seen. You must also assign a name to the menu at the top of its parameter panel before you can see it. To edit the first menu, select the menu command from Glazier's edit menu. The prev and next buttons switch to other menus. If a menu's parameter panel is showing when you add a menu, the new one goes to the left of the current one. Otherwise the new one goes to the right of all the menus.



Label	Key	Handler
Load		CmdLoad
Save		CmdSave
.....		
About		CmdAbout
Quit		CmdQuit



Key	Handler
F1	CmdAbout
Escape	CmdQuit
I	CmdLoad
L	CmdLoad

Use the parameter panel to add entries to menus. Each entry should have a text label and the name of that command's handler function. Currently, keyboard equivalents must be entered into the keystroke list separately. Cascading menus are also not yet implemented.

The keys entry in Glazier's edit menu lets you assign handlers to specific keystrokes. Just like control and menu handlers, key handlers are member functions of the window's class unless a scope qualifier is part of the handler name.

GLAZE LIBRARY

Glaze is the companion library for the Glazier editor, providing an object-oriented interface to the Win32 API. Often you will not need to include "windows.h" or any of Microsoft's headers when using Glaze, but it is compatible with them.

Windows

To open a window, derive a class from *WINBASE* and create a corresponding window using Glazier. In the window's parameter panel, make sure the class name matches yours.

```
class WINBASE : public WINHDL
{
    BOOL      Open      (WINDOW *def, HWND parent = NULL);
    void      Close     (void);
    void      Show      (void);
    void      Hide      (void);
    void      Invalidate (BOOL erase = TRUE);

    SLONG     OnCreate   (INTU wParam, SLONG lParam); // Sample message handlers
    SLONG     OnDestroy (INTU wParam, SLONG lParam);
};
```

Compiling the window with Res-Comp.EXE will generate an external *WINDOW* structure that you can pass to *WINBASE*'s *Open* (). This structure's name will be the saved window's filename prepended with *win_*.

You will also need a message map. This is an array of *WMLIST* structs, each of which contains a Windows message identifier and a procedure pointer to its handler. Glazier should eventually generate these, but currently you must supply one. It must have the same name as the compiled window, but prepended with *msg_*. Most of the required message handlers are members of *WINBASE* and its derivatives.

```
const WMLIST msg_display [] = {
    { WM_CREATE, (WMPROC) WINHDL::OnCreate },
    { WM_CREATE, (WMPROC) WINBASE::OnCreate },
    { WM_CREATE, (WMPROC) DISPLAY::OnCreate },
    { WM_SIZE, (WMPROC) WINBASE::OnSize },
    { WM_SETFOCUS, (WMPROC) WINBASE::OnSetFocus },
    { WM_KILLFOCUS, (WMPROC) WINBASE::OnKillFocus },
    { WM_COMMAND, (WMPROC) WINHDL::OnCommand },
    { WM_NOTIFY, (WMPROC) WINHDL::OnNotify },
    { WM_CLOSE, (WMPROC) DISPLAY::OnClose },
    { WM_DESTROY, (WMPROC) WINBASE::OnDestroy },
    { WM_DESTROY, (WMPROC) WINHDL::OnDestroy },
    { 0, NULL }
};
```

For a dialog, derive your class from *WINDLG* instead of *WINBASE*. The compiled *DIALOG* structure will be prepended with *dlg_* instead of *win_*. Note the differences in the message map, such as *WM_INITDIALOG* instead of *WM_CREATE*.

```
class WINDLG : public WINHDL
{
    void      Open      (const DIALOG *def, HWND parent = NULL);
    void      Modal     (const DIALOG *def, HWND parent = NULL);
    void      Close     (void);

    void      OnCreate   (INTU wParam, SLONG lParam); // Sample message handlers
    void      OnDestroy (INTU wParam, SLONG lParam);
};
```

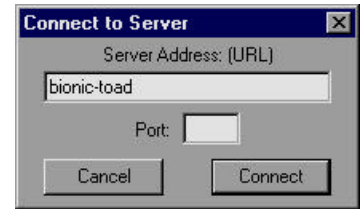
Controls

Win Dlg.H includes small classes for many of the control gadgets. While they are most commonly used with dialogs, they work just as well on windows.

During a dialog's *WM_INITDIALOG* handler or a window's *WM_CREATE* handler, execute the *INIT_* macro the resource compiler generates. It will attach member objects defined in your derived class to the UI controls according to the names specified in each control's parameter panel in Glazier.

```
class CONNDLG : public WINDLG
{
    CTLEDIT    addrbox;    // Control gadget interfaces
    CTLEDIT    portbox;

    void        Connect    (void);
    void        OnCreate    (INTU wParam, SLONG lParam);
};
```



The *WM_COMMAND* and *WM_NOTIFY* handlers defined in *WINHDL* will dispatch events to the command handlers specified in Glazier for menus, keystrokes and controls. In the command handlers you supply, use the *Get ()* and *Set ()* functions for the various control classes to read and write the displayed gadgets.

```
class CTLBASE // Base class for various control interfaces
{
    BOOL        Open    (const CONTROL *def, const WINHDL *parent);
    void        Assign    (const WINHDL *parent, INTU id);

    void        Clear    (void);
    BOOL        GetText    (STRING *str);
    void        SetText    (CONCH *str);
    void VAR    FmtText    (MSGID msg, ...);
};
```

Most controls have text captions, which are displayed in different ways. As the base for the control classes, *CTLBASE* provides access to the caption. *FmtText ()* generates a formatted string from a message resource, a la *sprintf ()*, and sends it to a gadget. *GetText ()* retrieves a string from a gadget. Each of the derived control classes has additional functions specific to a particular type of Windows gadget.

Graphics Windows

Glaze includes several window classes derived from *WINBASE* providing graphic displays. The *WINDIB* class uses a Win32 *DIBSection* as display memory, and a *PIXSOFT* that draws into it.

```
class WINDIB : public WINBASE
{
    PIXSOFT    pixdst;

    virtual BOOL    Alloc    (INTU width, INTU hght);
    virtual void    SetColors    (const RGBA *colors, INTU ncolors);
    virtual void    GetColors    (const RGBA *colors, INTU ncolors, UBYTE *index);
    virtual void    Paint    (BOX *box = NULL, BOOL async = FALSE);
    virtual void *    Lock    (BOX *box);
    virtual void    Unlock    (void *surface);

    SLONG        OnPaint    (INTU wParam, SLONG lParam);
    SLONG        OnErase    (INTU wParam, SLONG lParam);
    SLONG        OnPalChange    (INTU wParam, SLONG lParam);
    SLONG        OnQueryPal    (INTU wParam, SLONG lParam);
};
```

The *OnEraseBkgnd ()* and *OnPaint ()* message handlers (which must go into your message map) use the GDI to paint the image on the screen. Doing this on *WM_ERASEBKGN*D eliminates the flicker so many Windows programs exhibit when responding to *WM_PAINT* messages. The *SetColors ()* function sets the color palette for the window, while *GetColors ()* retrieves color indexes for an array of *RGBA* color definitions. These functions allow the window to use an identity palette while the *PIXSOFT*'s *Convert ()* function handles color translation.

Class *WINDX* is also a window with a *PIXDEST*, implemented using *DirectDraw* as the display memory. It shares the interface of *WINDIB* making these classes interchangeable. The relevant procedures are included in *WINBASE* as virtuals so an application need not know which API it is using.



Class *WINOGL* provides a window for a double-buffered OpenGL graphics display. Its *WM_PAINT* and *WM_ERASEBKGDND* handlers *Render ()* to draw the display. Override it to display your own.

```
class WINOGL : public WINBASE
{
    OPENGL *   opengl;

virtual    BOOL        Render        (const BOX *area);

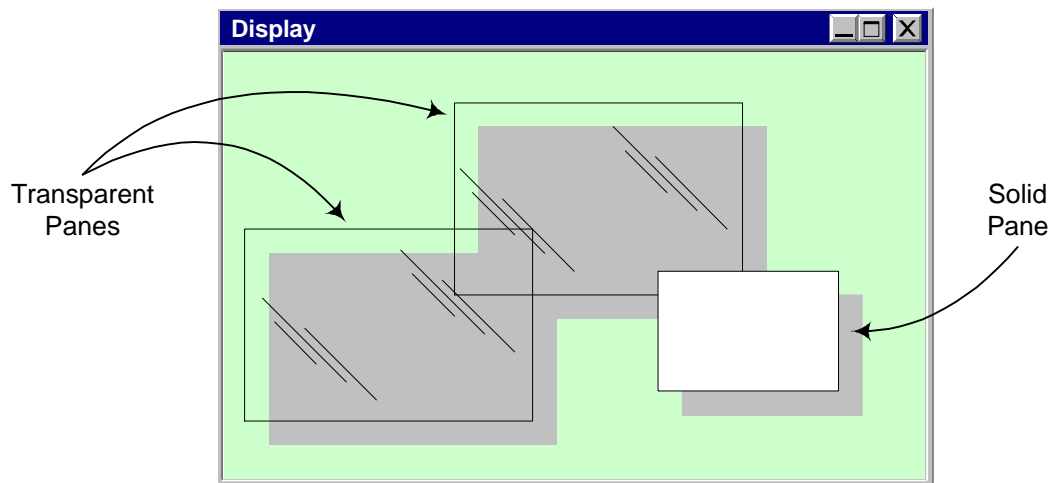
    SLONG        OnCreate        (INTU wParam, SLONG lParam);
    SLONG        OnSize         (INTU wParam, SLONG lParam);
    SLONG        OnErase        (INTU wParam, SLONG lParam);
    SLONG        OnPaint        (INTU wParam, SLONG lParam);
    SLONG        OnDestroy     (INTU wParam, SLONG lParam);
};
```

While *WINOGL* has the same virtuals as the others, its display memory is not accessible. It is partially interchangeable with *WINDIB* and *WINDX*. All of these window classes can implement the pane manager using their virtual procedures.

Putting a *PIXOGL* member into a *WINOGL* derived class results in an OpenGL window with a hardware accelerated *PIXTARG* interface, while putting an *OPENGL* object into a *WINDIB* derived class results in a *PIXTARG* with a software OpenGL interface.

PANE LIBRARY

A pane is a transparent sheet on the display, providing a surface for a graphical user interface object. It performs the same function as a Windows *HWND*, except that Windows handles transparency and color palettes particularly poorly, while the lack of double buffering causes flickering and tearing. Since key features of Animax are animation and transparency, it uses *PANE*'s instead of *HWND*'s in most cases. Panes also happen to be independent of the operating system, much simpler and potentially faster.



The *PANE* class is a base for customized graphical objects. The virtual function *Draw ()* should be overridden to paint the pane, just like a *WM_PAINT* message does under Windows. However, it is important that a pane retain enough state information to do this very quickly. Also, a pane should never paint itself at any other time. It should change its internal state variables and invalidate a region. Override *Mouse ()* to respond to mouse clicks, *Feedback ()* to change the cursor, etc.

```
class PANE : public LINKINT <PANE>
{
virtual    void        Draw        (GFXDEST *dst);
virtual    void        Mouse       (INTS x, INTS y, INTU state);
virtual    void        Feedback    (GFXDEST *dst, INTS x, INTS y);
};
```


A *SILL* is a container for *PANE*'s, displaying a linked list of panes layered with the first in the list on top. Adding a pane inserts it into the list according to its depth. The *PANE*'s box defines its position relative to its parent *SILL*. Coordinates within the *PANE* are relative to its box.

```
class SILL
{
LISTINT  <PANE>      panes;

void      Show      (PANE *pane);
void      Hide      (PANE *pane);
void      Surface    (PANE *pane);
void      Refresh    (void);
void      Refresh    (const BOX *box);
};
```

Inside the *SILL* class is a dirty rectangle system for refreshing the display. When a pane or a part of a pane changes, it calls a *Refresh ()* function to record the changed display area. At the end of every event, the dirty boxes are merged and those areas redrawn.

CODEC LIBRARY

The *CODEC* class is the base class for several interchangeable compressor / decompressor implementations. A compressor program can create an instance of any of these compressors, iterate over the source images as shown in the example below, then output compressed images.

```
class CODEC
{
virtual  BOOL      Init      (void);
virtual  BOOL      Configure (INTU pass);
virtual  void      Configure (BITRD *in);
virtual  void      Configure (BITWR *out);

virtual  void *     Prepare   (PIXMAP *targ);
virtual  BOOL      Process   (PIXMAP *targ, PIXMAP *prev, void *prep);
virtual  void      Build     (INTU pass, INTU maxbits);

virtual  void      Read      (void);
virtual  void      Write     (void);
virtual  void      Log       (FILEWR *out, ULONG fsize);
virtual  void      Verbose   (FILEWR *out, ULONG fsize);
};
```

While the Codec library provides many different compressors that can also decompress, a few are paired with high speed decompressors in the OsRay core:

- *CDQPLZ* generates predictive lempel-ziv for *PAKPLZ*. This is a lossless compressor for paletted images. It outperforms GIF by a fair margin.
- *CDQHAAR* uses the Haar wavelet transform and meshes with *PAKHAAR*. This is a lossy compressor designed for video, where speed is more important than image quality.
- *CDQDAUB* uses a Daubechies wavelet transform that *PAKDAUB* reads. This is a lossy compressor with quality similar to JPEG, but it can achieve much higher ratios. Since it is slow, it is best for images that can be decompressed in advance of being displayed, such as texture maps.

Some compressors generate a preparation data block. In the case of the predictive lempel-ziv codec, the prep data is a predictor table describing which pixel values follow each possible color by probability. Your compressor program should use a separate prep block for each color palette.

The Haar and Daubechies compressors operate on continuous data, such as gray scale or true color. RGB images are converted to YUV before compression, and quality is higher for intensity than chroma channels.

```
INTU  pass = 0;

while (app->Check() && cdc->Configure(pass))
{
    for each image...
    {
        cdc->Process (image, NULL, prep);
    }

    cdc->Build (pass++, 16);
}
```

The *PAKPIX* class is the base for the fast decompressors included in the OsRay core:

```
class PAKPIX
{
    void      Configure      (PIXMAP& dst);
    void      Configure      (PIXSOFT& dst);
    void      Configure      (void *pixels, INTU w, INTU h, INTU pitch, INTU lgres);

    virtual   void      Prepare      (PAKBIT& in);
    virtual   void      Decompress   (PAKBIT& in);
};
```

Use one of the *Configure* () procedures to specify the destination for the decompressed pixels. The *PAKBIT* class specifies the compressed source pointer. It allows the compressed data to be stored in a bit stream regardless of byte & word boundaries.

SLANG LIBRARY

Slang is a scripting language with a very simplified C-like syntax, designed for animation. It executes in a stack based virtual machine, but can call C and C++ as well as its own subroutines. Slang scripts can be attached to frames and sequences in Animax, and Res-Comp can also compile scripts for testing.

```
class AMEXEC
{
    BOOL      SetScript      (AMCODE *scr);
    void      SetNative      (const AMPROC *list);
    MSGID     Execute        (void *cip);
};
```

The OsRay core contains the abstract machine executive (*AMEXEC*), which executes compiled byte code (*AMCODE*). The Slang library is the compiler that produces this byte code. It currently recognizes the following grammar:

- Native procedure declarations
- Scripted procedure declarations
- Code blocks { ... }
- Comments: /* ... */ and // ...
- If (*expression*)
- Else
- While (*expression*)
- Assignments: cell = *expression*
- Math: +, -, *, /
- Procedure calls

```
NATIVE Print (value);

CountDown (value)
{
    while (value)
    {
        Print (value);
        value = value - 1;
    }
}
```

There is only one variable type, the CELL, a 32 bit signed integer. Procedures can receive cells as parameters, declare them as local variables, and return them as results.

In order to call native C and C++ procedures, Slang requires native procedure declarations: the keyword *NATIVE* followed by the prototype. These prototypes must be in Slang form, where parameters have no type keywords since they are all cells. As separate resources, Res-Comp can generate native procedure lists, *AMPROC* structures. To link compiled Slang to its C++ parent, *SetScript* () matches the scripted procedure calls to the natives most recently assigned by *SetNative* ().

```
Natives: TEST
public: void Print   (int value)
public: void Display (int value)
%0
```

STORAGE LIBRARY

Database systems are designed to store records reliably and access that data quickly. However, some applications do not require the complexity of commercial database systems. The storage library records blocks of data indexed by 32-bit keys. The library features:

- A single block of binary data can be stored under each index key.
- Index keys cannot be duplicated.
- Atomic write operations insure that a corrupted database is never written.
- Transactions insure that if any records are modified, all of them are.
- B+Tree hierarchy optimizes the number of sectors accessed per record.
- Multiple processes can read simultaneously, but only one can write.
- Solid state drives eliminate lengthy seek delays.

To prevent confusion with relational databases, I refer to these databases as 'data stores'. The library supports two types of data stores.

```
class DATAFILE : public STORE
{
    void      Configure      (INTU size);
    BOOL      Create         (CONCH *fname, INTU count);
    BOOL      Open           (CONCH *fname);
    void      Close          (void);
};
```

The *DATAFILE* class represents a data store contained in a single file. Its *Configure ()* procedure selects a sector size before a data file is created. The default is 256 bytes. The *[count]* parameter to *Create ()* specifies the number of such sectors to allocate for a file.

```
class DATADISK : public STORE
{
    BOOL      Create         (INTU disk);
    BOOL      Open           (INTU disk);
    void      Close          (void);
};
```

The *DATADISK* class represents a data store that fills an entire drive. Both types use all of the space allotted to them at creation. They cannot grow dynamically, so if one becomes full, it must be copied into a larger storage container.

```
class RECORD : public STATUS
{
public:    RECORD      (STORE& sto);

        BOOL      Head      (void);                // Record selection
        BOOL      Prev      (void);
        BOOL      Next      (void);
        BOOL      Tail      (void);
        BOOL      Locate    (INTU index);
        BOOL      Search    (ULONG key);

    inline  BOOL      Valid    (void) const;          // Data block access
    inline  ULONG     Key      (void) const;
    inline  INTU      Length   (void);
        BOOL      Read      (void *buffer, INTU length);
};
```

The *RECORD* class allows read access to individual entries within a data store. The *Locate ()* procedure selects a particular entry according to its numerical position, starting with 1. The *Search ()* procedure finds the entry with a specific 32-bit index key. *Prev ()* and *Next ()* step through the existing records sequentially from the current position, while *Head ()* and *Tail ()* select the first and last entries in the data store.

Once an entry has been selected with these operations, the record and its data block can be read with the access functions. *Valid ()* indicates whether a record is currently selected.



The *TRANSACT* class allows writing to existing records as well as creating new ones:

```
class TRANSACT : public RECORD
{
public:    TRANSACT    (STORE& sto);

        BOOL        Insert        (ULONG key);
        BOOL        Delete        (void);

        BOOL        Write        (const void *data, INTU length);
        BOOL        Commit        (void);
};
```

Since *TRANSACT* is based on the *RECORD* class, any of its inherited procedures can be used to select existing records to modify. The *Insert ()* procedure will create a new entry in the data store if the specified index key is unique. When *Write ()* is called on a particular record, that entry's data block is replaced with the new data. In order to append data to a record, it must first be read and a composite data block created in memory.

Transactions are used to insure that multiple operations are completed as a unit. None of the records modified with a *TRANSACT* object are changed until its *Commit ()* procedure is called. At that time, all of those records are modified at once with an atomic operation. If *Commit ()* returns *FALSE*, an error has occurred and the data store remains unchanged.