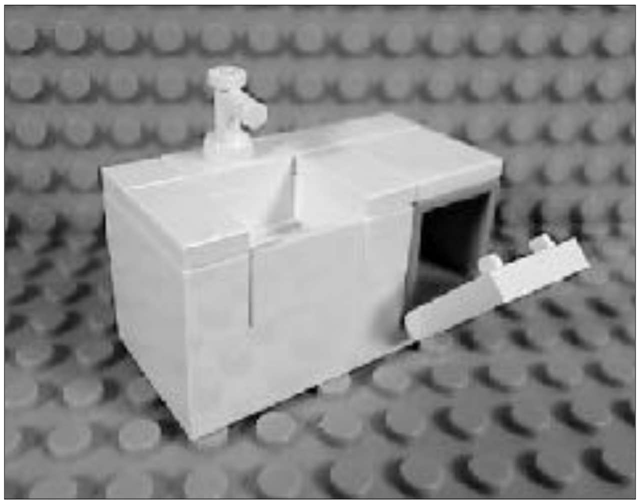


Odielib: A C Accelerated Math Library for Tcl

Presented to the 24th Annual Tcl Developer's Conference (Tcl'2017)
Houston, TX
October 16-20, 2017

Sean Deely Woods
Senior Developer
Test and Evaluation Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 20185
Email: yoda@etoyoc.com
Website: http://www.etoyoc.com

Abstract:

Odielib is a collection of tools and C accelerated math functions that I have collected over time, and repackaged in a variety of ways for several projects. The most interesting parts of Odielib for most users will be the 3d vector arithmetic and transformation functions, which are implemented as a custom TclObj type. This paper will describe some of the complexities of implementing custom TclObjs, as well as novel strategies used by this library for managing how data structures interact with the interpreter.

Version 1.0.1

Why This Paper Is For You

If you picked up this paper, and have gotten past the abstract you want to know what goodies await you by the end. While there is a lot to tell, here is a quick summary:

1. A mathematical framework for performing common mathematical operations for 2d and 3d graphics. Included in that framework namespaces tailored to solve specific problems in everyday Tcl/Tk programming.
2. Custom TclObj types which allow the outputs of many math functions to be stored and reused as Tcl values.
3. Several C accelerated TclOO classes to shepherd data sets.
4. 2d polygon operations, 3d polygon operations, and the ability to decompose polygons into line segments and reconstitute line segments into polygons.
5. A Tcl based build system which allows new functions to be dropped in with a minimum of fuss and bother.
6. The entire library packs itself into a single C source file amalgamation (similar to Sqlite). If you can't stand the build system, you can swipe one C file and one C header and add odielibc into your own project.

I want to play with this now

If you can't wait until the end of this paper to start playing, feel free to grab the fossil repository at http://fossil.etoyoc.com/fossil/odielib and compile it yourself. It can build as a standard(ish) TEA package:

```
fossil clone http://fossil.etoyoc.com/fossil/odielib odielib.fos
fossil open odielib.fos
tclsh make.tcl library
```

There is a standard autoconf and **Makefile**, but you can save yourself a step by just calling the Tcl based build system directly. The **make.tcl** script understands all of the standard operations, with the added ability to take arguments in for operations like install.

Taking Odielib for a spin

With your library built and in your path, your can start to play with the functions. Odielib tries very hard to blend in with Tcl. The commands return valid tcl values, and while accelerated in C, they still format nicely as strings and lists.

To your right is a quick and dirty demo of some basic vector arithmetic. The **::vector** namespace handles generic vectors (up to 16 dimensions). The output is the minimum size that will accommodate the dimension of the largest operand.

Vectors output by the **::vector** namespace are a custom Tcl_Obj that stores values as a C array of doubles. That same Tcl_Obj type is used for the other families of vectors operations. And because each namespace only deals with one family of vectors, it can do nice things like massage an empty list {} into {0 0 0}.

```
package require odielibc
set a {0 1}
set b {1 2 3}
set c [::vector::add $a {1 2 3}]
puts $c
{1 3 3}
::vectorxyz::midpoint {} $c
{0.5 1.5 1.5}
```

Background

Odielib is an offshoot of the Integrated Recoverability Model (IRM). IRM builds and displays models of ships. To help our users make sense of the model, we work very hard to make the graphical displays resemble general arrangements drawings. Many of the fundamental concepts (such as the wallet and slicer) were implemented by my predecessors on the project, including Richard Hipp and Clif Flynt. Over the 10 years I have been developer, more functions have been added, and the project has evolved considerably.

Until recently many of these functions were internal to the Integrated Recoverability Model's C library. For one customer we had to make a viewer which, if reverse engineered, would not expose our proprietary simulation code. The decision was made to split our viewer functions out from our simulations functions. Because of the general purpose nature of these facilities, we decided to open source the library as a gesture of good will to the Tcl community.

The Tcl API

Odielib breaks math functions into several domains. All of the domains share an internal representation for Vectors and matrices that are optimized for the most complex structure we support: a 4x4 affine transformation. Internally, all vector values are 4x4 arrays of doubles with a marker for how many rows and columns we are actually using. Vector values also are marked with a "form" tag, allowing us to distinguish between, say, polar coordinates and cartesian coordinates. Here is a breakdown of the math domains Odielib supports:

Namespace	Description	Example
::affine4x4	4x4 matrix of values expressed as a list of lists of 4 double values, or a single list of 16 double values.	::affine4x4::identity {{1.0 0 0 0} {0 1.0 0 0} {0 0 1.0 0} {0 0 0 1.0}}
::matrix	Generic two dimensional array as lists of lists.	(Usage varies)
::polygon	A data structure expressed as a series of duples mapping the perimeter of a 2d polygon	::polygon::create {0 0 10 0 10 10 0 10}
::polygonxyz	A data structure expressed as a series of 3 element lists representing the perimeter of a 3d polygon	::polygonxyz::create { {0 0 0} {10 0 0} {10 10 0} {0 10 0}}
::quaternion	A data structure expressed as a 1x4 matrix	::quaternion::create {0 0 0 1}
::shapes	A suite of routines designed to speed up the plotting of vector graphics on a tk canvas	(Usage Varies)
::vector	Generic Vector of N dimensions as lists	::vector::add {1 1} {1 2 3}
::vector2d	Two dimensional vectors represented as a flat list of doubles	::vector2d::add 1 1 2 3
::vector3d	Three dimensional vectors represented as a flat list of doubles	::vector3d::add 1 2 3 4 5 6
::vectorxy	Two dimensional vectors as Tcl values	::vectorxy::add {1 1} {1 2}
::vectorxyz	Three dimensional vectors as Tcl Values	::vectorxyz::add {1 1} {1 2 3}

Odielib also implements several TclOO based “data sets”. These are C data structures created and managed by a TclOO object. Those classes are:

Class	Description
::odielib::entity	A container for database records to cache properties used by individual entities, which are overlaid atop the properties of the type.
::odielib::plotter	A tool to maintain math transforms to allow 2d data points to be scaled and plotted using arbitrary zoom factors
::odielib::polygonhull	A set of 3 dimensional faces which represent shapes and figures in space
::odielib::segset	A set of 2d line segments which can be constituted into one or more polygons
::odielib::simtype	A container for database records to cache properties used by a set of like entities
::odielib::slicer	A set of building levels used to splay 3 dimensional layouts across a 2 dimensional canvas
::odielib::wallset	A set of 2d line segments on a canvas representing walls and rooms

Usage

Odielib was not designed from the top down, it evolved from the ground up. If a process was too slow in Tcl, we added a C accelerated routine. Over the years patterns emerged and a certain logic evolved. Well, 2 patterns actually. Which pattern depends on your ultimate product: vectors as values or vectors as as stream of doubles.

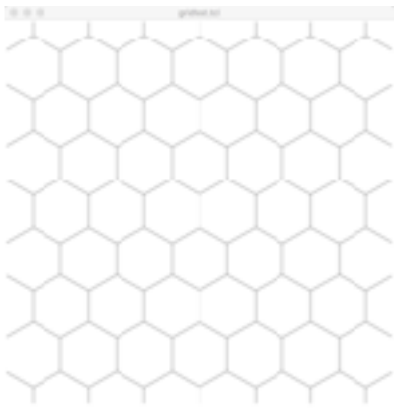
Vectors as Streams of Doubles

The **polygon**, **shapes** and **vector2d** namespace are tailored to speed up operations on a Tk canvas. The arguments and outputs take in streams of double values where it is assumed all of the odd indices are X and the even are Y. The also output data as a stream of double values.

```
::vector2d::add 12 34 56 78
> 68.0 112.0
```

In this longer example we use **::polygon::hexgrid_location** to compute the cartesian point where index X and Y occur for a hexagon grid. We then use the **::polygon::hexagon** function to generate the perimeter of the shape, now that we know the center and size. The hexagon function returns a stream of doubles, which is a format that the Tk canvas can use to define a shape. The result is the honeycomb pattern below:

```
package require Tk
set gridsize 100
canvas .c -width 600 -height 600 ; grid .c
for {set x 0} {$x < 10} {incr x} {
 for {set y 0} {$y < 10} {incr y} {
   ::polygon::hexgrid_location $gridsize $x $y 0 cx cy
   .c create polygon \
    [::polygon::hexagon $cx $cy $gridsize $gridsize 0] \
     -fill {} -outline grey -tags hex#$x,$y
 }
}
```



Odielib has utilities that allow us to interact with polygons without the Tk canvas. This is especially important for non-graphical applications. Let us say we had a batch routine that needed to understand how objects interacted with a hex grid similar to our previous example.

```
for {set x 0} {$x < 10} {incr x} {
 for {set y 0} {$y < 10} {incr y} {
   ::polygon::hexgrid_location $gridsize $x $y 0 cx cy
   dict set cell $x.$y [::polygon::hexagon $cx $cy $gridsize $gridsize 0]
 }
}
set X [::polygon::rectangle 50 50 50 50]
foreach {idx poly} $cell {
  set isect [::polygon intersect $poly $X]
  if {[lindex $isect 0]>0} {
    puts [list $idx {*}$isect]
  }
}
Output:
0.0 96.63070659909184 27.75 31.40544456622768
0.1 294.48119994212817 30.25 64.53941916244324
1.0 2113.0 53.5 49.5
```

In the above example, we populate a dict called **cell**. Every value in the **cell** dict is a data structure which can answer many questions relating to that shape. Those data structures are arguments to geometry tests in the ::**polygon** namespace. In this example we are using the **polygon intersect** function to measure how much of each cell overlaps a test polygon **X**.

Vectors as Values

```
::vectorxy add {12 34} {56 78}
> {68.0 112.0}
::vectorxyz add {} {}
> {0 0 0}
```

The **affine3x3**, **affine4x4**, **matrix**, **quaternion**, **vector**, **vectorxy** and **vectorxyz** namespaces treat vectors as values. Vectors are input and output as singular Tcl values. Kind of like passing lists around. Except the values aren't lists, they are the internal matrix data structure of Odielib. On object creation all of the elements of the matrix are initialized to zero, and the rows and columns are assumed to be zero. This is primarily to keep bad inputs from causing segmentation faults. But it has the nice side effect of allowing the user to add empty set to empty set and get {0 0 0}.

Because they share a common TclObj representation, passing data produced from one namespace as an argument to a function in another namespace is perfectly safe and acceptable. What the functions produce may be utter garbage. But it will be safe from a "will not crash Tcl" perspective. The price of decent performance is that we have to assume the user understands the math he or she is trying to use.

Tcl as an Expression Engine

Odielib does not provide an expression engine equivilent to the vexpr command in Vectcl. Well at least not anymore. In the early days it did provide an Reverse Polish Notation based stack expression system. But the architecture proved unwieldy and it was later removed. Though references may still exist in the code.

For the types of math operations Odielib is performing, formal orders of operation do not exist. Or worse, they may exist and there are competing standards. PEMDAS and other math conventions are no guide whatsoever.

The case below is an attempt to use the math of Odielib as a physics engine, albeit a simplistic one. A massless object in a frictionless space is imparted with angular momentum in the direction of precession. Attached to the center of this object is a thruster imparting an acceleration of 1 unit in whatever direction the object is facing.

```
# Calculate the location, orientation, thrust, and spin
# of an object in 3d space
set location       [vectorxyz create {0 0 0}]
set attitude       [affine4x4 identity]
set attitude_delta [affine4x4::rotate_precession 0.9]
set deltap         [vectorxyz create {1 0 0}]
set velocity       [vectorxyz create {0 0 0}]
for {set x 0} {$x < 10} {incr x} {
  # Compute one step
  affine4x4::*= attitude $attitude_delta
  set F [vectorxyz transform $attitude $deltap]
  vectorxyz::+= velocity $F
  vectorxyz::+= location $velocity
  puts [list f: $F x: $location v: $velocity]
}
```

The output of our experiment demonstrates the complexity of physics once you get rotation involved:

```
f: {0.62161 0.783327 0} x: {0.62161 0.783327 0} v: {0.62161 0.783327 0}
f: {-0.227202 0.973848 0} x: {1.01602 2.5405 0} v: {0.394408 1.75717 0}
f: {-0.904072 0.42738 0} x: {0.506354 4.72506 0} v: {-0.509664 2.18455 0}
f: {-0.896758 -0.44252 0} x: {-0.900069 6.46709 0} v: {-1.40642 1.74203 0}
f: {-0.210796 -0.97753 0} x: {-2.51729 7.23159 0} v: {-1.61722 0.764504 0}
f: {0.634693 -0.772765 0} x: {-3.49981 7.22333 0} v: {-0.982526 -0.00826063 0}
f: {0.999859 0.0168139 0} x: {-3.48248 7.23189 0} v: {0.017333 0.00855327 0}
f: {0.608351 0.793668 0} x: {-2.8568 8.03411 0} v: {0.625684 0.802221 0}
f: {-0.243544 0.96989 0} x: {-2.47466 9.80622 0} v: {0.38214 1.77211 0}
f: {-0.91113 0.412118 0} x: {-3.00365 11.9904 0} v: {-0.52899 2.18423 0}
```

The Build System

The build system for Odielib is the **Practcl** concept I presented at the 2016 conference. **Practcl** uses TclOO objects as containers of configuration data, as actors in synthesizing connector code, and as proxies to access elements of different tool sets. The tool sets supported are currently GCC/Automake and Microsoft Visual Studio. Though I have to admit that MSVC is a bit of a work in progress.

Markup exists to capture C functions, C implemented Tcl Commands, Data Structures, and C implemented TclOO methods. Practcl is targeted for the experienced C developer. It doesn't try to do argument validation. It doesn't deduce what the output to the interpreter will be. It's just a wrapper for raw C code.

The markup for a simple function is given on the right. "c_tclcmd" is a method for an object. It is assumed that the developer is familiar with the template for Tcl object commands, and can live with the standard names: *clientData*, *interp*, *objc* and *objv*. The body C code is exactly what would be entered if the functions was cut by hand. However, the transformation of the command name to a C function name, preparing the namespace, and injecting the command into the interpreter are all handled automatically by the build automation.

```
# A Tcl command in Practcl markup
my c_tclcmd  ::thestate::two_plus_two {
  Tcl_SetObjResult(interp,Tcl_NewIntObj(5));
  return TCL_OK;
}

# A C function in Practcl markup
my c_function {static inline double
Vector_GridScaler(double x,double grid,double grain)
} {
  double q;
  q=grid*round(x/grid);
  if((x-q)>grain) {
    q+=grain;
  }
  return q;
}
```

A plain C function is similar. The code itself is C, we are just using Tcl as a wrapper to capture what we need to integrate that code. We also include all of the information that will be needed to prototype the function. Depending on the function's role the prototype will be in either the library's header files (for exported functions) or in a declaration block of the C file (for static functions.)

Practcl can also integrate static C source files and headers, as well as compiled libraries. This is particularly important for bringing older projects up to speed. You don't need to rewrite your entire project in Practcl, just the parts your are trying to improve.

Because Practcl is running inside of a Tcl interpreter, we can also do things like unfold loops or template repetitive code. Now on the surface unrolling a loop sounds like something your compiler would do for you. Except that most applications are compiled with an -O2 or -Os level of optimization. Few applications utilize -O3, because it has a tendency to "optimize" beyond the point of actually improving the software's performance.

On the next page I implement a 4x4 matrix multiplication using ~23 lines of Tcl code. While that is not a terrible function to optimize by hand, it's not terribly fun either.

```
set body {}
for {set i 0} {$i < 4} {incr i} {
  for {set j 0} {$j < 4} {incr j} {
    append body "\n  double cell_${i}_${j}\;"
  }
}
for {set i 0} {$i < 4} {incr i} {
  for {set j 0} {$j < 4} {incr j} {
    append body \n "  cell_${i}_${j}="
    set terms {}
    for {set k 0} {$k < 4} {incr k} {
      lappend terms "A\[AFFINE_IDX_${i}_${k}\]*B\[AFFINE_IDX_${k}_${j}\]"
    }
    append body "[join $terms +]\;"
  }
}
for {set i 0} {$i < 4} {incr i} {
  for {set j 0} {$j < 4} {incr j} {
    append body \n "  R\[AFFINE_IDX_${i}_${j}\]=cell_${i}_${j}\;"
  }
}
my c_function {inline void Odie_Affine4x4_Multiply(AFFINE R,AFFINE A,AFFINE B)} $body
```

The advantage here is that the Tcl code still bears some resemblance to the algorithm one would find in a math textbook. The C code produced, however, is pretty tightly optimized. We use Tcl's loop constructs to walk through the logic of the algorithm, and we just store the final motions as C code.

The C API

Odielib is also designed to play well with others on the C API level. Of particular concern to most developers is how the Tcl_Obj is designed, how it stores its values, and how can it be swiped for their own projects.

Inside of Tcl values, all vectors and matrices are of type Odie_MatrixObj. That form allows for up to 128 rows and 128 columns. It also has a placeholder for "form" which define rules for how the matrix can be used and/or transformed by functions which need other forms. Units is for internal use of the forms. It is currently used for polar coordinates to indicate if the values are in radians or degrees.

```
struct Odie_MatrixObj {
  int refCount;
  char rows;
  char cols;
  char form;
  char units;
  double matrix[];
};
struct MatrixForm {
  int id;
  const char *name;
  int rows;
  int cols;
  const char *description;
  const char *(*xConvertToForm)(MATOBJ*,int form);
};
```

Each matrix form also registers a data structure. That data structure includes its expected size, and a placeholder for a function to transform other forms into it (or fail in the process.)

As a safety and convenience all matrices exported to the Tcl interpreter are pre-allocated to hold the largest matrix supported by the system. Currently a 4x4 affine matrix.

All vector and matrix C functions accept a pointer to an array of doubles as an input. I have also tried to enforce the convention that the output is always the first argument.

Odielib provides the function:

```
int Odie_GetMatrixFromTclObj(
  Tcl_Interp *interp, Tcl_Obj *tclObj, int form, Odie_MatrixObj **result
);
```

This function works in a similar way to **Tcl_GetIntFromObj** or **Tcl_GetDoubleFromObj**. The library will massage *tclObj* into an **Odie_MatrixObj** value and then set the result to the Tcl_Obj's internal representation. If the function returns anything other than *TCL_OK*, something went wrong and meaningful errors will be written to the *interp*.

Several functions exist to generate new **Odie_MatrixObj** data structures, depending on the information you have about them. All of the forms take care of allocating the memory for both the structure and the double array it tracks. The allocator uses the trick of allocating both the structure and the array at the same time in the same in the same *Tcl_Alloc*, so that freeing the **Odie_MatrixObj** will also free the double array. Also, the allocator provides memory that is zeroed out. With that said, Odielib still has to maintain its own refcount on data structures because the same Odie_MatrixObj could be sent multiple times as different Tcl_Obj structures. To manage the refcounts the library provides:

```
void Matrix_Free(Odie_MatrixObj *matrix);
```

When it comes time to write a matrix value back to the interpreter, Odielib provides a Tcl_Obj packing and encoding function. This function takes in an Odie_MatrixObj data structure, and returns a Tcl_Obj with that structure packed into the internal representation.

```
Tcl_Obj *Matrix_To_TclObj(Odie_MatrixObj *matrix);
```

Odielib increments its own internal refcount on structures fed into **Matrix_To_TclObj()**, and decrements the refcount on structures fed to **Matrix_Free()**.

A simple Tcl API function looks like the example below. **Odie_Matrix_To_Fit** is a routine that measures the inputs, computes the minimum size of the matrix to conform to both inputs, and returns a new structure with a compatible form and size.

```
my c_tclcmd  ::vector::add {
  Odie_MatrixObj *A,*B,*C;
  int i,size_c;
  if(objc < 3) {
    Tcl_WrongNumArgs( interp, 1, objv, "A B" ); return TCL_ERROR;
  }
  if(Odie_GetMatrixFromTclObj(interp,objv[1],MATFORM_null,&A)) return TCL_ERROR;
  if(Odie_GetMatrixFromTclObj(interp,objv[2],MATFORM_null,&B)) return TCL_ERROR;
  C=Odie_Matrix_To_Fit(A,B);
  size_c=C->rows*C->cols;
  for(i=0;i<size_c;i++) {*(C->matrix+i) = *(A->matrix+i) + *(B->matrix+i);}
  Tcl_SetObjResult(interp,Matrix_To_TclObj(C));
  return TCL_OK;
}
```

Implementing New Matrix Forms

New matrix forms are registered via the **odielib::vexpr_argtype** procedure in the Odielib build system. In the next example we'll be working with a custom datatype to represent a vector in three dimensions. The internal calls to this library do not need to know about the rest of Odielib. And in many cases, it is better to pass copies of operands around instead of the pointer to the original. To that end, our **vectorxyz** type has its own **Odie_GetVectorXYZFromTclObj** function which populates an existing double array instead of allocating a new one. It also provides its own **VectorXYZ_To_TclObj** which will copy the contents of an array into a new **Odie_MatrixObj**.

This custom type can also interact with **Odie_MatrixObj** structures directly, so we also demonstrate a modified add function that will read and write a value back to a Tcl variable. One of the nice/scary features of Odielib's value system is that we can modify values out from under Tcl's nose. We just need to be sure to invalidate the string representation.

Because the c functions all take **double *** as arguments, passing our Tcl value's internal value is as simple as accessing the *.matrix* field of the structure.

```
# In vectorxyz.tcl
# Define the custom type
::odielib::vexpr_argtype vector_xyz {
  typedef VectorXYZ
  forms {} rows 3  cols 1
  description {vector: X Y Z}
  function-convert Matrix_To_cartesian
}

# Add a C math routine
my c_function {static inline void VectorXYZ_Add(
VectorXYZ C,VectorXYZ A,VectorXYZ B
)} {
  C[X_IDX]=B[X_IDX]+A[X_IDX];
  C[Y_IDX]=B[Y_IDX]+A[Y_IDX];
  C[Z_IDX]=B[Z_IDX]+A[Z_IDX];
}
##
# Link the C routine to the Tcl API
##
my c_tclcmd  ::vectorxyz::add {
 VectorXYZ A,B,C;
 if(objc < 3) {
   Tcl_WrongNumArgs( interp, 1, objv, "A B" );
   return TCL_ERROR;
 }
 if(
Odie_GetVectorXYZFromTclObj(interp,objv[1],A)
) return TCL_ERROR;
 if(
Odie_GetVectorXYZFromTclObj(interp,objv[2],B)
) return TCL_ERROR;
 VectorXYZ_Add(C,A,B);
 Tcl_SetObjResult(interp,VectorXYZ_To_TclObj(C));
 return TCL_OK;
}
##
# Addition that writes back to a variable
##
my c_tclcmd  ::vectorxyz::add_inplace {
  Odie_MatrixObj *A;
  VectorXYZ B;
  Tcl_Obj *varname;
  if(objc < 3) {
    Tcl_WrongNumArgs( interp, 1, objv, "A B" );
    return TCL_ERROR;
  }
  varname=Tcl_ObjGetVar2(interp,objv[1],NULL,0);
  if(
Odie_GetMatrixFromTclObj(interp,varname,MATFORM_vectorxyz,&A)
) return TCL_ERROR;
  Tcl_ResetResult(interp);
  if(
Odie_GetVectorXYZFromTclObj(interp,objv[2],B)
) return TCL_ERROR;
  VectorXYZ_Add(A->matrix,A->matrix,B);
  Tcl_InvalidateStringRep(varname);
  return TCL_OK;
}
```

In Conclusion

This paper was a brief overview of the capabilities, design, and implementation of Odielib. The project is managed as a fossil repository at: http://fossil.etoyoc.com/fossil/odielib

My hope is that Odielib's ad-hoc architecture could form the basis of a numerical library for Tcl, on the same level as NumPy for Python. I could certainly use some help with writing new modules, regression tests, and documentation. Feel free to contact me at: yoda@etoyoc.com if you have questions, suggestions, (or even better) contributions. I do maintain a public sandbox for Odielib at http://fossil.etoyoc.com/sandbox/odielib, and that project allows users to self register and contribute code immediately. Changes are pulled into the sandbox from the official project nightly, though no changes are ever pulled in the other direction. When something earth shatteringly awesome does appear in the Sandbox, my plan is to just make a diff and apply a patch.

Acknowledgements

I would like to specially thank Richard Hipp and Clif Flynt. They were my predecessors on the IRM, and many of their designs have carried through into Odielib. I would also like to thank T&E Solutions for the time to develop much of this project on the clock as part of my regular duties.

While I'm at it, let me thank my wife Ginger for keeping and eye on the kids for a bit while I banged this paper together, my late Uncle Matt for pirating a copy of Turbo C for me back when I was a teenager, my Mom for buying my first computer, Claude Shannon for inventing information theory and Ogg, for inventing fire. At least I think his name was Ogg. The "gg" sound in his name doesn't really have a parallel in modern phonics.

The cover image was downloaded from:

http://img.auctiva.com/imgdata/0/9/6/1/1/2/webimg/575133520_tp.jpg

I was doing an image search for "Lego Kitchen Sink", and that image sort of spoke to me.