A CMake-Based Cross Platform Build System for Tcl/Tk

Clifford Yapp
Quantum Research International Inc.
Prepared under contract W911QX-06-F-0057 for the U.S. Army Research Laboratory*

October 6, 2011

Abstract

Defining build logic for a large software package in multiple software development environments entails a large up-front implementation cost and an ongoing maintenance burden. CMake is an open source cross–platform build tool that allows developers to define relatively abstract build logic that is automatically translated into a variety of build system formats, reducing the burden of supporting multiple development environments. BRL-CAD's integration of Tcl/Tk as a sub-build motivated the development of Tcl/Tk build logic compatible with BRL-CAD's new CMake logic. This paper presents a new CMake based build system for Tcl/Tk and a number of popular Tcl/Tk extensions.

Introduction

Large scale software projects require the development and maintenance of *build logic* governing the compilation, packaging, and installation source code. This logic is the interface between compilation tools (compilers, documentation processors, etc.) that actually translate code into usable form and the code itself. As such, it is the build logic that must identify any idiosyncrasies present in the system's compilation tools and libraries. Once identification is complete, the build system must also generate instructions the compiler can use to compensate for these differences. Over the years, each major software development platform has created systems to manage this process. Microsoft Windows has Visual Studio, Mac OS X has XCode, and most Unix/Linux style platforms have some form of Make, often augmented by GNU Autotools. Each of these tools manages this process for different versions of their specific platform, but generally support *only* that specific platform. This presents a challenge for cross–platform projects such as Tcl/Tk, which must build in *all* of these environments to achieve their goals.

The responsibilities of the build logic include (but are not limited to) listing source files, identifying compilation tools and options needed for files, identifying target libraries and executables, and sometimes expressing the logic for generating user-installable packages of the finished package. While specific compilation instructions are typically unique to each operating system and tool, the actual *task* to be accomplished is often the same. For example, when building a C library, many if not all of the C files themselves are common to all platforms. Despite this commonality, the addition of a single new C file requires altering not one but n build files where n is the number of build systems that need to be defined in order to support all targeted development platforms.

CMake[3] is a *metabuild* system designed to alleviate much of this problem by abstracting build logic one level above makefiles, XCode projects, and Visual Studio projects. Given portable source code, the build logic is expressed in a CMakeLists.txt file that gets translated by CMake into platform native logic using *generators*. The developer then uses the standard system tools to complete the build, and logic common to all platforms is expressed (and need only be updated) in a single set of build files.

Tcl/Tk faces precisely this cross-platform development problem, making the project a good conceptual match for CMake. However, until now Tcl/Tk's

*Approved for public release; distribution is unlimited.

existing build systems have proved adequate for most real-world production use. The marginal benefits of CMake were insufficient to justify both the effort of re-implementing Tcl/Tk's build system and the disruption of existing work-flows. Given these constraints, it is understandable that a cross–platform CMake build had not already been implemented for Tcl/Tk.[1]

Motivation and Requirements

BRL-CAD[1] is an open source Computer Aided Design software package developed by the Ballistic Research Laboratory (now the U.S. Army Research Laboratory.) BRL-CAD has made extensive use of Tcl/Tk since the earliest days of its development. Because so many of BRL-CAD's core abilities depend on Tcl/Tk, availability of Tcl/Tk on a targeted platform is a core requirement for deploying BRL-CAD on that platform. BRL-CAD has a long-standing policy: if system versions of required libraries are either absent or insufficiently modern at configuration time, the BRL-CAD build will utilize local copies of those libraries. As part of comprehensive configuration control, testing and dependency management, BRL-CAD bundles pre–configured copies of all external dependencies. In addition, it has occasionally been necessary to modify such libraries (Tcl among them) to support BRL-CAD's needs or fix bugs encountered. Modifications are contributed back upstream to the primary development teams when possible. It is much simpler to use upstream sources than to maintain a separate version of the source code. However, BRL-CAD deployment cannot wait on those fixes propagating through both the upstream acceptance and customer system upgrade processes. Moreover, the BRL-CAD developers need to be able to verify and validate BRL-CAD functionality for a given configuration that is independent of any platform environment. Consequently, BRL-CAD *must* be able to compile its own local copy of Tcl/Tk at need.

Tcl/Tk has supported multiple platforms for many years, but it currently uses the Tcl Extension Architecture (TEA) autoconf macro system on platforms using Make and either NMake or Visual Studio (MSVC) project files for native[2] Windows compilation. This presented a difficulty for the BRL-CAD project in that neither of these systems integrated well with BRL-CAD's own build systems. As a workaround, BRL-CAD used custom Microsoft Visual Studio files on Windows. On other platforms Tcl/Tk's own build system was usable with a Makefile.am wrapper. This approach worked but represented an undesirable ongoing maintenance overhead.

In the summer of 2010, the decision was made to unify BRL-CAD's build system infrastructure into a single CMake–based system in order to reduce long term maintenance costs and simplify building Windows releases. As most of BRL-CAD's core developers do not use Windows on a day–to–day basis for development, a single cross platform build system would mean build logic written or updated on non-Windows platforms would stand a good chance of working without extensive effort. However, to achieve the desired result the new system would have to build not just BRL-CAD but *all* of its bundled dependencies – including Tcl/Tk.

The initial attempt to integrate Tcl/Tk into a CMake-based BRL-CAD build made use of CMake's ExternalProject_Add functionality for triggering external build systems as sub-builds. Had this worked smoothly on all platforms, it would have been the simplest solution. With Make–based systems, the attempt was reasonably successful despite the drawback of *requiring* installation of the sub-build libraries before the CMake build itself could proceed. MSVC proved to be a considerably greater challenge – between difficulties integrating Visual Studio project files and the problems involved with running NMake build scripts from within Visual Studio, the initial attempts to integrate Tcl/Tk's own Windows build files were not successful. Rather than continue to struggle with the complexity of triggering multiple external build systems on multiple platforms, focus shifted to the integrated approach – implementing enough CMake logic to build the parts of Tcl/Tk needed for BRL-CAD. Implementing CMake build logic for Tcl/Tk would reduce the maintenance burden to a single system for all platforms and integrate well with BRL-CAD's new build logic.

A CMake-based build system for Tcl/Tk needs to satisfy the following requirements:

1. Build Tcl/Tk successfully on Windows (using MSVC), Linux, FreeBSD, Solaris, and Mac OS X from a single set of CMake build files.

2. Implement enough of the Tcl/Tk–specific compilation macro logic in CMake to support build-

[1]Twylite's Coffee project uses CMake to build Tcl, but is primarily focused on Windows: see http://dev.crypt.co.za/coffee
[2]"Native" in this case being defined as building without the use of Unix compatibility environments such as Cygwin.

ing Tcl/Tk on BRL-CAD's target platforms – the goal was to avoid significantly altering the Tcl/Tk source code itself.

3. Run tclsh and wish from within the build directory, without requiring installation. This is a necessity for BRL-CAD, which makes use of Tcl in its own build logic and must run tclsh prior to the installation step.

4. Support compilation of Tcl/Tk extensions, either in conjunction with BRL-CAD's own copy of Tcl/Tk or using a system Tcl/Tk. BRL-CAD sometimes needs to compile Tcl/Tk extensions even if a system Tcl/Tk satisfies the feature and version requirements, hence build logic for those extensions needs to support both cases.

Building Tcl/Tk – What It Takes

CMake provides very general mechanisms for expressing build logic, but still requires that any project-specific compiler options be included by the developer. It also requires that specific functionality tests for libraries, header checks, function checks, etc. be set up in the CMakeLists.txt file(s) according to the needs of the particular software in question. Hence, the first step in writing new build logic for Tcl/Tk was to examine the existing build logic to determine what functionality it provides.

Tcl Extension Architecture – Strong TEA

The venerable TEA system[4] implements a large number of tests designed to identify platform specific issues and quirks that may affect Tcl when trying to build. It also defines standard layouts, platform specific compiler flags, and a wide variety of other settings evolved over many years. It utilizes autoconf from the GNU Autotools suite.

There are two versions of this logic – one in Tcl/Tk proper whose macros use a SC_ prefix (SC standing for Scriptics) and an extended version using the TEA_ prefix used with extensions. Both files are named tcl.m4, and a comparison of the two reveals a great deal of shared code, but the tcl.m4 with TEA prefixes is regarded as the "official" TEA. System functionality tests (such as missing POSIX headers) required for compilation were of primary interest to a CMake effort. Detection of installed Tcl/Tk configurations is the responsibility of the FindTCL.cmake macro – that being the case, it was not necessary to translate TEA macros pertaining to Tcl/Tk configuration detection into the primary CMake build logic.

Because platforms such as HPUX, IRIX, and SCO Unix are no longer supported by BRL-CAD, logic specific to supporting them was not needed in the first–cut implementation of CMake logic. Hence, the decision was made to only implement as much of TEA's functionality as was needed for BRL-CAD's target platforms rather than attempting a full TEA implementation in CMake from the get–go.

Visual Studio, NMake, and MSYS/MinGW

Microsoft Windows–based software compilation is accomplished using a wide variety of development environments, some of which bear little resemblance to the standard Unix tools. One of the most common tools for building software on Windows is Visual Studio's Integrated Development Environment. Visual Studio also provides a command line utility called "nmake" which is similar in spirit to the Unix style Make. The open source community has produced compilation environments for Windows, notably GNU gcc within the Cygwin Unix emulation environment and the MinGW environment (often used with MSYS) which can produce native Windows binaries. Tcl's README indicates that the Cygwin environment is *not* supported – MinGW/MSYS and Visual C++ 6.0 + nmake.exe are the standard tools.

Visual C++ compiler flags have little in common with those supported by most open source C/C++ compilers, and there is not really a direct MSVC analog to the Autotools *configure* step. Feature detection on Windows is generally restricted to Unix-style emulation environments such as Cygwin. The introduction of CMake allowed for many new possibilities in that respect when building on Windows.

The Structure of Tcl/Tk – Separate But Intertwined

The first survey of the Tcl/Tk building system prompted the question "why not just generate a tcl_config.h header file to hold all of these options, instead of building up definitions on the command line?" A small trial quickly demonstrated that there is indeed a reason for the current Tcl/Tk approach. Tk makes use of "internal" Tcl headers. In order to build Tk, it is necessary to specify the location of a Tcl source archive. These internal Tcl headers in turn need proper definitions from the configuration logic. However, when building Tk, a hypothetical *Tcl* generated tcl_config.h header is not guaranteed to be present. If the Tk and Tcl builds are treated as separate systems, Tk would have to re-generate the *Tcl* configuration header in addition to its own and sort out how Tk headers might pull in *either* or *both* tcl_config.h and tk_config.h. Under the circumstances, it is simpler just to supply any needed definitions via command line arguments to the compiler – these are passed through to all headers as needed.

Unfortunately, this use of "internal" headers is also a fact of life in several common third party Tcl/Tk packages. Tcl/Tk 8.6 is introducing a new pkgs directory to help address this problem, but that only avoids the issue by allowing sub-build logic to assume a fixed parent location for source files. Another approach, used by the Visualization ToolKit (VTK), is to include local copies of various versions of the Tcl/Tk internal headers with the package source itself. Regardless of the approach used, it complicates the building (and build logic) of Tcl/Tk extensions.

Beyond straight C compilation, Tcl/Tk extensions also require pkgIndex.tcl files that instruct Tcl/Tk how to load that particular extension. This is of particular concern to BRL-CAD, because experience has shown it is all too easy to create confusing and dysfunctional situations when multiple Tcl/Tk installations are present. If Tcl's *auto_path* variable happens to be set in such a way that a local Tcl/Tk finds packages in a system Tcl/Tk installation, the results can be "almost working" runs of Tcl scripts that fail in cryptic and mysterious ways.

The CMake Build System

A full introduction to CMake is beyond the scope of this paper – for a more complete overview see Martin and Hoffman[2]. The focus here will be on differences between the TEA build system and CMake, as well as CMake solutions to particularly tricky compilation and installation issues.

Running CMake

Building Tcl/Tk with CMake is similar to the TEA build cycle, but the command line syntax and configuration options are somewhat different – see Table 1 for a mapping between TEA options and CMake.[3]

CMake itself can be run one of three ways – either as a straight command line program (cmake), with a curses based interface (ccmake), or with a graphical interfaces based on the Qt toolkit (cmake-gui). To specify settings on the command line, the prefix "-D" is used – e.g. -DCMAKE_INSTALL_PREFIX="prefix" instead of –prefix="prefix". All three front ends support the same basic abilities, although the Qt graphical interface in particular supports some nice extra features that help a new developer discover the system. When using the graphical or curses–based interfaces instead of the command line, *configuration* (detecting system characteristics) and *generation* (actual writing of the build files) are separate operations. The command line cmake binary combines both of these steps into one operation.

Layout

The basic source code layout of Tcl/Tk has not been altered, but the location of the CMake files relative to the source files *is* different than the corresponding TEA/win32 files. While the unix subdirectory contains the bulk of the TEA logic and the win subdirectory contains Windows specific build files, the primary CMakeLists.txt file that specifies sources for all platforms lives in the top level directory. The library and doc subdirectories have their own CMakeLists.txt files due to the specialized nature of the logic they require (more on this later,) but all C source code is handled by the top-level CMakeLists.txt file.

Macros defining CMake logic specific to Tcl/Tk are in a new top-level directory called CMake, in keeping with standard CMake conventions. Among the files present here is tcl.cmake, which is the closest match in the CMake logic to the original SC prefix tcl.m4 file.

For convenience, the current Tcl/Tk 8.6b2 CMake logic is organized with one higher top-level directory above Tcl/Tk and other extensions for which CMake build logic has been implemented. A small CMakeLists.txt file in this directory suffices to unify all of the subdirectories (tcl, tk and any extensions) into a single build. Among other benefits, this combines all configure stages for all of the packages into a single configure step – once a particular test is run for a particular subdirectory, CMake does not need to repeat

[3]With CMake, it is generally much better practice to run the configuration and building routines in a working directory *other* than the top-level source directory – either a subdirectory in the source tree or a directory entirely outside the source tree. For examples in this paper, a subdirectory named "build" located in the top-level source directory will be assumed.

Table 1: Configuration Options – TEA vs. CMake

Feature	TEA	CMake
Run configuration	../configure	cmake ..
Specify location of sources	–srcdir="DIR"	"DIR"
Installation prefix	–prefix=	CMAKE_INSTALL_PREFIX
Executable prefix	–exec-prefix="EPREFIX"	Not Implemented
Symlinks for manpages	–enable-man-symlinks	Not Implemented
Compress manpages	–enable-man-compression	Not Implemented
Add suffix to manpages	–enable-man-suffix=STRING	Not Implemented
Enable Threads	–enable-threads (off)	TCL_THREADS (AUTO)
Build Shared Libraries	–enable-shared (on)	BUILD_SHARED_LIBRARIES (ON)
Enable 64 Bit support	–enable-64bit (off)	TCL_ENABLE_64BIT (AUTO)
Disable rpath support	–disable-rpath (on)	N/A
Use CoreFoundation (OSX)	–enable-corefoundation (on)	TCL_ENABLE_COREFOUNDATION (ON)
Allow dynamic loading	–enable-load (on)	TCL_ENABLE_LOAD (ON)
Debugging Symbols	–enable-symbols (off)	Several CMake options
Use nl_langinfo	–enable-langinfo	TCL_ENABLE_LANGINFO (ON)
Enable "unload" command	–enable-dll-unloading	TCL_ENABLE_DLL_UNLOADING (ON)
Enable DTrace support	–enable-dtrace (off)	Not Implemented
Package as frameworks (OSX)	–enable-framework (off)	Not Implemented
Specify encoding	–with-encoding (iso8859-1)	TCL_CFGVAL_ENCODING (Defaults to cp1252 on Windows, else iso8859-1)
Install timezone data	–with-tzdata (autodetect)	TCL_TIMEZONE_DATA (AUTO)
Use Aqua windowingsystem (OSX)	–enable-aqua (no)	TK_ENABLE_AQUA (AUTO)
Use XScreenSaver	–enable-xss (on)	TK_ENABLE_XSS (AUTO)
Use freetype/fontconfig/xft	–enable-xft (on)	TK_ENABLE_XFT (AUTO)
Specify tcl source directory	–with-tcl=	TCL_SRC_PREFIX TCL_BIN_PREFIX
Use X11	–with-x	(auto)

it for the next. This painless integration of sub–builds is an important feature for BRL-CAD, and will hopefully prove a useful convenience for other developers.

Running from the Build Directory

One convenience offered by CMake is sophisticated control over the handling of run–time search paths (RPATH). With the correct options set, CMake's generated build files will set RPATH values to the correct values for build directory execution when compiling executables, and then automatically adjust them to the correct *installation* values when "make install" is run. This means developers do not even have to set LD_LIBRARY_PATH to run from the build directory, and using built-but-not-installed software within the build process itself becomes simpler.

Tcl/Tk has an additional complication beyond standard RPATH issues – pkgIndex.tcl files have to be correct for build paths in the build directory and install paths in the installation directory. The CMake solution implemented for this problem is to generate *two* pkgIndex.tcl files – one in the correct place relative to the build path locations of Tcl/Tk's files, and the other in a non-functional (within the build directory) location with the instructions to install *that* version when the time comes for installation. See Figure 1 for an example of the CMake code that achieves this for the Tk package.

Man Pages

Tcl and Tk use a shell script named installManPage to generate a large number of manual pages from a base set that are present in the Tcl/Tk doc subdirectory. This poses something of a problem in that CMake does not know ahead of time what files this script will generate, and thus cannot incorporate those generated files into its own install commands. One option would be to list explicitly every file generated by the installManPage script in the CMake logic, but this would be both extremely verbose and a maintenance burden. The solution currently in place runs the installManPage script during the configure stage and has CMake itself identify all the files generated. CMake is then aware of the full file list and can generate proper installation commands. The most significant drawback of this approach is that man page changes impacting the list of generated files require re-running CMake instead of simply re-running the build logic, but that appears to be the price that must be paid in order to allow CMake to perform installa-

```
# pkgIndex.tcl - installation location
get_target_property(TK_LIBLOCATION tk LOCATION_${CMAKE_BUILD_TYPE})
get_filename_component(TK_LIBNAME ${TK_LIBLOCATION} NAME)
file(WRITE ${CMAKE_CURRENT_BINARY_DIR}/pkgIndex.tcl
     "package ifneeded Tk ${TK_PATCH_LEVEL}
          [list load [file join $dir .. .. ${LIB_DIR} ${TK_LIBNAME}] Tk]")
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/pkgIndex.tcl DESTINATION lib/tk${TK_PATCH_LEVEL})

# pkgIndex.tcl - build directory location
FILE(WRITE ${CMAKE_LIBRARY_OUTPUT_DIRECTORY}/tk${TK_PATCH_LEVEL}/pkgIndex.tcl
     "package ifneeded Tk ${TK_PATCH_LEVEL}
          [list load [file join $dir ${CMAKE_LIBRARY_OUTPUT_DIRECTORY} ${TK_LIBNAME}] Tk]")
```

Figure 1: Example CMake pkgIndex.tcl generation logic

tion of the manual pages. BRL-CAD needs CMake to manage these generated files to ensure they are incorporated in binary packages, and the current approach meets that requirement. The routines only generate the pages if sh and sed are present, so the MSVC build does not use them.

Package Installation

Tcl includes a number of scripts that are installed in lib/tcl8, with subdirectories and file names based on the scripts themselves – for example, http/http.tcl is installed to lib/tcl8/8.4/http-2.7.5.tm in Tcl 8.5 and lib/tcl8/8.6/http-2.8.2.tm in Tcl 8.6. This location and naming appears to be based on the package version number and required Tcl/Tk version in the script itself. Initially the destination for each file was hard–coded in the library CMakeLists.txt file, but this proved problematic moving from Tcl 8.5 to Tcl 8.6. Current logic uses CMake's regular expression facilities and parses the required information from the tcl scripts themselves. This macro places all tcl8 script files correctly based on their own contents.

SC / TEA Macros

Most of the time spent in converting Tcl/Tk's build logic to CMake involved studying the macros in tcl.m4 and determining how to express their logic in CMake. After a few false starts a systematic approach proved necessary – a tcl.cmake file was organized along the same lines as tcl.m4, and whenever a test from tcl.m4 proved necessary the corresponding functionality was implemented in tcl.cmake. As of the time of this writing all SC tcl.m4 macros have not been implemented (see Table 2) but enough of them exist to successfully build on BRL-CAD's target platforms and more will be implemented if needed. Some of the TEA functionality (in particular, identifying Tcl configurations) has been expressed elsewhere in the new CMake build.

Dependent Options

Another feature available in CMake is a type of option that is displayed or not displayed based on values assigned to other options - a *dependent* option. Tk's CMake build logic makes use of this feature for features requiring the presence of X11 - the CMake GUI will not list those options for the user if the current windowing system is Win32 or Aqua. The Xft option is actually conditional on multiple variables - the Tk windowing system must be X11 and both xft and Freetype need to be found for TK_ENABLE_XFT to be displayed as an option. The code that achieves this is displayed in Figure 2.

Tcl/Tk Extensions

CMake uses pre–package routines, typically in files named according to the FindPKG.cmake template, and the *find_package* command to locate system installations of packages and libraries. CMake includes a FindTCL.cmake, but it proved insufficient for BRL-CAD. This necessitated the implementation of a new version, which has been submitted for upstream inclusion in CMake. Its distinct features include:

1. Detection of the windowing system in use by the found Tcl/Tk version (Aqua, X11, etc.). This is particularly important on Mac OS X.

2. Successful detection of a second system installation of Tcl/Tk if the first fails to satisfy specified criteria – for example, if X11 is required on OS X, the system Tcl/Tk framework will fail but an X11 version (if installed) will be found instead.

```
include(CMakeDependentOption)
CMAKE_DEPENDENT_OPTION(TK_ENABLE_XFT "Use freetype/fontconfig/xft" ON
                              "TK_SYSTEM_GRAPHICS STREQUAL x11;FREETYPE_FOUND;${X11_Xft_FOUND}" OFF)
```

Figure 2: Dependent Xft option definition in Tk.

3. Finer control of what is needed from a Tcl/Tk installation – for example, if Tcl without Tk is sufficient for a particular project, an option can be defined to indicate that to FindTCL.

BRL-CAD requires not just Tcl/Tk but a host of Tcl/Tk extensions and all of those extensions needed CMake logic of their own. For the most part routines already defined for Tcl/Tk in combination with the new FindTCL.cmake proved sufficient for both local and system Tcl/Tk extension compilation scenarios, but there were a few significant exceptions.

The use of internal Tcl headers remains a significant complication for compilation of Tcl/Tk extensions, and a system installation of Tcl/Tk is not sufficient in such cases – the Tcl source code must be available, just as in the case of Tk. In the case of BRL-CAD this situation is usually workable due to the Tcl source code being guaranteed to be available in BRL-CAD's own source tree. Currently BRL-CAD requires Tcl/Tk 8.5, but in order to support more general cases (such as using an 8.6 system Tcl/Tk) extensions need more than the Tcl/Tk 8.5 headers. Rather than accept that limitation, experiments are underway using a solution from the VTK codebase. Local copies of various versions of the internal headers are included in the extension's own source tree. The new FindTCL.cmake identifies the system Tcl/Tk version numbers and the correct internal headers are included from the extension's own source tree. This avoids requiring the developer to locate and download source trees that match the installed Tcl/Tk. Use of such local copies runs the risk of crashes if the system Tcl/Tk should happen to have modifications not compatible with the standard headers, but the same problem exists when downloading the Tcl/Tk sources themselves. The only sure solution is to build a local copy of Tcl/Tk as well, which defeats the purpose of using a system Tcl/Tk installation. Including the internal headers does increase the size of the extension source trees somewhat (approximately 2.4 megabytes, uncompressed,) but it is a relatively clean solution to an otherwise thorny configuration management problem.

Longer term, it would be ideal if extensions were no longer required to use non-public APIs to extend Tcl/Tk (or were rewritten to not use them if they don't really *need* to.) Working with the situation as it exists today header inclusion appears to be the most flexible and functional option available.

Extensions currently built with CMake in BRL-CAD include tkhtml, tktable, togl, incrTcl, iwidgets, and tkpng.

Results

Except for the lengthening of Tcl's configure step due to the inclusion of installManPage processing in the CMake configuration, the time needed for configuration and compilation is within ten percent when comparing a TEA based build and a CMake based build. The performance numbers below were generated on a Gentoo Linux machine with an AMD Athlon II X2 245 Processor. All builds are single core (e.g. make with no -j flag).

Operation	TEA (sec)	CMake (sec)
Tcl Configure	6.3	8.4
Tcl Build	48.2	50.5
Tk Configure	2.8	4.0
Tk Build	35.8	38.7
Total Time	93.1	101.6

In addition to matching TEA's compilation performance, CMake has successfully generated working Tcl/Tk build logic on Windows (MSVC), Mac OS X, FreeBSD, Linux, and Solaris (using gcc.) Generators used successfully so far include Visual Studio 8, Visual Studio 10, Unix Makefiles and XCode. There are a number of other possible generators to test, include Eclipse, KDevelop3, NMake Makefiles and MinGW Makefile. Clean integration with BRL-CAD's own logic simplifies cross–platform BRL-CAD development, and the new system has already replaced BRL-CAD's earlier Windows compilation logic in production use.

It is difficult to compare the size and complexity of build systems – the following table reports the line counts for Tcl's autoconf[4], Windows[5] and CMake[6] build systems. This is a raw number (without attempting to filter comments) and it should be noted again that the CMake build does not claim to implement all features of the TEA system.

Autoconf	Windows	CMake
7111	5746	4342

The initial implementation of a working Tcl/Tk build with CMake consumed about 12 man-weeks of effort, although the work was actually performed part-time over the course of one year. Initial efforts used the modified Tcl/Tk 8.5.9 codebase present in BRL-CAD's source tree. Subsequent work has focused on the latest 8.6 beta release. The initial 8.5 to 8.6 conversion of the CMake build system involved a few hours for the initial effort, and a couple of days for subsequent clean-up work in preparation for this paper.

Conclusions and Future Work

The Tcl/Tk CMake build is already the production method of BRL-CAD's Tcl/Tk compilation on Windows, and is being phased in on all other supported platforms. Based on experience accumulated thus far, building Tcl/Tk with CMake represents a fast, effective, low maintenance, and cross–platform solution. It is expected that the new system will reduce BRL-CAD's long term maintenance costs, particularly when it comes to supporting seamless portability to Windows.

The largest remaining task is to finish surveying the TEA build options and identify any tests or settings in the current CMake logic that are inconsistent with Tcl/Tk's Autotools build system. Other remaining items include general clean-up and addition of CPack logic to generate source tarballs, Linux RPM, Mac OS X pkg and Windows NSIS installers. Currently the build does not support running from the build directory when multiple configurations such as those used in MSVC and XCode (Debug, Release, etc.) are present – it may be desirable to generalize existing routines to support such configurations.

The BRL-CAD project will be maintaining and enhancing this new build system as part of its ongoing development, and invites other Tcl/Tk users and developers to build on what has been accomplished to date.

References

[1] BRL-CAD Development Team, BRL-CAD – an Open Source Solid Modeling System, http://brlcad.org

[2] Martin, K. and B. Hoffman, Mastering CMake: A Cross-Platform Build System , Kitware Inc., 2003

[3] Kitware, Inc., CMake - Cross Platform Makefile Generator, http://www.cmake.org

[4] Welch, B. and M. Thomas, "The Tcl Extension Architecture" *7th USENIX Tcl/Tk Conference*, Austin, TX, Feb. 14-18 2000.

[4]In the unix subdirectory – .in files and .m4 files
[5]In the win subdirectory: buildall.vc.bat makefile.bc makefile.vc rules.vc tcl.dsp tcl.dsw Makefile.in configure.in aclocal.m4
[6]Contents of CMake + CMakelists.txt files + FindTCL.cmake

Table 2: Mapping of TEA macros to CMake

SC Macros	TEA Macros	CMake Macros
SC_PATH_TCLCONFIG	TEA_PATH_TCLCONFIG	
SC_PATH_TKCONFIG	TEA_PATH_TKCONFIG	
SC_LOAD_TCLCONFIG	TEA_LOAD_TCLCONFIG	(part of FindTCL.cmake)
SC_LOAD_TKCONFIG	TEA_LOAD_TKCONFIG	(part of FindTCL.cmake)
SC_PROG_TCLSH	TEA_PROG_TCLSH	(part of FindTCL.cmake)
SC_BUILD_TCLSH	TEA_PROG_WISH	(part of FindTCL.cmake)
SC_ENABLE_SHARED	TEA_ENABLE_SHARED	
SC_ENABLE_FRAMEWORK		
SC_ENABLE_THREADS	TEA_ENABLE_THREADS	SC_ENABLE_THREADS
SC_ENABLE_SYMBOLS	TEA_ENABLE_SYMBOLS	
SC_ENABLE_LANGINFO	TEA_ENABLE_LANGINFO	SC_ENABLE_LANGINFO
SC_CONFIG_MANPAGES		
SC_CONFIG_SYSTEM	TEA_CONFIG_SYSTEM	
SC_CONFIG_CFLAGS	TEA_CONFIG_CFLAGS	
SC_SERIAL_PORT	TEA_SERIAL_PORT	SC_SERIAL_PORT
SC_MISSING_POSIX_HEADERS	TEA_MISSING_POSIX_HEADERS	SC_MISSING_POSIX_HEADERS
SC_PATH_X	TEA_PATH_X	(use FindX11.cmake)
	TEA_PATH_UNIX_X	(use FindX11.cmake)
SC_BLOCKING_STYLE	TEA_BLOCKING_STYLE	
SC_TIME_HANDLER	TEA_TIME_HANDLER	SC_TIME_HANDLER
SC_BUGGY_STRTOD	TEA_BUGGY_STRTOD	
SC_TCL_LINK_LIBS	TEA_TCL_LINK_LIBS	SC_TCL_LINK_LIBS
SC_TCL_EARLY_FLAG	TEA_TCL_EARLY_FLAG	
SC_TCL_EARLY_FLAGS	TEA_TCL_EARLY_FLAGS	
SC_TCL_64BIT_FLAGS	TEA_TCL_64BIT_FLAGS	SC_TCL_64BIT_FLAGS
SC_TCL_CFG_ENCODING		SC_TCL_CFG_ENCODING
SC_TCL_CHECK_BROKEN_FUNC		SC_TCL_CHECK_BROKEN_FUNC
SC_TCL_GETHOSTBYADDR_R		SC_TCL_GETHOSTBYADDR_R
SC_TCL_GETHOSTBYNAME_R		SC_TCL_GETHOSTBYNAME_R
SC_TCL_GETPWUID_R		SC_TCL_GETPWUID_R
SC_TCL_GETPWNAM_R		SC_TCL_GETPWNAM_R
SC_TCL_GETGRGID_R		SC_TCL_GETGRGID_R
SC_TCL_GETGRNAM_R		SC_TCL_GETGRNAM_R
SC_TCL_IPV6		SC_TCL_IPV6
	TEA_PREFIX	
	TEA_SETUP_COMPILER_CC	
	TEA_SETUP_COMPILER	
	TEA_MAKE_LIB	
	TEA_LIB_SPEC	
	TEA_PRIVATE_TCL_HEADERS	
	TEA_PUBLIC_TCL_HEADERS	
	TEA_PRIVATE_TK_HEADERS	
	TEA_PUBLIC_TK_HEADERS	
	TEA_PATH_CONFIG	
	TEA_LOAD_CONFIG	
	TEA_LOAD_CONFIG_LIB	
	TEA_EXPORT_CONFIG	
	TEA_PATH_CELIB	
	TEA_INIT	
	TEA_ADD_SOURCES	
	TEA_ADD_STUB_SOURCES	
	TEA_ADD_TCL_SOURCES	
	TEA_ADD_HEADERS	
	TEA_ADD_INCLUDES	
	TEA_ADD_LIBS	
	TEA_ADD_CFLAGS	
	TEA_ADD_CLEANFILES	