VEGA: Vienna Environment for Graphics Applications

Robert F. Tobler, Helwig Löffelmann, Werner Purgathofer[1]

Institute of Computer Graphics, Technical University of Vienna
A-1040 Karlsplatz 13/186/2

Abstract

This paper presents a software development environment for rendering applications. The main parts of this environment are a set of rules concerning the coding style, a set of tools to maintain source files and a number of libraries providing graphics functionality. The environment has been successfully used in a number of internal projects, dealing with rendering.

1 Introduction

VEGA has been introduced to standardize the software development efforts at the Institute of Computer Graphics. The goal of this standardization is to create maintainable software and benefit from the synergistic effects of code and software reuse.

This standardization is only meaningful if the majority of the software produced at the institute conforms to a common set of rules, uses the same tools and uses common libraries and file formats. For this reason a number of conventions and guidelines have been introduced, which should help to attain the standard goals of software engineering.

As a main vehicle of software reuse, a number of libraries have been implemented that provide the core functionality for graphics and rendering applications. The problem of programming Graphical User Interfaces has not been addressed, and standardizing efforts in this direction have not been attempted yet.

The experience gathered in previous projects (RISS [Gerv 88] and VAST [Gerv 93]) has been used, and the problems that have arisen in these projects have been addressed.

2 The Goals of VEGA

With the introduction of VEGA we try to ensure that the software created at the Institute of Computer Graphics meets a number of criteria. These criteria are of course the standard criteria established by software engineering which should be applied to any type of software, but in a university environment additional problems arise that have to be addressed.

---

1 email: rft@cg.tuwien.ac.at, helwig@cg.tuwien.ac.at, wp@cg.tuwien.ac.at

Maintainability:

Over time a huge number of people will develop software using VEGA. A large proportion of the people developing such software will be students. In general, the goal of a student developing a piece of software is normally to write a thesis based on the result of the software, not to produce maintainable code. Therefore the student will use all kinds of tools and software, just to complete the job at hand. For this reason it is not easy to maintain a large collection of software at a university.

A prime goal of VEGA is to make the software written at our Institute somewhat maintainable. We think that any effort in this direction will result in an incresed reuse of the software at our Institute.

Documentation:

Since it is the prime motivation of most programmers at universities to obtain results for some research, the documentation of the programmed software is often very scarce. Therefore a second goal is to introduce conventions on source code documentations, that make it possible to easily produce documentation of modules, libraries and source code.

Portability:

Another major goal of software development should be portability. In a university setting there are typically a number of different platforms in use, and in addition to that a lot of students want to do their software development at home. Therefore one of the goals of the development environment should be an easy way to port software to the major platforms in use.

Software Reuse:

Although mentioned at the end this is actually a kind of meta-goal of VEGA: all goals listed so far have software reuse as their motivation. The goal of VEGA is of course to increase the reuse of all software written at our Institute, and avoid the duplication of efforts among the developers using VEGA.

3 The Parts of VEGA

In order to address all of the stated goals it is necessary to introduce a number of guidelines and tools for software development. The following sections gives an overview of the aspects that are covered by these guidelines and the tools that have been introduced to maintain these guidelines.

3.1 Programming Language

Writing major parts of the software in the same language, simplifies technical details like linking, porting and other related problems. If we had based our decision solely on the portablity of the language, the logical choice would of course be ANSI-C [Ansi 90]. In previous projects we made the experience, that C's protection mechanisms to shield different parts of the implementation from each other, are not adequate. Thus a lot of variables and functions will end up in the global namespace, and special efforts have to modularize the code.

We also based our decision on the current trend towards object oriented programming. The compromise between these two goals is C++ [Stro 91]. Although it is not as portable as C, a certain level of portability can be maintained by avoiding some newer features of the language. The C++ class mechanism makes it possible to implement object oriented designs, and also provide protection between different modules of the code.

3.2 Coding Style

One of the biggest problems in maintaining a huge number of source files, is the readability of the code. In order to increase the readability it is necessary to impose a number of conventions on the coding style. The conventions that we adopted were chosen in such a way as to keep the number of rules that have to be considered, very low. The conventions specify (among other things) the following aspects of the coding style:

- source file headers
- recommended indentation and tabs
- naming conventions for identifiers
- rules for documentation

3.3 VEGA Tools

In order to maintain a huge base of software, it is necessary to use a consistent set of tools that ensure that all sources conform to the guidelines that have been established.

3.3.1 Makefiles

For every sizeable project, it is necessary to create a system that makes it possible to easily recompile parts that have been changed. On UNIX systems the standard way of doing this, is by using the `make` utility.

Maintaining a large number of makefiles which should be portable is not very easy. For this reason a makefile system has been introduced, which makes it possible to create projects similar to those in integrated development environments: it is only necessary to specify all sources and the desired output. The platform dependent parts of all makefiles have been centralized so that they can easily be ported.

3.3.2 Source File Tools

Maintaining all the established guidelines for a huge number of source files requires a set of tools that operate on these sourcefiles. Some of the tools that have been introduced are the following:

Header Tool:

this allows to generate and update the headers of source files that conform to the established coding style. Using a fixed format for file headers makes it possible to introduce tools to extract certain information about files (e.g. author, history, projects, aso.) from these file headers.

Manpage Tool:

this tool generates UNIX man pages from c++ header files. This makes it possible to keep all API documentation of all software in the c++ headers, and reduces the maintainance overhead for documentation.

Renaming Tool:

on some platforms filenames suffer severe limitations in length. Since these restrictions impede the readability of filenames, a renaming tool has been introduced, that allows to map long filenames to short ones and backwards. (the tool knows about c++ include statements and renames them accordingly). Thus it is possible to use the longer and more readable file names on platforms that support it.

3.4 VEGA Libraries

The main vehicles of software reuse at our institute are a number of libraries that can be used by all projects developed under VEGA. These libraries have been especially designed to be portable, consistent and easy to use.

The following short description of the VEGA Libraries is not complete, but it highlights some special features of the libraries that are of interest to the developer of graphics applications.

3.4.1 The VEGA Base Library

The Base Library contains core functionality which defines some common abstractions for all other software. It is not specific to any application area.

ErrorHandlers:

The exception facility of the C++ language has not been implemented on a number of important platforms. For this reason an alternate scheme for reporting errors has been introduced that is based on *error handlers*.

An error handler is an object that error messages and numbers are sent to, so that they can be reported or retrieved. Error handlers can be defined seperately for each library, module or even function, and the behaviour of each error handler can be changed from the default which is to report messages and exit, to record the errornumber for later retrieval.

Tracers:

For debugging code it is often necessary to insert statements into the code, hat report some status. These statements are not intended to stay in the code, but they will be removed after the debugging session. In order to make it possible to leave these statements in the code without suffering a performance penalty, *tracers* have been introduced. Debugging output is now sent to these tracer objects, and each tracer object has a state which decides if the output is displayed or suppressed. These checks are only performed if the code is compiled in a debugging version. In the production version of the software no trace code is left, thus there is no reason to remove the statements that perform the debugging output.

3.4.2 The VEGA Math Library

This Library provides some mathematical core functionality for implementing specialized graphics classes.

Mathematical Constants and simple Functions:

This module provides an extension to the `<math.h>` module of the C standard library. A consistent naming scheme for mathematical constants has been introduced and a number of simple useful functions like `min`, `max`, `abs`, `clamp`, aso. have been added.

Two-, Three- and Four-Dimensional Coordinates:

These types represent arbitrary coordinates which are not restricted to any specific use. Thus various different graphics types can be based on this implementation of coordinates: colors, vectors, points, aso.

Vectors and Matrices of arbitrary Dimensions:

These types represent n-dimensional vectors and n x m-dimensional matrices of single- or double-precision floating point values. The classes have been designed to make it possible to write an interface for code of the Numerical Recipies [Press 92].

3.4.3 The VEGA Graphics Library

The VEGA Graphics Library contains a large number of useful classes for 2D and 3D Computer Graphics. It does not contain any code for specific Graphical User Interfaces.

Two- and Three-Dimensional Vectors and Points:

These types represent the core of the graphics functionality of the VEGA Libraries. They have been designed in a typesafe way, so that only geometrically meaningful operations can be performed (i.e. it is not possible to add two points). There are single-

and double-precision versions of these types, so that it is possible to adapt the memory usage and precision requirements to the application at hand.

Transformation Objects:

Vectors and points by themselves are not sufficient to write serious graphics applications. It is often necessary to transform them in various ways. For this reason a number of transformation classes have been defined. Thus each type of transformation can be represented by a transformation object, and the transformation can be easily and efficiently applied to a set of vectors and points. The available transformation objects are: scaling, special rotations around major axes, general rotations (implemented via quaternions [Shoe 85]), shears along each major plane, linear transformations, translations, and general homogeneous transformations. Operator overloading has been used to introduce all concatenations of transformation objects and all transformations of points and vectors with these transformation objects.

Bidirectional Transformation Objects:

It is often necessary to maintain a transformation and its inverse to convert points and vectors between two coordinate systems. If matrix inversion is used to do back transformations, this could result in severe numerical instabilities [Press 92]. For this reason bidirectional transformation objects have been introduced that maintain both transformations in matrix form. These objects are compatible to all other transformation objects, and therefore it is very easy to build complex transformations and implicitly maintain the inverse tranformation.

Color Classes:

These types represent colors in various different precisions: 8 bit per color RGB with alpha channel (Color32), 16 bit per color RGB with alpha channel (Color64), single- and double-precision floating point RGB. Many operations on the color values have been introduced via operator overloading. Additional color representations like XYZ, Lab, Luv a.s.o will be added in the future.

4 Implementation Issues

During the implementation of the libraries and tools, a number of problems had to be overcome, which had some effect on the interface of the library and the tools provided.

4.1 Performance

The structure of the VEGA Libraries is very modular an thus different parts build on each other. For example the two- and three-dimensional vectors in the Graphics Libary are based on the coordinates introduced in the Math Library. It would seem that this results in a performance penalty for the functionality for vectors and points. This has been avoided by aggressive use of inlining and loop unrolling by hand. Examining the compiled code produced by the GNU C++ Compiler, we found that very little of the perceived overhead remains in the final application.

4.2 Library and Application Sizes

Due to the huge number of functions and inlined functions, the libraries and applications are somewhat large. This problem has been addressed with the use of shared libraries on platforms which support this facility. Therefore it is also recommended to split the functionality of applications into different libraries: this makes it possible to reuse parts of the functionality without any physical overhead.

4.3 Portability

The VEGA libraries have been ported to the following platforms: SGI Irix 5.2 using SGI's CC, MS-DOS using Borland C 3.1, Linux 1.1.52 using GNU g++ 2.5.8 and NeXTstep 3.2 using GNU g++ 2.5.8 and 2.6.1. During these porting efforts it has been found that templates comprise the C++ feature that generates the most portability problems. For makefiles, the GNU version of the `make` tool has been chosen, since it provides some important functionality not available in the standard version of `make` on some platforms.

5 Future Development and Conclusion

The tools and libraries introduced so far cover only the very basic functionality, which is necessary to develop graphics and rendering applications. The VEGA system will be extended by various modules and libraries in order to increase its applicability. The following libraries are being developed, but have not been completed so far:

- Silicon Graphics OpenGL Interface: an interface that hides the OpenGL function based interface, so that VEGA classes can be used.
- Geometry Classes: a number of classes to represent geometries and functions on geometries.
- Image Classes: a number of classes to perform image manipulations.

The core functionality provided in the introduced parts of the libraries have been used for a number of internal rendering projects and have been found to drastically reduce the source code size of these projects. In addition to that, the use of error handlers and tracers has been found to make debugging of code a lot simpler.

6 References

[Ansi 90] X3 Secretariat: *Standard - The C Language*. X3J11/90-013. Computer and Business Equipment Manufactures Association, 311 First Street, NW, Suite 500, Washington, DC 20001, USA

[Gerv 88] M. Gervautz, W. Purgathofer: "RISS - Ein Entwicklungssystem zur Generierung realistischer Bilder" (RISS - A Development System for Genrating Realistic Images), in W. Barth (ed.): *Visualiserungstechniken und Algorithmen*, Informatik Fachberichte 182, p. 61-79, September 1988

[Gerv 93] M. Gervautz, R. Devide: "VAST - An integrated Animation System Based on an Actor - Controller Structure", *Proc. of the 4th EUROGRAPHICS'93 Animation and Simulation Workshop*, Barcelona, September 1993

[Press 92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: *Numerical Recipes in C - The Art Of Scientific Computing*, 2nd edition, Cambridge University Press, 1992

[Shoe 85] K. Shoemake: "Animating Rotation with Quaternion Curves", *Computer Graphics*, 19 (3), pp. 245-254, SIGGRAPH Conference Proceedings, July 1985

[Stro 91] B. Stroustrup: *The C++ Programming Language*, 2nd edition, Addison Wesley, 1991