ViennaData 1.0.0

User Manual

Institute for Microelectronics
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria

Main Contributors:

Karl Rupp

Current Maintainers:

Karl Rupp

Institute for Microelectronics
Vienna University of Technology
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria/Europe

Phone	+43-1-58801-36001
FAX	+43-1-58801-36099
Web	http://www.iue.tuwien.ac.at

Contents

Introduction

When designing classes in object-oriented programming languages, one has to distinguish between intrinsic properties that are inherent to the modelled entity, and data that is associated with the entity. For example, a triangle is intrinsically defined by three disjoint points, while associated data can be color or texture information in graphical applications, or material properties for two-dimensional scientific simulations.

While associated data depends on the respective application, intrinsic properties do not. Therefore, when designing reusable classes, associated data must not be part of the class itself. Reconsidering the example of a triangle, a class member for color information would immediately result in a potentially significant overhead for scientific simulations and thus reduce the reuseability of the class. However, for graphical applications the color information for the triangle still has to be accessible. This is where `ViennaData` comes into play, because it provides a convenient means to store and access associated data for small, reusable classes with intrinsic data members only.

As an example, suppose there exists a minimalistic class for triangles called `Triangle`, which provides access to its vertices. For objects of this triangle class, color information modelled by a class `RGBColor` should be stored within some application. The following lines accomplish that for a triangle object `tri` using `ViennaData`:

```
//store RGB color information:
viennadata::access<string, RGBColor>("color")(tri) = RGBColor(255,42,0);

//get color information at some other point in the application:
RGBColor col = viennadata::access<string, Color>("color")(tri);
```

In `ViennaData` data is associated with objects by keys of user-defined type. In the example above, the color information is stored using a key `"color"` of type `std::string`. However, the user is free to use any key of any type[1]. Thus, any associated data object `data` of a generic type `DataType` can be stored and accessed for an object of any type using a key `key` of type `KeyType` by (cf. Chapter 2)

```
//store 'data' for obj
viennadata::access<KeyType, DataType>(key)(obj) = data;

//retrieve the stored data:
DataType data2 = viennadata::access<KeyType, DataType>(key)(obj);
```

Therefore, `ViennaData` allows to design and work with classes that focus on their intrinsic properties only, and ensures uniformity of associated data access over an arbitrarily large set of objects of different type. This is in particular important if several different libraries with different coding styles are used within a project.

[1] Some technical restrictions apply, see Chapter [MISSING].

With little additional configuration (cf. Chapter 3), the code for color information access using ViennaData can be reduced to

```
//store RGB color information
viennadata::access<color_info>()(tri) = RGBColor(255, 42, 0);

//get color information at some other point in the application:
RGBColor col = viennadata::access<color_info>()(tri);
```

This allows uniform, type-safe data access throughout the whole application. As shown in Chapter 4, there is virtually no abstraction penalty for this gain in flexibility and uniformity. Explicit use cases are given in Chapter 5.

Internally, ViennaData stores data in tree-like structures, for which the object address is used as key for storing the data. However, if objects provide a means to identify themselves by numbers in the range $1, \ldots, N$, the internal datastructure of ViennaData can be conveniently customized to a more appropriate storage scheme. Similarly, if it is a-priori known that for a particular key type only one key object is used, the internal datastructure of ViennaData can be adapted to that. Chapter 6 explains the internals.

Finally, a few words shall be spent on the pitfalls of ViennaData. Since data is associated with objects, object lifetimes have to be kept in mind. If an object, for which associated data is stored, reaches the end of its lifetime, the user has to care for the associated data separately. This can be very simple and fully transparent for the user in some cases, and rather tough in other cases. More details are given in Chapter 7.

Chapter 1

Installation

This chapter shows how ViennaData can be integrated into a project and how the examples are built. The necessary steps are outlined for several different platforms, but we could not check every possible combination of hardware, operating system and compiler. If you experience any trouble, please write to the maining list at

viennadata-support@lists.sourceforge.net

1.1 Dependencies

- A recent C++ compiler (e.g. GCC version 4.2.x or above and Visual C++ 2008 are known to work)
- CMake [1] as build system (optional, but highly recommended for building the examples)

1.2 Generic Installation of ViennaData

Since ViennaData is a header-only library, it is sufficient to copy the viennadata/ folder either into your project folder or to your global system include path. On Unix based systems, this is often /usr/include/ or /usr/local/include/.

On Windows, the situation strongly depends on your development environment. We advise users to consult the documentation of their compiler on how to set the include path correctly. With Visual Studio this is usually something like C:\Program Files\Microsoft Visual Studio 9.0\VC\include and can be set in Tools -> Options -> Projects and Solutions -> VC++-Directories.

1.3 Building the Examples and Tutorials

For building the examples, we suppose that CMake is properly set up on your system. The other dependencies are listed in Tab. 1.1.

Tutorial No.	Dependencies
tutorial/tut1.cpp	OpenCL
tutorial/tut2.cpp	OpenCL, ublas
tutorial/tut3.cpp	OpenCL, ublas
tutorial/tut4.cpp	ublas
tutorial/tut5.cpp	OpenCL
benchmarks/vector.cpp	OpenCL
benchmarks/sparse.cpp	OpenCL, ublas
benchmarks/solver.cpp	OpenCL, ublas

Table 1.1: Dependencies for the examples in the examples/ folder

1.3.1 Linux

To build the examples, open a terminal and change to:

```
$> cd /your-ViennaData-path/build/
```

Execute

```
$> cmake ..
```

to obtain a Makefile and type

```
$> make
```

to build the examples. If desired, you can build each example separately instead:

```
$> make blas1              #builds the blas level 1 tutorial
$> make vectorbench        #builds vector benchmarks
```

Speed up the building process by using jobs, e.g. make -j4.

1.3.2 Mac OS X

The tools mentioned in Section 1.1 are available on Macintosh platforms too. For the GCC compiler the Xcode [2] package has to be installed. To install CMake, external portation tools such as Fink [3], DarwinPorts [4] or MacPorts [5] have to be used.

The build process of ViennaData is similar to Linux.

1.3.3 Windows

In the following the procedure is outlined for Visual Studio: Assuming that an OpenCL SDK and CMake is already installed, Visual Studio solution and project files can be created using CMake:

- Open the CMake GUI.
- Set the ViennaData base directory as source directory.

- Set the `build/` directory as build directory.
- Click on 'Configure' and select the appropriate generator (e.g. `Visual Studio 9 2008`)
- Click on 'Generate' (you may need to click on 'Configure' one more time before you can click on 'Generate')
- The project files can now be found in the `ViennaData` build directory, where they can be opened and compiled with Visual Studio (provided that the include and library paths are set correctly, see Sec. 1.2).

The examples and tutorials should be executed from within the `build/` directory of `ViennaData`, otherwise the sample data files cannot be found.

Chapter 2

Basic Usage

In this Section the fundamental operations in ViennaData are covered in full detail. All functionality is available out of the box. If `ViennaData` access turns out to be performance critical, ample optimizations can be triggered, which is discussed in Chapter 3.

2.1 Access

The main function in `ViennaData` is the `access()` function. From a user perspective, the call

```
viennadata::access<KeyType, ValueType>(key)(obj)
```

returns a reference to the data of type `ValueType` stored for an arbitrary object `obj` and identified by the `key` object of type `KeyType`. The rest of this section is devoted to the various possible variations.

As an example, to store RGB color information on objects `obj1`, `obj2` and `obj3` using a `std::string`, say `"color"`, as key, the code

```
viennadata::access<string, RGBColor>("color")(obj1) = RGBColor(255,42,0);
viennadata::access<string, RGBColor>("color")(obj2) = RGBColor(0,42,255);
viennadata::access<string, RGBColor>("color")(obj3) = RGBColor(10,20,33);
```

Note that `obj1`, `obj2`, and `obj3` can be different type. The data can be recovered directly as

```
RGBColor col1 = viennadata::access<string, RGBColor>("color")(obj1);
RGBColor col2 = viennadata::access<string, RGBColor>("color")(obj2);
RGBColor col3 = viennadata::access<string, RGBColor>("color")(obj3);
```

In addition, several different colors can be associated with a single object as

```
viennadata::access<string, RGBColor>("front")(obj1) = RGBColor(255,42,0);
viennadata::access<string, RGBColor>("back")(obj1) = RGBColor(12,21,255);
viennadata::access<string, RGBColor>("side")(obj1) = RGBColor(30,20,10);
```

and similarly for `obj2` and `obj3` if desired. Since any type that fulfills the `LessThanComparable` concept [CITE STL] (i.e. the `operator<` can be used for comparisons) can be used as key type, one can also write

```
viennadata::access<string, RGBColor>("front")(obj1) = RGBColor(255,42,0);
```

```
viennadata::access<long,   RGBColor>( 12345 )(obj1) = RGBColor(12,21,255);
viennadata::access<double, RGBColor>( 42.42 )(obj1) = RGBColor(30,20,10);
```

If color is to be saved in different color spaces, this could be accomplished by

```
viennadata::access<string, RGBColor>("front")(obj1) = RGBColor(255,42,0);
viennadata::access<long,   HSVColor>( 12345 )(obj1) = HSVColor(12,21,255);
viennadata::access<double, CMYKColor>(42.42 )(obj1) = CMYKColor(0,1,2,3);
```

At this point you may think that the second template argument is somewhat redundant. In that case, please read about the design decision in Chapter **??**.

In the last examples, only one key object per key type has been used. As will be shown in Chapter 3, one can completly turn off run-time comparisions of keys, so that one can write

```
viennadata::access<front_tag, RGBColor>()(obj1) = RGBColor(255,42,0);
viennadata::access<back_tag,  HSVColor>()(obj1) = HSVColor(12,21,255);
viennadata::access<side_tag, CMYKColor>()(obj1) = CMYKColor(0,1,2,3);
```

where `front_tag`, `back_tag` and `side_tag` are empty tag classes [CITE] that are solely used as a key type. Again, the stored data can be retained directly by

```
RGBColor  col1 = viennadata::access<front_tag, RGBColor>()(obj1);
HSVColor  col2 = viennadata::access<back_tag,  HSVColor>()(obj1);
CMYKColor col3 = viennadata::access<side_tag, CMYKColor>()(obj1);
```

2.2 Copy

viennadata::copy()

2.3 Move

viennadata::move()

2.4 Erase

viennadata::erase()

2.5 Query Data Availability

viennadata::find()

Chapter 3

Advanced Usage

Explain a few advanced concepts.

3.1 Retrieving IDs from Objects

viennadata::dispatch_traits

3.2 Register Data

viennadata::register

3.3 Dense Data

viennadata::reserve viennadata::storage_traits

3.4 Default Data Types for Keys

viennadata::default_data_type

3.5 Type-Based Keys

viennadata::dispatch_traits

Chapter 4

Benchmark Results

4.1 ViennaData Storage Models

compare access times for sparse storage and dense storage, including various dispatches

4.2 ViennaData vs. Traditional OOP

Compare ViennaData with traditional OOP concepts where data is injected as class members

Chapter 5

Use Cases

In this chapter we discuss two use-cases for `ViennaData`. The two use-cases are intentionally chosen to be very distinct from each other. The first use-case is admittedly artificial, since the author of `ViennaData` has not been involved in programming computer games yet. The second use-case describes the solution of partial differential equations by means of discretization methods. Since the author is actively working on the simulation of semiconductor devices, this can be seen as the roots of `ViennaData`.

5.1 Human-Like Characters in Online Games

5.2 Solution of Partial Differential Equations

Chapter 6

Library Internals

What is going on internally.

6.1 Classes

Main classes and their interaction

6.2 viennadata::all

Explain effort required for making viennadata::all work.

6.3 Programming Techniques

A few words here

Chapter 7

Pitfalls

7.1 Object Lifetimes

Object gone, data gone...

7.2 Key-Data Registration

If you forget to register your data, you may not be able to find it again...

Change Logs

Version 1.0.0

First release

License

Bibliography

[1] “CMake.” [Online]. Available: http://www.cmake.org/

[2] “Xcode Developer Tools.” [Online]. Available: http://developer.apple.com/technologies/tools/xcode.html

[3] “Fink.” [Online]. Available: http://www.finkproject.org/

[4] “DarwinPorts.” [Online]. Available: http://darwinports.com/

[5] “MacPorts.” [Online]. Available: http://www.macports.org/