AnimaLand Documentation

Tommi Ilmonen

March 17, 2004

Contents

Chapter 1

Main Software Components

AnimaLand is based on a number of components. Many software components are divided into multiple libraries.

1.1 VEE – Visual Effects Engine

VEE is a second order particle system engine. It is used to calculate particle systems and render them. The system has been built for fairly high performance, good stability and reasonable flexibility – a mixture suitable for real-time systems.

VEE has a collection of diffeerent particle types and force types. It also includes a full OpenGL renderer and a minimal Renderman renderer (RIB generator).

VEE takes the vector and matrix classes from Fluid 2.4. These are renamed to VEE_Vector3, VEE_Matrix3 etc. to make it possible to migrate to different vector library at some stage. These renamed vectors and matrices are used directly also in GEE.

VEE is divided into several sub-libraries with the following functionality:

- base – This is a large library that contains plenty of particles and force classes
- opengl – Handles the rendering of particles with OpenGL
- effects – Some predifined particle effects. These are largely outdated or really not needed
- extensions – Some non-trivial classes that depend on some libraries that depend on the “base”
- Qt – Implements a Qt widget set that can be used to visualize VEE and GEE on a desktop using OpenGL. Depends on the Qt class library by Troll Tech. Also introduces the image io-plugins needed by the opengl-library.
- test – A simple test application that can run a few simple test particle systems. Largely obsoleted by the python stuff, but useful for basic troubleshooting.

- python – Wraps most of code in VEE, GEE and Solar into Python. Uses SWIG to generate the wrapper files.

1.2 GEE – The Geometry Engine

GEE is the component that is responsible for doing bulk of the modeling and animation work. GEE uses VEE to handle particles and also borrows primitive classes from VEE (bounding boxes, color classes etc.).

GEE is basically a modeling/animation engine. It includes basic graphical primitives and tools (operators) to craft them. The most important primitives are:

- 3D path – hand/tool motion path (GEE_Path3D)
- Triangle mesh – An ordinary triangle mesh with an array of vertices and triangle indices (GEE_TriangleMesh)
- Lines – A collection of line segments (GEE_Lines3D)
- Particle clouds – A collection of stationary particles (GEE_ParticleObject)

Some important operators are:

- Path to triangle mesh conversion (GEE_ActionPathToMesh)
- Path to lines conversion (GEE_ActionPathToLines)
- Path to particle cloud conversion (GEE_ActionPathToParticles)
- Erase primitives around a path (GEE_EraseAroundPath)
- Move vertices/particles around a path (GEE_MoveAroundPath)
- Recolor vertices/particles around a path (GEE_ColorAroundPath)

GEE is organized into two libraries:

- gee – all of the functionality, but no rendering
- opengl – OpenGL rendering of the graphics

GEE is distributed along with VEE in the same source package.

1.3 AnimaLand – The Real Application

Animaland is the component that takes a bunch of low-level libraries and turns them into a 3D application. The components of Animaland are:

- Crystal::Engine – Manages input devices and animation.
- VRJuggler-app – A VR application that is used by the artists to create art.
- Qt-app – A Qt-based host (GUI) that can be used to run the engine in a normal window. Useful for debugging and viewing what happened.

1.3.1 Crystal::Engine

Engine is the largest part of AnimaLand. It runs animations with GEE and creates new operators into GEE from the user input.

1.3.2 VRJuggler-app

VRJuggler app runs the engine as a VRJuggler application. This is a fairly thin wrapper that uses VRJuggler to manage the wall projection and OpenGL context switches.

1.3.3 Qt-app

A Qt application was developed so that one could test the system in a normal desktop environment. The Qt app is typically used with pre-recorder user data to replay the session. One can then take screenshots or movies of the animations.

This app is also useful so one can run simple tests with real user data wihtout using the full VR application. A single-threaded Qt app is much easier to debug than a full VR application. It can also be run under Linux with valgrind to check memory leaks etc.

Chapter 2

External Components

2.1 C++ & STL

C++ is our main programming language. Another one is Python (section 2.7)

STL (Standard Template Library) containers are used all over the place. In particular map, vector and list classes are often used.

2.2 OpenGL

Pure OpenGL is used to render the graphics in real-time.

2.3 Renderman

VEE includes code to render some objects (simple particles and mesh particles) with Renderman. This support is incomplete compared to the OpenGl code. GEE lacks Renderman support. It is possible to create it when needed.

2.4 Fluid – Flexible Input Design

Fluid is library for managing novel input devices such as data gloves and motion trackers. It includes code that reads the raw data from those devices (basic input) and code that refines the data (data processing / gesture recognition).

In AnimaLand Fluid is used to handle the input devices and do some basic processing on the data.

Fluid also gives the motion data to the VRJuggler so that VRJuggler can handle the wall projections correctly.

2.5 ConfigReader

This is a small library that can read basic configuration information from a file (and also save itself to a file). The file format is extremely simple: Data is organized into named chunks. Inside chunks are named variables that contain some value. All data is represented in human-readable ASCII form.

A simple example file with two chunks follows. This is an example of a file that defines the textures used for the particle systems.

```
Store {
  texturefile = textures.png
}

Texture-0 {
  bitmap = images/noise.jpg
  decaypower = 0.8
}
```

ConfigReaded is distributed in the same source package with VEE.

2.6 Qt

Qt is a GUI-library for UNIX, Windows and Mac OS-X. It contains a widget library and plenty of utility classes that are useful across platforms (XML, database access, time & date management, threading).

VEE and GEE have soft dependencies on Qt. This means that Qt is not a critical part of these components and Qt-related code could be removed without massive rewrites.

VEE has a test bench based on Qt and AnimaLand has a playback engine based on Qt.

For more information on Qt see www.trolltech.com.

2.7 Python

Python is an object-oriented scripting language. It is used to test and prototype VEE and GEE components. One can use the VEE and GEE components in a Python shell and write small scripts that run some test or demonstration.

Python is useful over C++, since one does not need to compile or link code to run tests with slightly different parameters. There has been some consideration that Python would overtake more of the functions of C++, but at the Python is simply a semi-interactive debugging test environment.

The C++-to-Python bindings are generated automatically with SWIG (see section 2.8).

For more information on Python see www.python.org.

2.8 SWIG - Simplified Wrapper Interface Generator

SWIG is a tool that wraps C and C++ to a number of target languages. Without SWIG we would need to do a lot of work to make C++ classes available to Python interpreter. Instead of such laborious work we give the C++ class definitions to SWIG that creates a huge amount of C++ code that wraps VEE and GEE classes to the Python interpreter. The code is then compiled into a Python module (a shared library) that we can load into the Python interpreter.

For more information on SWIG see www.vrjuggler.org.

2.9 VRJuggler

VRJuggler is a large VR-framework. In our work we only use VRJuggler to manage the wall projections and to manage OpenGL contexts and windows in AnimaLand.

For more information on VRJuggler see www.swig.org.

2.10 WML – The Wild Magic Library

Wild Magic Library (WML) is a C++ class library by David Eberly. It has code for 2D and 3D graphics and geometry handling and a full game engine. VEE and GEE borrow some algorithms from WML, for example spline and intersection calculations. A minimal version of WML is included in the VEE source distribution so one does not need to load the library separately when using VEE or GEE.

Since WML is developed on Linux, Windows and Mac it does not always compile out-of-the-box on IRIX. Typically some minimal changes need to be made to make it compile and run properly on IRIX.

We often need to convert vectors and matrices between the Fluid and WML classes. Since the classes are equivalent we do this with brutal cast operation. For example:

```
VEE\_Vector3 v(1, 2, 3); // VEE\_Vector3 = Fluid::Vector3T<float>
Wml::Vector3<float> wv;
wv = * (Wml::Vector3<float> *) &v;
```

In general we try to hide the use WML and limit its visibility into just the code really needs WML. This is done to minimize the dependency on WML. WML is only used when it offers algorithms that we sorely need, but do not want to write ourselves.

For more information on WML see www.magic-software.com.

2.11 Opcode – Optimized Collision Detection

Opcode is a C++ library for detecting collisions between polygon meshes and other kinds of geometry. It is included in VEE since we needed an alternative collision detection system for VEE (besides the plane-grid method).

Opcode is hardly very clean C++, but it works. Some hours of work is needed to make it compile on IRIX, since the code is such a mess of defines, namespaces and strange include ordering. Once compiled it is stable and fast.

For more information on Opcode see www.codercorner.com/Opcode.htm.

Chapter 3

Some Remarks

3.1 Installation & Compilation

It takes some time, effort and UNIX environment variables to compile AnimaLand. The basic procedure is outlined below. It is typically best to take the soources from CVS repository rather than from the net, since CVS has the newest code and is most likely to compile.

During the installation all kinds of environment variables are needed. A wise person puts the requireed environment variables into his/her shell initializations one does not have to set them at each session.

1. Install Mustajuuri. Get the sources from CVS, configure and compile as instructed in Mustajuuri homepage.

2. Install Fluid. Get the sources from CVS, configure and compile as instructed in Fluid homepage (or somewhere).

3. Install VEE. Get the sources from CVS, configure and compile as instructed in VEE homepage. VEE includes GEE and all sorts of third-party stuff that is automatically compiled as well.

4. Install AnimaLand (Crystal). Get the sources from CVS and compile. No configuration is necessary.

3.2 Thread Safety

GEE is not thread in the general sense. The data traversal is thread-safe how-ever. This is done to make is possible to render the scene with multiple threads at the same time.

The rendering threads can work in parallel, but no other action may take place during this time. In particular one cannot do anything to the animation while rendering is in progress.

3.3 Common Utility Classes

3.3.1 template class VEE_ReferenceObject

This class implements a pointer-sharing approach to managing objects. For example VEE_ReferenceObject¡float¿ object can be copied and all copies will point to the same actual variable.

This class is used widely in VEE to minimize memory allocations when some potentially large object is shared between two higher-level objects.

3.3.2 template class VEE_RefPtr

This class implements a pointer-sharing approach to managing pointers. This class is useful when an object of abstract base needs to shared between several higher-level objects. The pointer will be deleted when the last link to it is broken.

This class is used widely in GEE to make sure data in world and action manager is deleted at the appropriate moment (and not before).

3.3.3 template class VEE_ClonablePointer

This class is used to spread objects of abstract type across the system. The template type must implement “clone”-method that replicates the object. Whenever a copy is made the object is then cloned. This cloning approach is useful when we need to copy objects without knowing the exact type of the objects.

This class is used widely in VEE to copy operators.

Chapter 4

GEE File Format

This chapter describes how the GEE file format is built. The exact details of the file format are not documented here, but rather the approach used. The exact file format can be extrapolated by looking at the source files :-)

GEE uses a chunk-based binary format to store its data. Each object compound typically saves itself to a separate chunk in the file. It is hoped that such approach would separate errors in file parsing.

Each object its own data format that it can read and write. The hosting system does not really care about how some object stores itself.

File byte-order is little endian.

4.1 Common file-format properties for all kinds of objects

The file is organized into chunks. The chunks begin with an identifier. Identifier is followed by 32-bit version number. The version number is specific to each object type and useful when building backwards-compatible readers. After the version number comes the actual data in binary format.

4.2 Typical file format header for main-level objects

Main-level objects are objects that inherit GEE_Data or GEE_Action.

These objects typically print the C++ class name into the beginning of the file as a normal NULL-terminated string.

4.3 Typical file format header for low-level objects

Low-level objects are smaller that main-level objects. Often a main-level object contains a number of low-level objects. Vertices and triangle indices are typical low-level

objects.

These objects typically use a specified 32-bit integer to check that the data following in the stream really represents the object. Usually the objects build the 32-bit integer out of four characters that somehow represent the type. For example GEE_PolygonVertex uses “PGVX” as an id. The ids should (but do not really have to) be unique for each class. Usually these ids are constructed with GEE_CONST macro that can be found in header “gee_macros.h”.

Typical abbreviations are:

- DT - Data
- IN - Index
- PG - Polygon
- PR - Particle
- TR - Triangle
- VX - Vertex

4.4 Tools for reading & writing files

GEE_File is the most common stream class for writing files. It is declared in file “gee_io.h”. It can write most common data types (including vectors and matrices) to the file. It uses highly specific function names to guarantee that the user really knows what is the data type he/she is about to write into the file.