Helma Technical Overview

Tommi Ilmonen
Markku Reunanen

December 1, 2004

Contents

Chapter 1

Introduction

Figure 1.1: An artist is building a particle cloud.

Helma is a research project to create an immersive 3D drawing/sculpting/animation software. The basic aim is to enable intuitive artistic free-hand work in a virtual reality (VR) environment. The research hypothesis is that such a tool would enable new kinds of digital artistic expression. Figure 1.1 shows the Helma system in use — an artist is building graphics in a virtual room.

The Helma project merges the Crystal project by Tommi Ilmonen (section 11.19) and Antrum (by a student group in Helsinki University of Technology, see section 11.20). Helma is funded by the National Technology Agency of Finland — Tekes — and several companies: Valkeus Interactive Oy, Metso Paper Oy, Numerola Oy and the Finnish Science Center Heureka.

1.1 The Helma People

Helma is made primarily in Helsinki University of Technology (HUT) Telecommunications Software and Multimedia Laboratory (TML). Other organizations are Tampere University of Technology (TUT) Department of Electronics and HUT Automation Technology Laboratory. The people working with Helma are:

- Artist Wille Mäkelä is the project leader. Wille makes art, leads the project, specifies and tests the user interface.
- M.Sc. Tommi Ilmonen is the technical leader and a programmer in the project. Tommi is also the main author of this document. Tommi is responsible for geometry and animation management and the main author of several support libraries (Fluid, VEE, Mustajuuri, DIVA, AC).
- M.Sc. Markku Reunanen is another programmer. Markku is responsible for the upcoming new WIMP and hardware-related problem solving (Näprä, exhibition setups).
- Professor Karri Palovuori at the TUT Department of Electronics is responsible for finger tip tracking (hardware and software) of the Näprä.
- The physical construction of the Näprä was designed and built by people in the HUT Automation Technology Laboratory.

Of these people Wille, Tommi and Markku can be considered to the central Helma-team and others contribute their special skills for some limited part of the system.

1.2 About this Document

This document describes the technical foundations of the Helma system. It is written so that information about the internals of the system would be available to people joining the project or wishing to know what is happening in the project. For this purpose we briefly illuminate each component in the Helma architecture without digging too deep into any particular topic. The reader is expected to have a solid technical background — otherwise the wisdom of this friendly document may well be difficult to comprehend.

This document avoids forward-looking statements — it is a compact description of the system as it is. A thorough description of the system would be ten times longer. For background information, development ideas, future roadmap or scientific view of the results, you should consult some other information source (e.g. conference papers, Helma personnel). Some expected near-term technical changes are described in chapter 13. Throughout this text we try to avoid all kinds of humor, except where it is the only way to describe reality accurately.

The reader is encouraged to give feedback about the usability of this document, any inconsistencies and typos.

Chapter 2

Hardware

Helma needs a collection of exotic hardware to run. This chapter gives overview of devices that are needed to run Helma. Figure 2.1 Gives a schematic overview of Helma hardware components.

At the moment three computers are needed to run helma system. One is the motion tracker computer (an industrial PC with special tracker boards), another is the Linux PC that processes sound and handles Näprä input and third is the graphics computer that runs the application and does the graphics rendering.

2.1 EVE

We run Helma in the EVE virtual room in HUT/TML (figure 2.2). This is a normal back-projected VR system with four walls and 14 loudspeakers.

2.2 Motion Tracker

We use off-the-shelf commercial motion tracker by Ascension. Any tracker is ok as long as we can make it work properly. A reasonable tracker is one that gives us the location and rotation of each motion sensor in same decent sample rate (at least 20 Hz).

2.3 Näprä

In Helma the motion of fingers is tracked with Näprä (figure 2.3). Näprä is a combination of finger tracking and triggering hardware. Finger tips are tracked with ultrasonic technology. The thumb, index- and middle finger have metal caps and they provide us with contact information.

Beneath the two remaining fingers are buttons for additional triggering input. We use identical Näpräs in both hands.

2.4 Graphics Computer

To run Helma you need a computer that handles the graphics and main application (see figure 2.1). This computer should have miraculous processing and graphics resources. The machine must be able to output 3D graphics to multiple projectors.

Currently we use an SGI Onyx2 computer for this purpose. In the future we aim to use a Linux cluster in its place (with master/slaves -architecture).

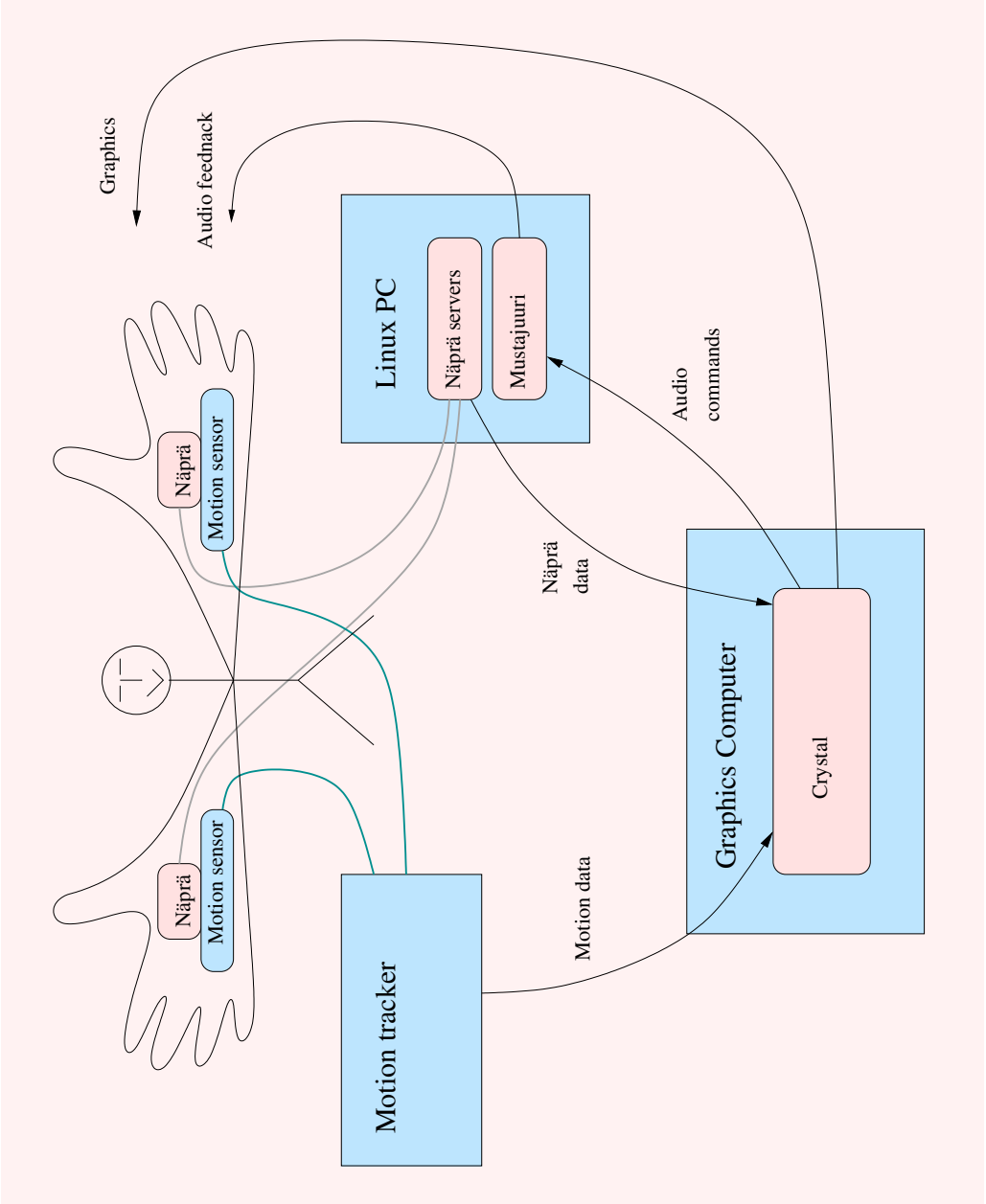


Figure 2.1: Helma hardware setup. The computers communicate via Ethernet.

2.5 Peripheral Linux PC

In addition to the main graphics computer we use a Linux PC that handles exotic input devices (data gloves, Näprä) and produces audio output. The input devices are managed with Fluid servers (section 6.1.5). The audio output is calculated with Mustajuuri (section 6.1.2).

The devices are not connected directly to the main computer since we want to isolate the expensive main system against electrical hazards. Additionally some of the programming work is easier to do on a commodity Linux system rather than on a rare SGI hardware with buggy SGI

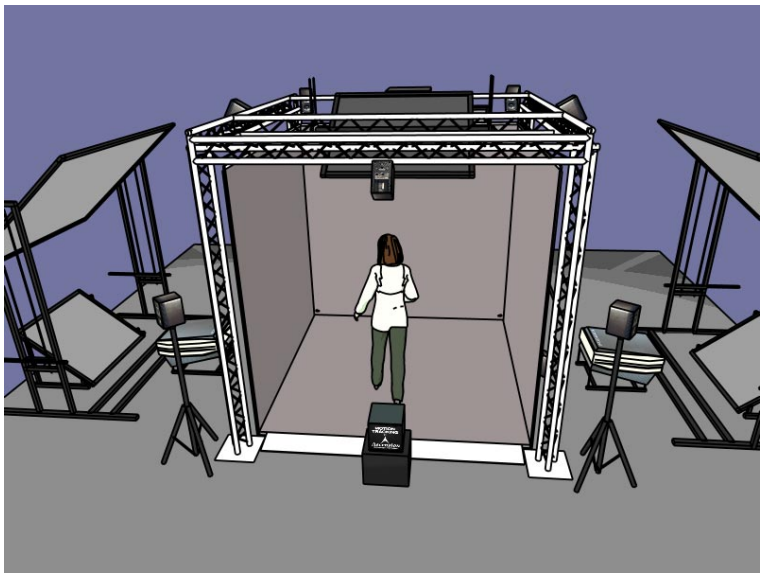


Figure 2.2: Experimental Virtual Environment — EVE.

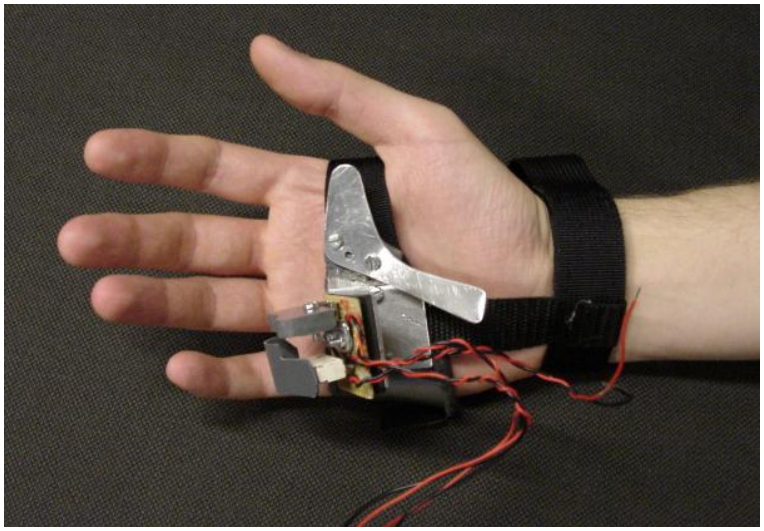


Figure 2.3: Näprä prototype. There are (will be) microphones at finger-tips.

software.

Chapter 3

General Application Structure

Helma is a modular component-based application. Its basic structure is shown in figure 3.1. The following sections outline what all of these components are and how they co-operate.

3.1 Fundamental Concepts

In this section we cover the most fundamental concepts that are needed to understand how Helma works.

These concepts are implemented in the GEE library; see section 5.2 for more thorough explanation of the actual implementation.

3.1.1 Geometry

The 3D objects that are in world are called geometry. All geometry types are derived from a common base class — GEE_Geometry. All things that are visble are called geometry. Typical examples are triangle meshes, polylines and particle clouds. Geometry objects are always made up of smaller primitives — vertices, triangles, particles etc.

While all visible things are geometry, geometry can also be invisible. Hidden geometry is a nice way to maintain common data in the world (=path information).

Geometry objects have little internal logic — they represent the state of the world, but do not alter anything voluntarily.

The geometrical primitives are further explained in chapter 8.

3.1.2 World

All geometry objects reside in a world. The world is simply a geometry pool that keeps track of all current objects. Besides keeping the geometry the world stores miscellanous information that is without a proper home: texture name tables and the scene background color.

When one needs to find a piece of geometry, then one can query the world with the proper object id. This method is much more safe than having ordinary C/C++ pointers that look into the world.

3.1.3 Operator

All changes in the world are caused by operators. An operator has the capability to alter some geometry in the world. An operator may create, modify or delete geometry objects.

Operators have some duration — a beginning time and an end time. Between these times (limits points inclusive) the operator is active and does something in the world.

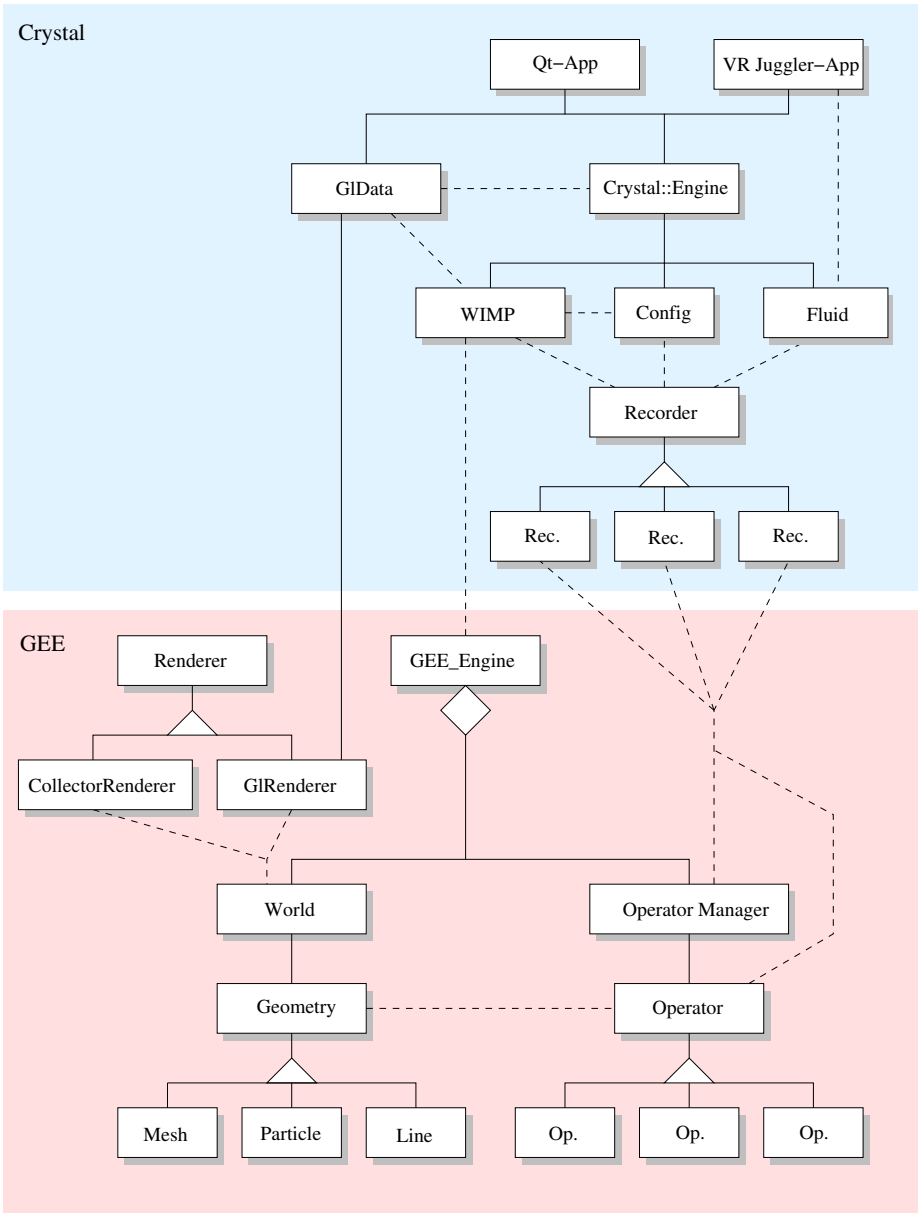


Figure 3.1: Main components of Helma.

3.1.4 Operator Manager

Operator manager keeps track of all operators. It knows how to run operators from frame to frame, how to do undo and redo and how to read/write operators to a file.

3.1.5 AU-Coordinates

In the system there are two kinds of time — animation time (counted in frames) and user time (the real time that the user is experiencing). In figure 3.2 these two times are shown. Operators can freely overlap in the animation time, but they cannot overlap in the user time. In animation

time the operators are sorted by their beginning time. In user time indexing is used, each operator gets a unique index number. The opetators are evaluated from beginning to the end (animation time) and from older to newer (user time).

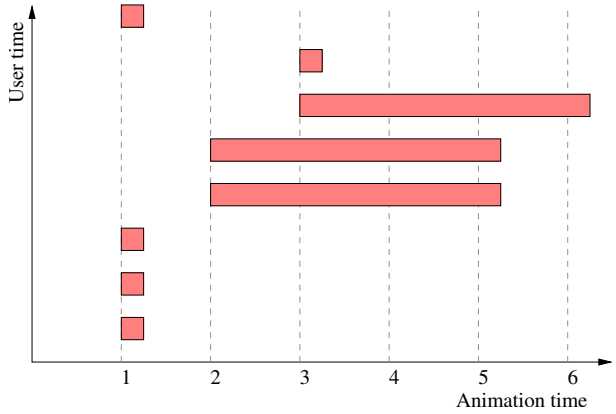


Figure 3.2: Animation/User (AU) time coordinates. First the user has made three operators to frame 1, then two longer operators that span from frame 2 to frame 5, then an operator that spans frames 3-6 and finally two operators to frames 3 and 1. The animation time is expressed in frames while the user time is quantized into operator count. Each operator has some unique location in the operator stack. There can be any amount of time between the creation of operators.

Once we sort operators into AU-coordinates we can do number of things: Undo and redo become trivial — we only need to pop and push operators in the user time stack. We can do any operation at any time — create objects, modify objects etc. Operators can be moved in the animation time. The operators can be streched in the animation time.

This approach enables us to use same operators for both instantenous work (a large amount of geometry comes up at a one frame) and time-based work (the creation of the model becomes animated). This means also means that all operations that one can perform on the geometry can be animated. This maximum animatability was desired since this enables some new ways for an artist to work: Animations can be made one frame at a time with the previous frame as a starting point for the next frame.

There are some potentially negative side effects from this approach to the animation: 1) Animation requires processing — to get from frame x to frame x+1 may take a lot of computational resources. 2) There are no explicit models — we only have a world with some geometry but we do not really have 3D objects in the traditional sense (altought geometry groups an be thought of as objects, section 3.1.9).

The AU coordinates are not explicitly expressed in any C++ class – they are an implicit part of the the general architectrure.

3.1.6 Persistence of Geometry

In Helma geometry is persistent: As animation time ticks forward geometry stays unchanged unless it is explicitly changed. This is called inheritance (or persistence) — frame n inherits all geometry from frame n-1 unless some operator comes along and makes changes into the geometry.

When we rewind the state of the world to the beginning, then all geometry in the world is deleted.

3.1.7 Geometry Generations

Each geometry has a generation (or version) number. This number is incremented whenever the geometry is modified. This information is used in the OpenGL renderer that creates new display lists for the geometry as needed.

A user should not be aware of the generation information in any way.

3.1.8 Layers

When the user is modifying existing geometry (e.g. changing color) it is typical that the operation should not affect all geometry in the world, but rather just some parts. It is not possible to do spatial separation between intended and unwanted geometry (since geometries may overlap).

To make it possible to craft objects in this selective way we have layers. Each geometry object and operator belongs to some layer. In practice a layer is a mini-world that is independent of other layers.

Layers are not explicitly expressed in any particular class. Rather all operators and geometries have knowledge of their layer and can act accordingly.

3.1.9 Geometry Groups

The geometry in the world can be grouped. If for example someone has made geometry objects that need to be moved together, then this can be accomplished by grouping the geometries together and moving the group as a whole.

This reflects the grouping behavior found in most 2D/3D vector-drawing applications.

3.1.10 Operator Groups

Typically it is necessary that we change the timing of the animation. For this purpose it must be possible to adjust the time stamps of the operators. Since the operators are related to each other we also need to be able to maintain the temporal relationships between operators.

To handle these needs we have operator groups. Operators can be grouped in the timeline and the timing of the group can be adjusted as desired. Figure 3.3 illustrates how operator grouping works.

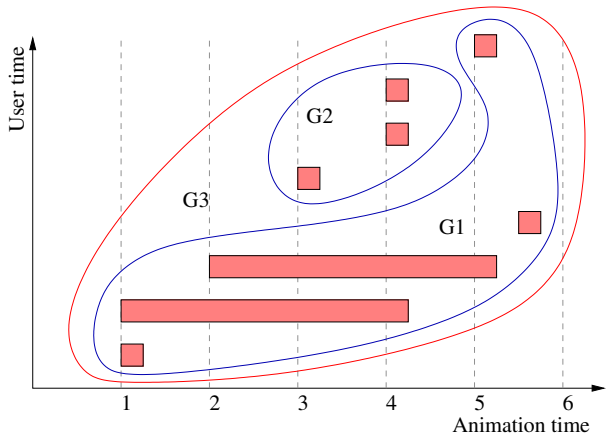


Figure 3.3: Operator grouping example. First the user has grouped four operators to group G1, then another four operators to group G2, and finally groups G1 and G2 to a group G3. The groups can be moved about in the animation time and streched.

An operator group is not limited to any particular layer. A group may span multiple layers. The group of an operator is presented in one class member that holds the id of the group in question.

3.1.11 Config

Current editing parameters (profile size, color, material etc.) are stored into config. Config has no internal logic. It is simply a parameter container that separaters recorders from the WIMP.

WIMP (section 3.1.12) stores parameters into the config and recorders (section 3.1.14) read their input values from config.

3.1.12 WIMP

WIMP (Windows, Icons, Menus and Pointers) is the collection of widgets that the user needs to configure tools and make them work. These widgets include buttons, sliders, menus, color selectors etc. The WIMP component creates an actual user interface from these basic widgets and sends stores relevant parameters to the config (section 3.1.11). In figure 3.4 is one example of WIMP in use. WIMP is also known as the "kiosk" — it resembles a kiosk with all sorts of goodies on shelves.

WIMP does not create operators or geometry, but it activates recorders (section 3.1.14) that do the real work. WIMP is a highly uncreative piece of work: At best it mimics 2D GUIs without using VR in any special way — kind of 2D in VR. Since Config separates WIMP from recorders the WIMP can be replaced with some more innovative approach is we ever come up with one.

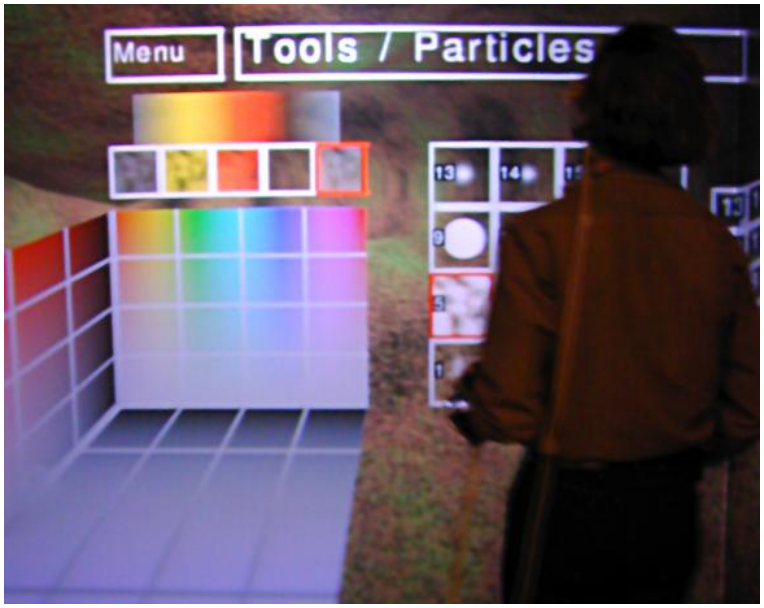


Figure 3.4: The WIMP.

The current Helma WIMP (starting from 5/2004) is configured via an XML-file[1]-file that has specifies the widgets, their functions and locations. The user can move the WIMP components during run-time. As the application exits, the modified WIMP configuration is stored back into the configuration file.

[1] For more information on XML (extendible markup language) see www.w3c.org/XML

This highly configurable approach is means that a programmer does not need to do the layout of the WIMP in code but rather specify the functional parts in an XML file. An advanced user can then fine-tune the user interface to to his/her liking and the changes are remembered between sessions.

The WIMP shows a graphic demonstration — demo curve or sphere — of what one could create with the current tool. This is useful information since one can immediately see what kind of results can be expected with the current drawing color, profile (for tubes), texture (for particles) and other parameters without actually drawing anything. See? Effective user interface design :-)

3.1.13 Quick Choices

The main WIMP is large, and using it typically causes abrupt brakes into the workflow. For the reason we have created a “quick choices” -mini WIMP that is used while drawing.

The quick choices are activated by moving one hand above another. Once activated the user is offered the WIMP entries most imortant for the tool in use. The quick choices are deactivated by moving the hand away from its vicinity. Unlike the main WIMP the quick menu does not hide the user’s graphics.

3.1.14 Recorders

Recorders are classes that turn user input into operators. All recorders inherit a common base class. Typical recorder takes all parameters that Näprä and magnetic tracker have to offer and turns them into operators that have the data inside.

A user is does not need to be aware of the recorders — they are part of the application background magic.

3.1.15 Timeline

All operators exist in a time-line. On the time-line they can be moved, scaled etc.

Time-line exists, but there is no real user interface into it.

3.1.16 Gallery

Gallery is the clip-board of Helma. The user can take the state of the world (or some layer), take an exact copy of the geometry it and store the geometry into the gallery. Later on the user may copy objects from the gallery to the world.

3.1.17 Snapshots

For the gallery to be of any use we must be able to get data in there. For this purpose we have a snapshot feature. Snaphost takes some geometry and makes an independent copy of the geometry. Besides gallery snaphost are useful when one needs to stop the inheritance of geometry between frames.

3.2 Support Components

In the previous section the concepts and components that are fundamental to the Helma system were explored. In the following we will cover the support components that are necessary to the system, but are not conceptually very important.

3.2.1 Renderers

Any data needs to be rendered. For this purpose we have separate renderers that traverse all objects in the world and render them. The most important is the GEE_GlRenderer that is used to display graphics with the OpenGL. Another one is CollectorRenderer.

The collector is a fake renderer that also traverses the world, but instead of rendering anything, it stores all objects to two lists (one for solids, and one for transparencies). These lists are then rendered by GlRenderer (solids front-to-back and transparencies back-to-front).

See also section 11.11 if you want to understand how rendering really happens.

3.2.2 OpenGL Data (GlData)

Helma uses plenty of OpenGL resources — textures, display lists, various renderers etc. These resources are represented by a series of objects that are packed into one GlData packet. These packets are thread-safe.

3.2.3 GEE_Engine

This is simple utility component that manages world and the related operator manager.

3.2.4 Crystal::Engine

This is a large object that knows how to run WIMP, Config, Recorders and GEE_Engine.

3.2.5 Different Applications (Qt and VR Juggler)

There are two different application hosts that can run Crystal::Engine. One is a Qt version that runs as a normal desktop application and the other is VR Juggler version that runs Helma system in a Cave.

3.3 Modes

There are two modes in the Helma main application.

3.3.1 Drawing/Animation Mode

Drawing mode is used for crafting the graphics directly. This mode offers direct-manipulation approach to making graphics.

3.3.2 WIMP Mode

WIMP mode is used to select and configure tools (sprays, magnets, meshes etc.).

3.4 Remarks

By now the reader may have noticed that Helma does not follow the approach of typical animation applications (Maya, Softimage, Lightwave). In the following we cover some differences:

In many applications there are objects that are first modeled and after modeling animated. Not so in Helma. The world is a place with some amount of geometry in different forms. The goemetry can be mutilated from frame to frame in any way. One could say that the world is a place where material (geometry) that can be crafted with a selection of tools.

In the end it means that Helma by nature produces not models, but a series of instructions (operators) that create models. Anybody trying to use Helma models in other application will probably have to export some particular frame/layer combination as a model.

3.5 Pros & Cons

Some of the problematic issues with this architecture:

- Replicating the modeling instructions is difficult. The system should produce exactly the same animations and models at every single pass. This is not quite as simple as it sounds: In the real-time modeling phase the system uses slightly dufferent code path compared to remodeling at later time. While this code is 95% identical the remaining 5% can and does still cause trouble.
- How to fix buggy operators? Suppose we make a new operator and we find at later stage that the operator is buggy. If we fix the problem without further ado then any older files will be displayed with different looks — the new operator will construct geometries that differ from what they were originally. If a lot of art has been made with a buggy operator we have little choice but to make work-arounds so that the old files are processed as they were (with the bugs) and the new files are processed with new, corrected code.
- A complex animation requires huge amounts of computation per frame. It can be difficult to compute the animation in real-time (and also impossible to precalculate the content of the animation). This means that the user may be forced to create the animation in slow-motion and it can be viewed in real-time only by making a movie (avi etc.) out of it.

Some of the positive things about our approach:

- The artist can work with the history of the geometry. Since the original geometry construction commands are still available one can adjust parameters after making the models, for example by changing the original texture of particle.
- Unlimited undo and redo are automatically built into the system.
- The same tools are used to create animations and static art. This reduces programming work, simplifies the application and lowers the artists' learning curve for making animations — any tool that is used for static art automatically extends to animation.
- The system is extendable. We can add new geometrical primitives and new operators with minimal changes to the basic application behavior.
- User interface, graphics algorithms and the file format are decoupled. This enables us to develop them individually without significant side effects into other areas. In particular this means that we can test and modify different user interface ideas without breaking the underlying graphics engine or file formats.

3.6 Usability Pitfalls

In this section we analyze some weak points of the Helma system. In particular there are some usability considerations that are caused by the above architecture.

3.6.1 Unwanted Operator was Here

When making animations one problem is that an operator may affect geometries we did not want it to affect. The following example illustrates the problem:

1. User draws a human figure into animation frame 1

2. User then distorts the figure in frame 2
3. User then draws some objects around the figure in frame 1
4. As we go into frame 2 the objects are distorted with the figure

One might suggest that we solve this problem by making a new rule: “An operator should not affect geometry that was not around when the operator was first used.” The following counter-example proves that the new rule creates new problems:

1. User draws a fishing net into animation frame 1
2. User then distorts the net in frame 2
3. User then adds more detail to the net
4. As we go into frame 2 the new details should be distorted with the original net

These problems can be overcome with proper use of layers and advance planning. But for impulsive non-planned approach they are a clear problem. Possibly the “new rule” should be implemented — although it may eventually make the system more complex to understand (yet another non-obvious control).

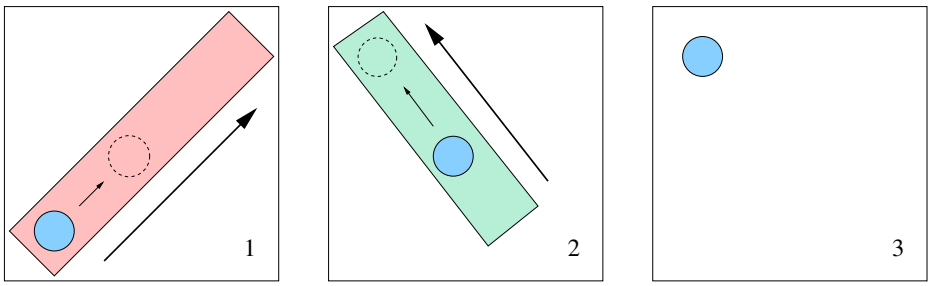
3.6.2 Stretching Animation Events Affects the Outcome

With Helma the user can adjust the timing of the animations. In theory one can stretch time and the animations will react predictably — they will simply play faster or slower. Unfortunately there are corner cases when this is not the case. Figure 3.5 shows a practical example of how this may happen.

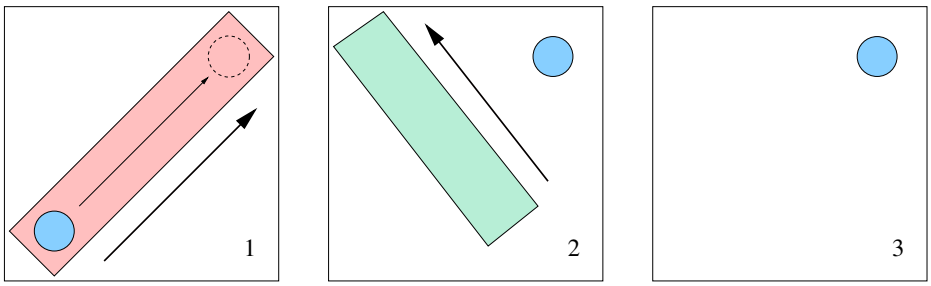
The Helma system is usually ran at a fixed frame-rate. This guarantees that running two passes over the same animation does has identical results as long as the the animation is not modified. As one can see here modifiying the timing (strecthing operators in time) may change the results of the animation.

Currently there is a machanism that should handle most of these situations correctly. The operator manager does not run the operators direcrlt, but calls operator groups to do the work. The groups then run their operators in a way that matches the original sequence. This should keep operators in sync as long as they are within the same operator group.

Whether or not this problem surfaces remains to be seen.



(a) At first the particle is moved by one magnet stroke and then by another.



(b) If run the animation faster the first magnet does all of its work at one step and the second magnet stroke has no effect.

Figure 3.5: A particle is dragged by by two magnet strokes. Depending on the timing of the strokes the particle ends in different locations.

Chapter 4

Example Scenarios

In this chapter we go through some scenarios and explain what happens in the Helma system.

4.1 Basic Drawing

Consider the following scenario:

1. User selects red drawing color
2. User selects a polyline tool
3. User draws polyline A
4. User selects blue drawing color
5. User selects a profile tool
6. User draws tube B

This scenario is accomplished with the following user/software actions:

1. User selects red drawing color.

   With the WIMP the user specifies some color. The color is stored into the config.

2. User selects a polyline tool.

   With the WIMP the user selects a tool. WIMP sets up a new recorder that knows how to create polylines.

3. User draws polyline A.

   When user triggers the beginning of the line the polyline recorder PR creates two operators: Operator O1 that contains the path information and operator O2 that turns the path information to a polyline. O1 creates path information object P1 into the world. O2 creates a polyline object P2 into the world and makes P1 invisible. PR sets the beginning times of O1 and O2 to the current animation time.

   While the line is being drawn PR collects user input data from motion tracker and näprä and pushes them to O1 that pushes them into the P1. O2 continuously takes data from P1 and moves it into P2.

   When the user triggers the end of the line PR sets the end time stamp of O2.

4. User selects blue drawing color.

   With the WIMP the user specifies some color. The color is stored into the config.

5. User selects a profile tool.

   With the WIMP the user selects a tool. WIMP sets up a new recorder that knows how to create triangle meshes.

6. User draws tube B

   Creates two operators: Operator O3 that contains the path information. When user triggers the beginning of the tube the tube recorder TR and operator O4 that turns the path information to a triangle mesh. O3 creates path information object P3 into the world. O4 creates a triangle mesh object P4 into the world and makes P3 invisible. TR sets the beginning times of O3 and O4 to the current animation time.

   While the tube is being drawn TR collects user input data from motion tracker and Näprä and pushes them to O3 that pushes them into the P3. O4 continuously takes data from P3 and moves it into P4. O4 knows the structure of the mesh and enables the use of triangle strips for greater performance.

   When the user triggers the end of the tube TR sets the end time stamp of O4.

4.2 A Case with Undo

1. User draws tube A
2. User erases some triangles of A
3. User moves some vertices of A
4. User uses undo until we arrive to the situation where A does not exist

This scenario is accomplished with the following user/software actions:

1. User draws tube A

   As in previous scenario we get operators O1, O2 and geometry obejcts P1 and P2.

2. User erases some triangles of A

   When user triggers the beginning of the erase action the erase recorder ER creates two operators: Operator O3 that contains the path information and operator O4 that erases vertices around the path. O3 creates path information object P3 into the world. ER sets the beginning times of O3 and O4 to the current animation time.

   While the user erases vertices ER collects user input data from motion tracker and näprä and pushes them to O3 that pushes them into the P3. O4 continuously takes data from P3, seeks all vertices close to path points and erases them.

   When the user triggers the end of the tube ER sets the end time stamp of O4.

3. User moves some vertices of A

   Identical to the previous step, but instead of erasing vertices the relevant operator moves vertices.

4. User uses undo until we arrive to the situation where A does not exist

   As user triggers a single undo action, one operator is removed from the operator stack. Finally we end up in a situation where there are no operators in the stack (or rather, our undo pointer points to the beginning of the operator stack).

Chapter 5

Main Software Components

Helma is based on a number of components. The main components and their dependencies are illustrated in figure 5.1. Many software components are further divided into multiple libraries.

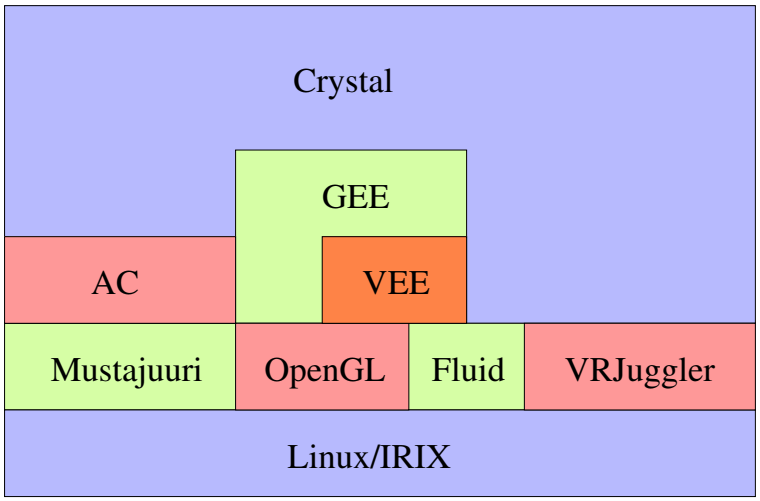


Figure 5.1: Main software components of Helma.

5.1 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 different 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 6.1.5. These are renamed to VEE_Vector3, VEE_Matrix3 etc. to make it possible to migrate to different vector library at some stage. It is also aesthetically more pleasing to use VEE_Vector3 than Fluid::Vector3 in VEE library. 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.

VEE is written by Tommi Ilmonen, Tuukka Heikura, Marko Myllymaa and Janne Kontkanen. VEE also includes source from third parties (exact information can be obtained from the web-site). For more information on VEE see www.tml.hut.fi/˜tilmonen/vee.

5.2 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)

The geometrical primitives are further explained in chapter 8.

While the number of geometrical primitives is fairly small there are a number of operators. Some important operators are:

- Path to triangle mesh conversion (GEE_PathToMesh)
- Path to lines conversion (GEE_PathToLines)
- Path to particle cloud conversion (GEE_PathToParticles)
- 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 (geometry types, operators), but no rendering
- geegl — OpenGL rendering of the graphics

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

5.2.1 Viewer application

The VEE/GEE package contains a viewer application that can be used to view the GEE animation files with a normal computer. This viewer is typically used to take screenshots of the artwork made in the VR environment. Figure 5.2 shows a screeshot of the viewer in use.

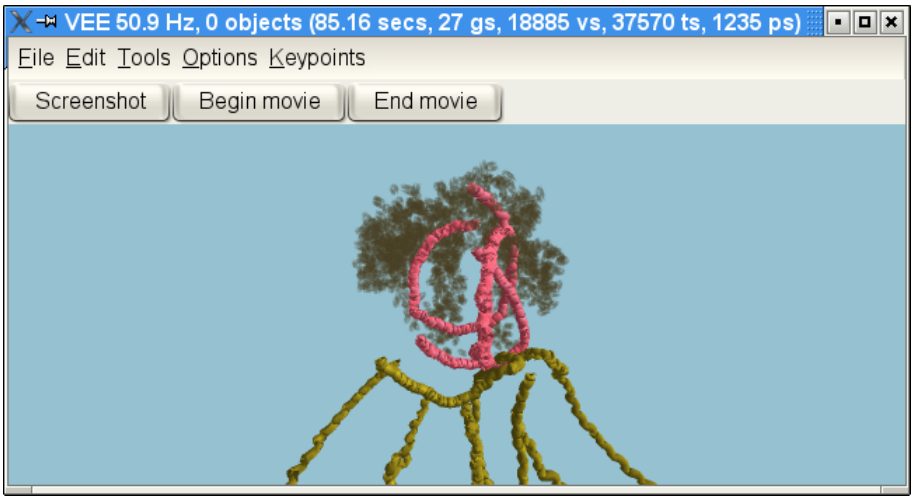


Figure 5.2: The viewer application shows a piece of higher art (a self-portrait) by Markku Reunanen. The figure has 27 geometries, 18885 vertices, 37570 triangles and 1235 particles. In this case it is running at fairly unimpressive 50.9 frames per second.

5.3 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.
- VR Juggler-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.

5.3.1 Crystal::Engine

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

5.3.2 VR Juggler-app

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

5.3.3 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 geometry 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 geometry 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 (section 6.3.4to check memory leaks etc.

5.3.4 Widgets

Crystal includes a number of 3D widgets — sliders, buttons, menus etc. These are embedded into the engine library. All widgets whare a common base class (Crystal::Widget). Widgets can be moved with the aid of a scene-graph. This is scene graph is a minimal one and it only includes the hierarchical transformations for setting the locations of the widgets.

For widgets the input data from Fluid is reconstructed into a very simple format that reminds us of mouse input data in 2D widget libraries. This is done so that widgets are easier to write and we do not need to do the conversions at each widget class.

Anybody used to the Qt way of thinking will find striking similarities between Qt widget/object model and Crystal widgets.

Widgets know how to render themselves using OpenGL. Ideally this would not be the case, but the whole widget library was began as a quick hack and this minor (?) flaw is still with us.

Widgets also know how to store themselves into an XML file and how to retrieve themselves from an XML file.

Chapter 6

External Components

Helma relies on a collection of external software components. We try to avoid work and use the work of others — thus this wildly modular approach.

6.1 Compulsory Home-Made Software Components

This section covers Helma softwaer support libraries that have been written in HUT/TML, mostly by Tommi Ilmonen. These libraries are available in the Internet, but if you really want to use them then it is usually best to get the soiurces from the CVS repository.

6.1.1 DIVA Libraries

The DIVA (Digital Interactive Virtual Acoustics) research group has lent its name to a few basic C++ libraries. These are the divabase- and divaio libraries. The divabase library contains wrappers for the most annoying platform-dependent functions — threawding, networking and endianness issues. The divaio library has methods for audio and MIDI IO on IRIX and Linux.

The DIVA libraries are distributed with Mustajuuri and Fluid source packages.

6.1.2 Mustajuuri

Mustajuuri is an audio application and a toolkit. It includes everything from simple first-order IIR filters to complex plugin graphs.

The Mustajuuri development environment is widely used in VEE, GEE, ConfigReader and Crystal. This basically means that the makefile system always eventually goes to Mustajuuri to do real work.

In Helma Mustajuuri is used to do the sound synthesis and VBAP panoration for the user interface.

For more information on mustajuuri see www.tml.hut.fi/~tilmonen/mustajuuri.

6.1.3 Qt Audio Widget Library — QAWL

QAWL is a fairly small library that provides audio file viewing tools.

- Progressive waveform preview calculator (calculates preview files without freezing the application)
- Qt-based waveform display widget
- Qt-based waveform renderer
- OpenGL-based waveform renderer

The last component is used in Crystal to render the waveform of the sound-track.

QAWL is an almost independent component in Mustajuuri.

6.1.4 AC — Auralization Control

Since Mustajuuuri is typically run on a remote host, a simplified control library was developed. This is the main purpose of AC. In practice AC knows how to create the control messages needed by Mustajuuri and sends them over an TCP/IP connection to Mustajuuri. Mustajuuri then receives the messages and runs sound synthesis accordingly.

AC is distributed with Mustajuuri.

6.1.5 Fluid — Flexible User 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 Helma Fluid is used to handle the input devices and do some basic processing on the data.

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

Fluid is written by Tommi Ilmonen and Janne Kontkanen. For more information on Fluid see eve.hut.fi/fluid.

6.1.6 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
}

/* This is a comment. */

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

The original configreader flex parser came from Janne Kontkanen. It was then modified by Tommi Ilmonen and finally rewritten as C++ by Tommi. ConfigReaded is distributed in the same source package with VEE.

6.1.7 Solar — Simple Object Loader and Renderer

Solar is an oldish small component for loading and rendering 3D models. It was made so that one could have a background scene in Crystal. At the moment it works well enough, but there is no need to develop it further since GEE also includes similar functionality.

GEE does not use Solar's graphic primitives since the data structures in Solar do not lend themselves too well to dynamic object modeling. Solar does a good job whem the models never change.

- solar (=solarbase) — The base library, including basic graphic primitives (trianle soups, meshes etc).
- solarani — Minimal animation features (move object to location, rotate object).
- solarascii — Print out the data of a model.
- solargl — OpenGL renderer for Solar objects.
- solarlwo — Load LightWave models.
- solarperf — Loads many kinds of models by using SGI Performer to do the dirty work.
- solarloaders — Loads files using either solarlwo or solarperf

Solar is distributed with VEE.

6.1.8 CaveProj

CaveProj is a minimal library that loads VR Juggler configuration files and calculates relevant projection matrices for the projection walls. It was made to replace the projection calculation code in the VR Juggler software package (section 6.2.7).

6.1.9 VR Smuggler

Since VR Juggler is an unnecessarily complex piece of software we wrote a small replacement for the OpenGL display and context management — the VR Smuggler. Both CaveProj and VR Smuggler are packed into one library that is distributed with the Fluid package (under fluid/src/smuggler).

6.2 Compulsory Off-the-Shelf Software Components

This section covers components that either link directly to the Helma enviroment (as libraries) or are compulsory for some other reason. You need these components to compile Helma. By off-the-shelf components we mean that they can be downloaded from some place in the Internet.

6.2.1 C++ & STL

C++ is our main programming language. Another one is Python (section 6.2.5)

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

Helma components need a fairly modern C++ compiler that supports templates, namespaces, nested classes and RTTI (Run-Time Type Information). The compilers that we are using are gcc on Linux (starting from gcc version 3.2.0) and MipsPro CC on IRIX (version 7.3.1 seems to be the best).

6.2.2 OpenGL

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

6.2.3 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.

6.2.4 Qt

Qt is a GUI-library for UNIX, Palm, 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 Helma has a playback engine based on Qt. Besides the GUI functionality there are other features Helma uses from Qt: The current image (texture) loader in VEE uses Qt's image loader code and Qt's DOM XML parser is used whenever XML is used.

Note that Qt is the only component with special license issues: It is distributed under a triple license — GPL, QPL and commercial license. The QPL and GPL are viral licenses — they require that any work based on Qt must be open source. If proprietary development model is needed then one can buy the commercial license.

On Windows Qt always requires a commercial license. These are per-developer licenses and they cost real money.

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

6.2.5 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 moment Python is simply a semi-interactive debugging test environment.

The C++-to-Python bindings are generated automatically with SWIG (see section 6.2.6).

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

6.2.6 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.

There are deficiencies in SWIG — it cannot parse properly all C++ headers that are relevant in Helma project. One also needs to careful with the order of including C++ fiels into the wrapper definitions file. In practice the problems are rare enough to be a minor nuisance compared to the features that SWIG offers. A fairly new SWIG version (1.3.19 etc.) is needed to generate the wrappers.

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

6.2.7 VR Juggler

VR Juggler is a large VR-framework. In our work we only use VR Juggler to manage the wall projections and to manage OpenGL contexts and windows in Helma. Bitter experience has thought us that VR Juggler is not quite as flexible, simple and easy-to-learn as one would wish. Consequently we try to avoid VR Juggler as much as possible and use other available APIs instead (in particular Fluid).

For more information on VR Juggler see www.vrjuggler.org.

6.2.8 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.

We also sometimes take code from WML and put it into Fluid or VEE. In particular many more complex matrix/vector functions are taken from WML (inversions, rotations etc.).

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

6.2.9 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 seems to be stable and fast.

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

6.2.10 PLIB — Steve's Portable Game Library

PLIB is a collection of games-related graphics tools. PLIB includes a scene graph, object management basic primitives (vectors, matrices etc.) and all sorts of usual 3D game code. For us the most important component is the OpenGL-based text rendering engine. This is the only component that Helma uses from PLIB. The relevant code is located in the PLIB's Fonts'n'Text (FNT) -library. FNT-library is used for all OpenGL text display.

For more information on PLIB see plib.sourceforge.net.

6.2.11 Libaudiofile

DIVA audio file IO is based on the libaudiofile library. Libaudiofile was originally made by SGI, but an open-source implementation (for Linux etc.) was later made by Michael Pruett. Libaudiofile handles most common audio file formats (waf, aiff). Sometimes if you try open a file that does not exist libaudiofile crashes — sigh.

For more information on libaudiofile see http://www.68k.org/∼michael/audiofile.

6.2.12 System Audio and MIDI Interface

Each operating system has some special audio interface that DIVA IO classes uses. On Linux it is the OSS and ALSA, and on SGI the libaudio. These need to be configured properly for audio to work.

For a each new system new interface wrapper classes are typically needed.

6.3 Auxiliary Tools

This section covers tools that are commonly used to maintain and compile the Helma components.

6.3.1 Doxygen and Graphviz

Doxygen is a tool that generates documentation from C/C++ source files based on comments with special tags. Doxygen can use the "dot"-tool in Graphviz-package to generate fancy graphics to the documentation.

Doxygen is used for all API-level code documentation in Helma. Unfortunately not all code is documented fully, but such is life. Fortunately all code is self-documenting — you can simply start to write code.

For more information see www.doxygen.org and www.research.att.com/sw/tools/graphviz.

6.3.2 CVS — Concurrent Versions System

For source code management we use CVS. CVS is wonderful, multiple developers can write code in their own trees and then contribute to the central code repository. Each internal Helma component (Fluid, Mustajuuri, VEE, Crystal) has an independent CVS tree where the project is stored.

For more information on CVS see www.cvshome.org.

6.3.3 Linux & IRIX

Currently Helma runs on Linux and SGI IRIX. In practice porting to a POSIX-compliant system should be fairly easy. In the future SGI will probably die or fade (with their outdated MIPS and IA-64 technologies) while Linux is a growing platform.

6.3.4 Valgrind

From Valgrind's home page: "Valgrind is a GPL'd system for debugging and profiling x86-Linux programs. With the tools that come with Valgrind, you can automatically detect many memory management and threading bugs, avoiding hours of frustrating bug-hunting, making your programs more stable."

Valgrind must be one of the most useful open-source application development tools out there. Anybody doing any development anywhere should check their applications and libraries with valgrind every once in a while. We use Valgrind every sometimes to check that there are no memory errors in Helma code base. Having such an accurate error-finder reduces time spent in bug-hunting.

Valgrind typically spots problems that we cannot fix — errors in system libraries etc. Since these errors often hide the real problems one can use a suppression file to hide unnecessary warnings. This suppression file — vee.supp — is in the vee/src/python directory.

For more information on Valgrind please see valgrind.kde.org.

6.3.5 GNU Make and Autoconf

We use makefiles for project management. Nobody has said that makefiles are easy to use or fun, but they work fairly reliably in a multi-platform development environment (at least more reliably than any other alternative we know).

To complement makefiles we use autoconf to adjust the platform-specific configurations. Autoconf is one of the most frightening pieces of software we use. Nobody knows how it works or what it really does, but it seems to do something.

For more information on these tools see the GNU web site — www.gnu.org.

6.3.6 Performance Profiling Tools

The Helma applications should be as fast as possible — fluent graphics are a critical part of VR usability. To spot performance bottle-necks one needs tools that can profile the application behavior.

Gprof is a tool by the GNU project to profile C/C++ applications. One needs to compile all relevant source code with special compiler flags. Then the application is ran as usual. Upon exit the application dumps profiling data to a file. The profiling data can be turned into more human-readable form with a special "gprof"-application. To get even more human-readable representation one can use the kprof application (kprof is part of the KDE desktop environment). The gprof/kprof -system works with GCC and we use it on Linux.

On IRIX one can use applications "pixie", "ssrun" and "prof" to analyze application performance. These are part of the speedshop software package. First one uses pixie to transform any binary into an instrumentation version. Then we run the application in ssrun (ssrun app.pixie). Prof then processes the performance data file into more understandable form.

None of these profiling tools work with threaded applications. This is sad — the most critical applications (VR Juggler-app and Qt-app) cannot be tested with these tools. The most effective way to use profiling tools is to run them on the viewer application (that uses most of the CPU- and GPU-intensive code). So far the analysis has shown few if any places for optimizations — most of the time is spent in OpenGL and there is little we can do to help performance there.

For more information on these profiling tools see respective IRIX/Linux manual pages or try a Google search.

6.4 Remarks about Portability

Almost all of the above software components are also available on other common platforms: Windows and Apple OS X. The components that are not instantly available can be ported without excessive work. Most components also work on other interesting systems: Linux/embedded and Palm, but porting Helma to such platforms is likely to be a bigger task.

There are some legal and technical caveats in such porting work (e.g. the Qt license on Windows — section 6.2.4), but in principle the system is portable to any reasonable platform. A reasonable platform is one with full threading-support, normal file- and networking features and the usual C/C++ development tools.

Chapter 7

Application Control Flow

In this section we explain how the VR version of Helma works. To illustrate this topic we go through a single processing cycle from one rendered frame to the next. That is: the user is in the Cave and the system shows something on the walls — the graphics on the walls are updated as quickly as possible. Here one frame means interval between graphics updates.

The key actors in the following scenarios are:

1. VR Juggler (VRJ) — The VR Juggler library
2. Crystal::Engine (CE) — The main object in Helma
3. Helma VR Juggler-application (VRJ-app) — A class that runs CE inside VRJ
4. Helma Qt-application (Qt-app) — A class that runs CE inside Qt

And now one operation cycle (see also figure 7.1):

1. In the beginning our application has been loaded into VR Juggler and VR Juggler has the control.
2. VRJ calls the “update” method of our VRJ-app in the main application thread.
3. The VRJ-app owns a CE object whose “update”-method it calls. Now CE has the control.
4. CE updates the Fluid input and processing layers.
5. Control is given to the WIMP that processes whatever is needed and stores any new parameters to the config.
6. Inside WIMP control is given to the active panel.
   (a) If user wants to load/save a new file the operation is done immediately.
   (b) If user changes parameters, then config is changed.
   (c) If new recorder is needed, then the recorder held by CE is updated immediately.
7. CE runs the current recorder. The recorder may put new data into the operators held by operator manager.
8. CE updates the state of the operator manager with the current time stamp.
9. Operator manager runs the operators that are active, thus causing animation and/or editing to occur.
10. If geometry is modified or deleted, this is stored for use in the later “draw” function.
11. CE updates the animated particle system.
12. CE calls collector renderer to collect and sort all particles and geometry.
13. CE returns control VRJ-app.

14. VRJ-app returns control to VR Juggler.
15. VR Juggler readies the rendering threads that render the graphics.
16. The threads call the “draw” method of VRJ-app.
17. VRJ-app continues the calls to CE and calls its “draw” method in multiple threads. VRJ-app has OpenGL-context -specific GlData objects that are transmitted to CE.
18. All display lists that point to deleted objects are erased.
19. CE renders all geometry and widgets using the GlData objects. If a geometry has changed the relevant display list is regenerated.
20. Control returns to VR Juggler and we return to step 1.

Control flow of the Qt application is nearly identical to the VRJ-application. The main difference is that all operations are done in a single thread and the top level is control is in Qt rather than VR Juggler. Additionally there are operations that one can do in the Qt GUI (record movies, take screenshot etc.).

To make the matters more complex we also have VR Smuggler based application. In this case the logic is basically identical to the VRJ-app.

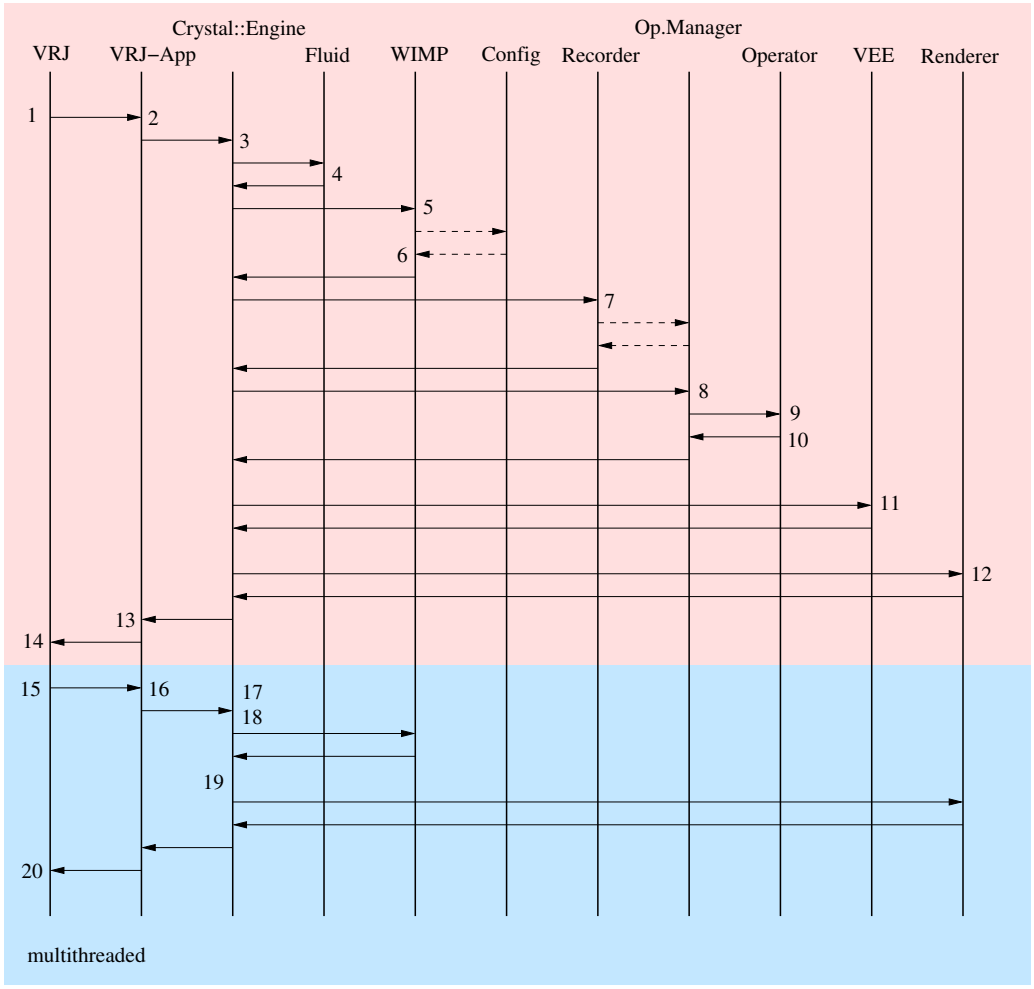


Figure 7.1: Sequence diagram of the control flow.

7.1 About Rendering

When the system is in the animation mode rendering is fairly straightforward: The CE collects the particles and geometries in the “update” -phase and later on renders them.

In the WIMP-mode things are different: We may have visible GEE objects all over the place (gallery, demo etc.) that should be rendered. The trick we do is that each WIMP component stores its geometry into the collector renderer in CE at each frame (with proper transformations).

7.2 Hardware & VR Juggler

The architecture above is largely dictated by the kind of a platform that Helma runs on. To make give the reader a better understanding of how the hardware and software tools limit our design choices we illustrate the internal structure if the graphics computer and how VR Juggler works with the graphics hardware.

The graphics computer (or cluster) must be able to render multiple screens simultaneously. Since a single graphics board cannot run four walls at one time there must be multiple graphics cards in the computer (in SGI lingo graphics cards are called pipes). Since the cards are separate entities they may belong to different X-windowing display. Effective use of graphics resources requires that one pushes OpenGL commands to all available boards in parallel. An alternatively would be to render one wall at a time — thus running the multiple graphics boards one at a time and losing plenty of performance.

To be able take full advantage of the underlying hardware VR Juggler uses a separate rendering thread for each window. For this reason it must be possible to do multi-threaded render of the graphics in the application. The VR Juggler master loop looks roughly like this:

1. Run the application logic (virtual method “preFrame”) in one thread.
2. Render the application data (virtual method “render”) with multiple threads.

In general programmers try to avoid threaded programming unless they are forced into it. This axiom holds here also — the application logic is single-threaded and rendering is parallel due to hardware requirements.

There are some unnecessary problems with the current system. In particular we would like to have more direct access to the OpenGL. Unfortunately VR Juggler keeps us away from the graphics details and we cannot do things (like render some graphics on display Y, take screenshot and store the screenshot to the file).

So far we have been discussing the topic in terms of a single graphics computer. Running on a cluster of computers does not affect things radically. Each graphics window is still opened to some X-windowing display. If the displays are remote, then GLX will transmit the OpenGL calls transparently across the network.

The only new consideration in the clustering situation is (or should be) the bandwidth of the network and transmission overhead: The bandwidth of the intra-computer graphics bus (AGP etc.) is always much higher than the inter-computer communication (Gigabit Ethernet etc.). In clustering environment one must minimize the amount of geometry sent over the network. This can be done by using display lists for all stable geometry. GEE already includes support for using display lists, so cluster support should be relatively easy to implement if we need to use a cluster at some point in the future.

Chapter 8

Geometrical Primitives

This chapter gives overview of available geometrical primitives and how they work.

8.1 3D Path

This geometrical primitive is often used, but seldom visible. It is a class that holds the motion data of one hand motion sensor and related Näprä parameters.

3D paths are used to store user input. Other operators then turn the path into other kinds of geometrical primitives (meshes, lines, particles etc.) and actions (move, rotate, distort, recolor, retexture etc.).

8.2 Triangle Mesh

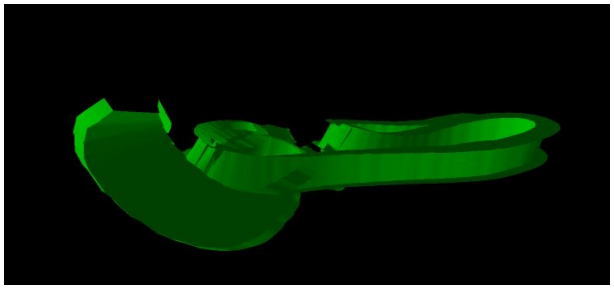


Figure 8.1: A triangle mesh made with an I-bar profile.

Triangle mesh class is the usual kind of triangle mesh — there vertices with some parameters (location, normal, color, texture-coordinates) and triangles that are formed by indexing the vertices. Figure 8.1 shows an example of a mesh.

A peculiar feature of this triangle mesh is that instead of using normal C-style arrays both triangles and vertices are stored into STL maps. Thus both vertex and triangle arrays can be sparse. This data storage approach is used since it offers a reasonable performance in all the operations that we must perform on the mesh:

- Add vertex
- Erase vertex
- Add triangle

- Erase triangle
- Find connected vertices
- Find connected triangles

8.3 Line Group

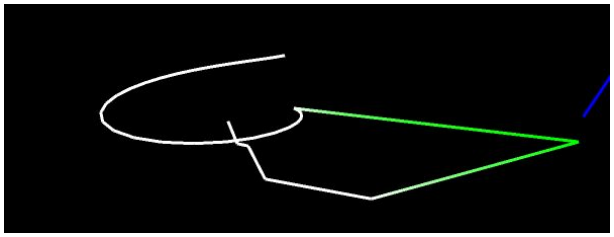


Figure 8.2: A line group.

Polylines are another way to model solids. The width of lines is defined in pixels and they have no internal structure. A line group may be broken — in that case it is constructed out of multiple polyline segments.

Internally a line group is composed of a list of vertices. Each vertex has connectivity information that marks whether the line continues from vertex n to vertex n+1.

8.4 Particle Cloud

Figure 8.3: A particle cloud. The base path is identical with figure 8.2.

Particle clouds are a way to model fuzzy things. Figure 8.3 shows an example of what a cloud may look like. The particles have location, color, size, texture, and velocity. In GEE the velocity is only used to specify how the particle should be aligned.

When writing particles to a file the name of the texture is used. Internally particle textures are identified with an integer, but since the integer id may change names are used to make the system more robust and future-proof.

8.5 2D Profile

8.6 Group

Geometries can be grouped together. Once grouped the geometries will behave like one geometry and they can be moved, scaled and erased together. Once a geometry is inserted into a group it is marked as invisible and further access to the geometry will take place through the group. Surprisingly a group can span multiple layers. It is not obvious that this is particularly useful feature, but since the feature is free (zero coding effort), we use it.

Groups structure can also be displayed: The OpenGL renderer shows a stree structure to show that some geometries form a group.

8.7 Brush

The brush tool is used to paint multiple strokes of mesh, lines or particles at once. To be precise the brush tool is not a geometrical primitive — it is internally implemented as an operator that uses one 3D path to create multiple 3D paths. Thus one can in principle connect any kind of geometry generator operator to the brush tool and run erasers, magnets etc. with it.

8.8 Remarks

At the moment there are 3 primitives that have some kind of curve/polyline meaning: line groups, 3D path and 2D profile. This feels redundant. In the future we may decide to forget some of these classes and make do with fewer geometrical primitives.

Chapter 9

Features of Helma

This chapter gives a list of the current Helma features. First is the list of GEE-based features. These use the GEE code base which is a fairly clean system without dirty hacks.

[F1] One can draw 3D lines.

[F2] One can draw triangle mesh (tube) with some profile.

[F3] One can draw/spray particles into the space.

[F4] One can draw at once multiple strokes of tubes, lines and particles with a brush tool.

[F5] One can insert textured rectangles into the world.

[F6] One can set the current drawing color, including RGB, alpha and burn -components

[F7] One can load any predefined textures into the system. This applies to both particle textures (F 3) and textures for the rectangles (F 5).

[F8] One can set the size of the current drawing tool.

[F9] One can set the radius of the current drawing/adjustment tool.

[F10] One can select the profile to be used for meshes.

[F11] One can choose how the profile is rotated when drawing (“motion” or “sensors” -rotations are available).

[F12] One can select the texture texture to be used for particles.

[F13] One can move the vertex locations of all drawn geometries (= lines, triangle meshes and particle clouds).

[F14] One can erase the vertices from all drawn geometries.

[F15] One can erase complete geometries.

[F16] One can recolor the vertices of all drawn geometries.

[F17] One can recolor complete geometries.

[F18] One can change the texture of particles.

[F19] One can move geometries without changing their rotation.

[F20] One can move geometries and change their rotation at the same time.

[F21] One can apply uniform scaling to all geometries.

[F22] One can apply non-uniform scaling to all geometries.

[F23] One can turn objects and groups around their center axis.

[F24] One can group geometries. A group behaves like a compound geometry.

[F25] One can ungroup a group.

[F26] One an copy one gallery item back into the world. The insertion happens at specific time and layer.

[F27] All edit operations (F 1-F 26) can be animated.

[F28] One can control the shape of the 3D tube with näprä. The profile is deformed with with the current näprä shape.

[F29] Several tools besides the drawing can take their effective radius from näprä.

[F30] All edit operations can be done with either hand.

[F31] All edit operations can be done in parallel, with one hand using one tool and the other hand using another tool.

[F32] All edit operations can be instant (non-animated).

[F33] All edit operations can be undone and redone.

[F34] When performing an edit operation the relevant geometries are highlighted before the actual operation is began to let the user know the scope of the operation.

[F35] One can set the background color of the world.

[F36] One can use a background scene.

[F37] One can hide a background scene.

[F38] One can show or hide the group structures in the scene.

[F39] Animations and models can be saved to a file and loaded from a file.

[F40] One can adjust the volume of the WIMP sounds.

[F41] One can play a predefined sound-track along with the animation.

[F42] One can adjust the volume of the sound track.

[F43] One can specify OpenGL light stes from a configuration file.

[F44] One can select the OpenGL light set with the WIMP.

[F45] One an copy the current world state or selection into the Gallery.

[F46] One copy items from the gallery.

[F47] One can set the available textures from a configuration file (any png-image will work).

[F48] The raw user input data (motion tracker data, Näprä, Wand) can be stored to a file and replayed later.

[F49] The user can navigate in the scene (change the location of the physical space in relation to the virtual space).

[F50] The user can move the WIMP to any location.

[F51] The user can operate the WIMP with either hand.

[F52] The user can select tool individually for the right and left hand.

[F53] The user can tune the WIMP by changing its layout interactively or via an XML file.

[F54] The system display a demonstration graphic made with the current tool.

[F55] If the user navigates far from the WIMP the WIMP follows the user.

[F56] The user can return to the origin of the world (after getting lost).

[F57] The user can move to the center of the graphics (after the graphics are lost).

[F58] The user can select lights from predefined lighting-sets.

[F59] The user can adjust näprä calibration parameters via the WIMP.

There are also features that are coming from the old Crystal. These features need to be updated to use the new GEE environment. Many of these features are dirty hacks and sorely need a rewrite or update.

[F60] One can specify the size of a particle over its lifetime.

[F61] One can specify the color of a particle over its lifetime.

[F62] One can specify age-limit of a particle.

[F63] One can set moving particle sources to the scene.

[F64] One can set stationary particle sources to the scene.

[F65] One can set parameters (rate, radius, emission volume, etc.) of the particle sources.

[F66] One can set force fields (blow, vortex, explosion) to the world.

[F67] One can draw triangle meshes with metaballs.

[F68] One can adjust the play-back speed of the animation — starting from zero.

The previous features related to the VR application which is what we usually mean by “Helma system”. In addition to the VR system we have a desktop application for loading and displaying animations. This viewer-application has the following features:

[F69] One can load and play animations made with the VR system.

[F70] One can adjust the playback speed.

[F71] One can take screenshots of the OpenGL window contents.

[F72] One can take movies about the animation with motion blur.

There is also a Qt-app for viewing the user actions. This has features similar to the previous viewer-application but it does not load actual animation files from the hard-disks, but:

[F73] Replay the user input data, resulting in more or less the same modeling and animation sequences.

Chapter 10

Installation & Compilation

It takes some time, effort and UNIX environment variables to compile Helma. The basic procedure is outlined below.

There ase two places to get the source from:

1. The CVS trees of the relevant source packages. These have the newest code and almost always compile nicely together.
2. The Helma tarballs in the web-site. These packages contain all of our own software: Mustajuuri, Fluid, VEE and Crystal.

10.1 Manual Installation

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 Crystal (Helma main application). Get the sources from CVS and compile. Configuration is not needed.

10.2 Build Scripts

There are also scripts that one can use to do the installation. These scripts are used to make the release versions of Helma (section 11.16). These scripts can be found under the crystal/scripts directory. Typical way to use them to handle the installation is (for the web-release, on the IRIX platform):

```
# Uncompress the source package
tar xvfz helma-0.1.0.tgz
# Enter the Helma directory
cd helma-0.1.0
```

```
# Compile all components (using only one CPU)
zsh compile.sh 1
# Start the application with wand control
zsh helma-wand.sh
# Start the application with Näprä control
zsh helma.sh
```

This system has not been tested too much so your mileage may vary. The scripts should work on IRIX and SUSE Linux 9.1 systems. If the compilation process manages to create a binary beneath the “crystal/vrjuggler” -directory (under IRIX-6.5, Linux-i686 or similar) then it was successful.

10.3 Compilation Environment

The build environment has a number of targets that the makefiles support. Some of the most useful ones are:

- all — builds the default targets (typically a shared library out of a collection of sources)
- dep — rebuilds the dependency information and makes sure the compiler won’t regenerate the information until requested
- don — enables rebuilding of dependency information
- clean — removes object files
- cleanlibs — removes libraries, useful if you need to only relink everything
- dlib — build a dynamic library
- slib — build a static library
- dbin — build an executable with dynamic linkage (usually what we want)
- sbin — build an executable with static linkage (sometimes useful for etecting linker problems)

These targets can be issued at the source directories or in higher-level directories. The latter will result in resursive traversal of all required child directories.

Mustajuuri build environment can create one library and/or one executable per directory. There are a number of useful options for make that one can use (for Mustajuuri, VEE and Helma):

- -k — If some source file cannot be compiled, compile the rest anyhow
- -j n — Parallel compilation, a must on multiprocessor system. Note that the Fluid make system does not work properly with -j option. Also you may need to do multiple passes to get everything compiled, but in the end things usually work ok
- OPTIMIZE=FORCE — Force optimization on.
- OPTIMIZE=DEBUG — Force debugging on.

The last two items are used to override directory-specific optimization settings.

Chapter 11

Some Remarks

11.1 VEE_Operator vs. GEE_Operator

Both VEE and GEE contain operator classes. Despite similar naming these are completely different things. GEE operators have been discussed in this document while VEE operators are of no importance in this discussion.

A programmer using both VEE and GEE should be aware of these different components (and not get confused by them).

11.2 Configuration Files

The behavior of many Helma's modules can be configured via configuration files. Below is a list of configuration files that one can use to modify system behavior. Their exact file formats etc. can typically only be deduced by looking at comments inside the files and the source code that reads and interprets the files.

- anima-config — Main configuration file for the Crystal environment
- helma.xui — XML-based Helma VR WIMP specifications
- fluid-config — Fluid input device configurations
- vee-gl.config — VEE particle texture configurations
- eve.vrjfluid.config — VR Juggler configuration for Crystal

11.3 Thread Safety

GEE is not thread safe 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.

11.4 Debug and Error Output

There are special functions for error reporting and error output. These functions are called XXX_trace, XXX_error and XXX_fatal (where XXX is the name of the module, for example VEE or GEE). All of these functions work like printf, with one important difference: Unlike printf they work properly with threads. They also provide us with a few code points where we can intercept textual output and potentially silence the system without changing the code.

Each library should have its own functions that output the name of the library before the actual message.

11.5 The Interplay Between Operators and Geometry

Let us study a practical situation: An operator wants to move some vertices of a geometry object. How exactly does this happen?

The most direct approach is to write an operator that knows how handle each geometry type correctly — writing a separate code path for each geometry type. If there are n geometry types and m operators with slightly different operating principles, then we end up with n∗m different nearly identical code paths. Thus this approach is not acceptable.

Another method is to create an identical proxy API for each geometry type. Thus the operator could ask the object for vertex locations and try to modify them. This is probably a reasonable approach, with two basic problems. 1) The API needs to cover all possible features of all possible objects — it becomes large and difficult to maintain and use. 2) The use of such an API would result in several virtual function calls per transaction, decreasing performance.

The third method is to use an intermediate API to communicate the needs of the operation. In this case we create an abstract mover class that hos only one virtual function for moving vertices. Any operator that wants to move vertices needs to inherit the mover base class. The geometry objects in turn have virtual function that has the mover object as an argument. The geometry object iterates over all of its vertices and passes them to the mover that moves the vertex. The class structure that is relevant here is illustrated in figure 11.1. The drawback of this method is that we need to specify an API for each operation type (move, delete, select, recolor etc.). If some operator finds that the API is insufficient then it cannot use the framework. At any rate this is the method that has been used in Helma.

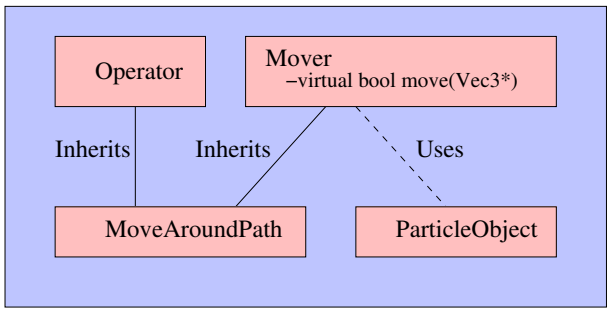


Figure 11.1: The intermediate API “Mover” that enables any geometry type to communicate with objects that move vertices. In this example the particle object pushes the locations of its vertices one at a time to the mover object.

It is of course possible for operator to approach the geometry obejcts directly, for example a tesselator is probably easiest to make by having an operator that uses dynamic_cast to find triangle meshes and then has only one code path for this purpose.

At any rate these options are not mutually exlusive — different operation styles can co-exist and the operation logic can be changed without modifying the system in general or breaking old code.

11.6 Particles vs. Vertices

In Helma there are geometrical primitives that have similar operations (for example moving). We typically consider a particle to be conceptually equal to vertex — it has a location, color and other properties usually associated with vertices. In the case of particle clouds vertex-based operations are mapped to affect particles.

11.7 Identifiers

In GEE each geometry and data object has a unique identifier (an integer). The identifiers are allocated from a central repository. Each new object has an id that is larger than the ones before it. This makes it easy to compare ids and sort operators based on their age.

The ids are used widely to find geometry objects from the world. Instead of a dangerous pointer (that may point to a deleted object etc.) one uses identifier that is a much more safe way to get geometry from the world.

There is one issue with identifiers: Merging operators from different files. Since the operators in two files may be equal we need to modify some operators as prepare to load a merge operators from the file to the current operator manager. In this situation we remap the ids — all operators know how report the ids they have and can return them. We then allocate new ids and remap the old ids to the new ones. This remapping is simple in principle, but in practice it is a powerful source of bugs.

Besides making file merging possible this method removes any empty space between ids and shifts them to the lowest possible numbers (one kind of defragmenting of identifier space).

11.8 Coordinates

Helma uses normal right-handed coordinates (like OpenGL). Positive X is to the right, Y is up and Z is depth.

11.9 Random Numbers

There are many cases where an animation/modeling system needs random numbers. Usually these random numbers need to be repeatable — a random number generator must produce exactly the same “random numbers” for each pass.

The only way to guarantee that this can be done is to 1) use own platform-independent non-changing random number generator (VEE_RandomUniform etc.) and 2) make sure the seed number does no change over time. Example of how to deal with random numbers can be found from class GEE_PathToParticles.

11.10 Vector and Matrix Classes

The vector and matrix classes come from Fluid. They are template classes, so one can have float-, and double vectors in the same application. Due to performance consideration these data types do not initialize their values in any way. One needs to always set the values of all elements in the code.

The float versions of the Fluid templates are renamed with typedef using the “VEE” -prefix, Fluid::Vector3T<float> becomes VEE_Vector3 etc. It is assumed that there won’t be any need for widespread double-precision arithmetic in near future.

All matrices are row matrices. Since OpenGL uses column matrices we usually transpose the Fluid matrices before using them with OpenGL.

It should be noted that many of the external libraries we use have their own vector and matrix classes (AC, WML, PLIB, OPCODE). We use a brutal in-place casting to convert between the data types. For example three floats form a vector with x-, y-, and z -components no matter which library is in question. Simple C-style type casting is used to convert vector and matrix types between different libraries.

11.11 Rendering and Transformations

In Helma each geometry has some 4x4 transformation matrix. Groups bring hierarchical transformations to the system. In threory hierarchical transformations are really simple — we simply push the matrices to OpenGL that does all of the transformations for us.

In reality things look a little different: To be able to render things effectively we need to use spatial sorting and render all non-transparent objects front-to-back (for minimal refilling in graphics hardware). For transparent objects we must use back-to-front rendering with strictly correct order. To be able to do all of this sorting we use a collector renderer that keeps that of the hierarchical transformations and sorts the geometries based on their distance from the camera, collector also stores the total transformation matrix of each geometry. In the real rendering phase we iterate over the two lists (solids and transparencies), apply the total transformation matrix and render the geometries.

The collector is run once per every frame in the main application logic thread and the render threads then render the lists in parallel.

The matters are further complicated by the use particles. The particles are technically billboards that face the the camera. The vertex locations need to do be calculated in world-coordinates, not in the coordinates of the particle cloud. Given that the user may apply an arbitrary transformation to the cloud, for each particle we must calculate the location of the particle in world coordinates and make sure we know its velocity (also in world coordinates). For this reason when collecting the particles for sorting we store the transformation matrix for each particle. In the rendering phase the matrix is given to the particle renderer that uses it to calculate both location and velocity of the particle in question.

There are some more nasty details that one needs to keep in mind (light locations, initial world transformations etc.) — a courageous reader can find the details in the source code.

BTW: All pixels with transparent content should be rendered back-to-front. The best thing one can do is to sort all triangles by distance, clip the interleaving triangles and then render them. We do not do this, geometries are sorted by their center points alone (particles are rendered individually back-to-front).

BTW2: Once graphics hardware starts to support the OpenGL shading language (GLSL) we can discard at least some of these dirty matrix stack manipulations and also move graphics calculations from CPU to the GPU.

11.12 Operator Optimizations

Often an operator must perform operations in some part of space — for example change the color of all vertices in a given region. The brute-force implementation would iterate over every single vertex that can be found in the world and adjust them. Such operation would have a huge overhead if there are many vertices that are not affected by the operator.

To solve this problem each geometry and adjustment operator (operators that must adjust vertex colors and locations) has a pre-calculated bounding box. These bounding-boxes are used to check if it is at all possible for particular operator and a geometry to intersect. Thus we can often do early rejection of vertices in geometries that would not be affected by the operator.

There are of course even more effective spatial division techniques for this purpose, but this one is fairly simple and improves calculation speed by magnitudes.

Of course one has to remember that operators usually work in world coordinates and the geometries can be transformed in the most bizarre ways — make sure you transform your objects to the world coordinates before doing the calculations.

11.13 Coding Style

There exists a document that describes the HUT/TML coding style, written by Tommi Ilmonen. That document gives basic outline of the programming style that is used also in the Helma project. In this section we cover some issues that are specific to Helma.

11.13.1 Prefixes (VEE, GEE, Crystal, Fluid)

Each software component has its own prefix that is attached to all symbols within the library (classes, typedefs, unions, global functions). This may be a namespace (Crystal, Fluid) or a short text that is part of the symbol name (VEE, GEE).

In this document we do not use the prefixes unless they are necessary to disambiguate meaning — (Crystal::Engine vs. GEE_Engine for example).

11.13.2 Functions that Return a Pointer

In VEE, GEE and Crystal there are plenty of functions that return a pointer. For example if you want to access some particular vertex of a triangle mesh you can get a pointer to the vertex.

By returning a pointer (rather than a reference) we implicitly declare that the function may fail. If the object you are looking for does not exist, then zero-pointer (NULL) is returned.

11.13.3 Variable Names

In general one can name functions and variables in any way. There are a few conventions that are meant to narrow down the choices in common situations:

- Use “location” rather than “position” or “translation”.
- When using a pair of variables to indicate some range, use “xxxLow” and “xxxHigh” as variable names.
- If the same variable is present in different forms (e.g. float and int), use some Hungarian-style naming — for example “int iMax” and float “fMax”.
- Often one can create unique variable names by abbreviating type names — for example a variable of type “CleverObfuscatedWekkuli” could be “cow”.

11.13.4 Caching Variables to the Stack

There are many cases in VEE, GEE and Crystal where class member variables are temporarily stored into the stack (into variable that are local to the function). This is done since access to stack is always faster than access to some unknown pointer location (and it makes for smaller binaries). Typically this is not critical, but if a variable is used many times in succession, then it may be worth it to load it into stack (or register, depending on how the compiler sees the situation).

This optimization applies only to small variables (float, ints and similar). If the variable is large (takes more memory than a 4x4 matrix) then it is usually best not to use a trick like this.

11.13.5 Include Ordering

In C/C++ nearly all source files begin with a series of include declarations. Typically there are 1-30 includes. To make it easy to spot missing (or unneeded) includes the includes are always sorted with a few simple sorting rules.

1. In an "xxx.C" file the corresponding header xxx.h is always included first.
2. The includes of one library are are grouped together
3. Within the library the files are sorted alphabetically
4. The library-includes are sorted in the following order:
   (a) Crystal
   (b) GEE
   (c) VEE
   (d) Solar
   (e) Fluid
   (f) Mustajuuri
   (g) DIVA
   (h) Auralization Control
   (i) Qt
   (j) PLIB
   (k) VR Juggler
   (l) STL and other C++ headers
   (m) ANSI C headers

To get a good example with plenty of includes you can read the file crystal_engine.C.

Sometimes a particular header needs to be included before another can be included. In these cases we have no choice, but to break our rules until we get a working compilation. Libraries that tend to cause these roblems are Opcode and VR Juggler.

11.13.6 Float vs. Double

Througout the Helma components we usually use 32-bit floating point numbers rather than 64-bit numbers ("floats" and "doubles" in C/C++ jargon). This choice is done to reduce memory consumption which in turn improves performance.

Only when we are truly afraid of inadequate numeric resolution do we use the double precision. Currently the only place where this is done are the geometry transformation matrices that are usually expressed in double precision. The higher precision is used since the transformations are accumulated to the same matrix and any numeric errors can build up in this process.

11.13.7 Performance vs. Safety

In programming it is common that high application performance requires the use of tricks that are likely to result in more bugs. While both performance and safety are desireable they are eternal enemies to each other. Thus one must strike a reasonable balance between them. In Helma we typically favor safety since we do not have an army of coders that could make ultra-fast code. At the same time there is always certain respect for performance — in the most critical loops we always try to make things work as fast as possible. In the case of Helma system application performance (=frame rate, tolerance of large graphic models) is in fact a critical component of usability. Well, the same could be said of almost all applications on planet earth...

11.13.8 Trace Functions and Keyword fname

Helma software components typically print error or debug messages to the console when it is deemed necessary. The errors are reported via functions that work very much like the “printf”-function in ANSI-C. These functions are named by the library that they are used in: VEE_error, VEE_trace, MJ_error, MJ_trace etc.

Nearly always we print (in addition to the debug output) the name of the function that caused the printout to happen. This enables us to quickly locate the code that is having trouble. If a function uses debug output more than once we typically store the function name into a local character string. The name of this variable is always “fname”. Thus fname is in practice a kayword that is expected to contain the name of the function.

This helps us move code between different modules — the debug function can be copied from function to function along with the code and the name of the function will be reported correctly even after the move.

11.13.9 Perfection

Code needs to be good, *not* perfect.

11.14 Common Utility Classes

11.14.1 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.

11.14.2 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. In the end this class implements a garbage collecting system.

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

11.14.3 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.

11.14.4 Class GEE_Debug

GEE_Debug is used to check which parts of the application should produce debugging output. The user may specify named debugging targets that need debugging. The system in turns checks what parts produce debug output before printing the output. It is also possible to switch all debugging on at once.

When using this class in your code it is typically not necessary to check the debugging values at all times. Rather one can cache the debugging information to some static variable at e.g. constructor and use this information in the future.

11.15 Caveats

In this section we cover some common problems and how to avoid them.

11.15.1 Helma is Slow

In Fluid, Mustajuuri, VEE and Crystal each library has its own optimization flags (CUSTOMFLAGS in Makefile). For stable libraries this usually means good otpimisations and for unstable(development libraries we usually use debug flags. When compiling the Helma system it is easy to forget this and just use the default flags. As a result you may end up with a debugging version of many libraries. In particular the GEE libraries may be compiled this way, resulting in very slow code (up to 10 times slower than fully optimized code).

The solution to this problem is to force the optimizations on as described in section 10.

11.15.2 XXX_moc.C Fails to Compile (Qt Problem)

Qt needs a special meta-object compiler (moc) that reads some C++ header and generates some source code that is needed by the Qt's signal/slot framework. The generated moc-files are specific to that Qt version: A moc-file created with Qt version m.n cannot be used to compile against Qt x.y. Instead the moc-files must be regerated for each Qt version.

This causes some problems if the machines that one wants to use have different Qt versions — say Qt 3.3.0 in a Debian/Linux system and 3.0.2 in an IRIX system. In this case one cannot compile Helma source code for both platforms in one directory, but must take two copies of the source code from CVS and compile them separately.

BTW: For this reason one should never put the moc-files into any CVS repository.

11.15.3 Python Script Crashes

The most usual reason for a Python script to crash is incorrect memory usage. Python does not really know when an object will be deleted inside C++. As a result Python tries to delete objects that are already deleted in C++ (or vica versa). Fairly often you must manually tell Python not to delete objects. This is done by setting the "thisown" -variable of the corresponding Python object to zero. There are plenty of examples of this in the "vee/src/python" -directory.

A typical special case is that a script fails and returns immediately. The script may then return before you have managed to set the "thisown" -variable. As a result the script crashes. For this reason you must check the error output before the crash to see what really happened. Using a debugger is not going to help at all. Often the bug is easily fixed once you realise that the crash was caused by script returning prematurely, not some special memory management error.

11.15.4 Operator Fails to do Anything

Sometimes an operator just does not do anything. A common reason for this is that the operator is on one layer and the geometry you would like to modify is on some other layer. Move both to the same layer and things start to work.

11.16 Development and Deployment Strategy

There are people who use Helma and there are people who develop Helma. There are machines where it is used and others where it is developed. By counting the permutations one can see that there are a number of versions of Helma floating around. Below we outline these versions, and how they relate to each other. These versions correspond to our work in HUT/TML.

- Helma 1 — This is a stable release version that is used by the artists to make art.
- Helma 2 — This is an alternate stable release version that is used by the artists to make art. When Helma 1 is in use we can update new code into Helma 2. Then users switch to Helma 2 and Helma 1 can be updated. The version that is in use is modified as little as possible — preferably not at all. Both Helma 1 and 2 are located on the local filesystem of the main graphics computer. Hopefully this makes it possible to run them even when the TML NFS[1] breaks down (as it often does).
- Helma on a local workstation — A developer typically has an unstable development version on his Linux-computer. This is where much of the work is done.
- Helma on a shared disk — A developer may also have a version that is located on a shared filesystem (the "project" directory in TML). This is where we test VR code that cannot be tested on a desktop machine.

The master source code of all components is located in the CVS repositories. When a new piece of code is written it is merged into the CVS and then extracted into the other directories. Most of these versions are located on local disks where one does not get back-ups. Thus all artwork needs to be copied elsewhere and all important code, resource files etc. need to be pushed into the CVS.

One should commit changes to CVS frequently to minimize the risk of making inconsistent updates. Inconsistencies are not dangerous, but one gets away with less work by tackling any problems earlier rather than later.

11.17 Performance Bottleneck Analysis

The Helma system is fairly flexible and offers the user different ways to overload the system: Depending on the way Helma is used the bottlenecks are in different places.

Pixel fill capacity is heavily used when the user makes particles that are large. In these cases the GPU has to fill many large polygons.

Vertex processing capacity is heavily used when the user makes meshes that have a huge number of small polygons. Depending on the system the limit might be 1000 or 1000000 polygons, but the limit can always be reached.

CPU load can get out of hands when one does some of the nastier edit operations — for example using a large magnet on a dense mesh may require massive amounts of computation.

Network bandwidth can be a limiting factor with PC-cluster solutions: If the system cannot use display lists (highly volatile data sets) then all geometry must be transmitted at each frame. This demands more bandwidth any local area network (LAN) can offer.

11.18 Sane System Requirements

Helma code assumes certain very basic proerties from the underlying hardware/compiler combination. It should be difficult to find a system that break these rules:

- Byte is 8 bits
- Signed integers use two's complement representation

[1] NFS = Network File System

- “short” takes 2 bytes
- “int” takes 4 bytes
- “long” takes 4 or 8 bytes
- “long long” takes 8 bytes
- Floating point numbers use IEEE 754 representation

11.19 The Old Crystal

Helma is built on top of the old Crystal software, originally by Tommi Ilmonen. As a consequence it inherits a lot of code from that project. Crystal was meant to be a quick hack — something to be tested and forgotten. This did not happen, instead the application lives on.

In the Helma project significant portions of the old Crystal are erased. The parts to be re-written include WIMP and the whole animation/modeling engine. The aim is to over time mutate Crystal to be a clean well-designed application with no dirty hacks. Doing so, we fight the temptation to always start everything from scratch.

11.20 Antrum

Antrum is a simple 3D drawing application. It was made in HUT by a student group as a programming course assignment long before the beginning of the Helma project. Antrum was commissioned by Wille Mäkelä and Tommi Ilmonen worked as a technical support contact for the student group.

Later on there was an Antrum-II project (in another course) that tried to fix the usability problems in the first Antrum. Unfortunately the Antrum-II contained so many new problems that it became virtually useless.

Over the years Wille Mäkelä has used Antrum and made a lot of art with it. This experience is used in Helma to guide the development effort.

None of the Antrum source code is used in Helma. In the Antrum-I project there were seven students and in Antrum-II some more. In practice majority of the code was written by Sami Luokkamäki, without whom both projects would have failed miserably. According to Sami, the Antrum code base is a stinking pile of garbage and useless for future development. We believe him.

11.21 License Issues

Helma source code is licensed under the Lesser General Public License (LGPL), version 2.1. This means that the code is usable in open-source and proprietary projects (within the limitations of the LGPL license). Most support libraries are also released under the LGPL or even more liberal license terms. The only exception is the Qt library (section 6.2.4) that is released under the GPL, QPL and a commercial license.

To be exact the project personnels’ intellectual property rights are transferred to Helsinki University of Technology that transfers them to Tekes. The project partners in turn have decided that the source code should be published in open-source fashion and that LGPL is a fitting license. Tekes then gives us the right to distribute the source code we have written under the LGPL.

To make matters more complicated one must also keep in mind that most of the home-made software components have been released before the Helma project under the LGPL. While the license is the same and some authors are the same (=Tommi Ilmonen) the IPR route is somewhat different since in these cases the original authors really own their IPR (to the old source code). Since we are still working on the same source packages a single soure file may have code with

only one license (the LGPL), single author (e.g. Tommi Ilmonen), but multiple copyright holders (Tommi Ilmonen, Tekes etc). The lesson of this convoluted discussion is that if many pieces of software were not released now and earlier under the LGPL then we might have a problem with sorting the IPR issues.

For further discussion on GPL, LGPL and other possible licenses see for example the GNU web site: www.gnu.org/licenses.

Chapter 12

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 code :-)

GEE uses a chunk-based binary format [1] to store its data. Each object typically saves itself to a separate chunk in the file. It is hoped that such approach would separate errors in file parsing.

Each object has 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.

12.1 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.

After the version number comes the actual data. There are two basic ways to write the data:

1. Write the variables to the file in some specific order.
2. Write an identifier tag for each variable and then the actual variable.

The first approach is faster to read/write and uses slightly less disk-space. The second is more robust — often we can add new variables without any backwards-compatibility issues. Typically we use 16-bit integers as tags. The variable data follows the tags and a special end-tag marks the end of a chunk. When reading data the parser code reads one tag/variable pair at a time. Missing tags are no problem in this case (as long as default values are ok). Given these pros and cons we usually use the firts approach for the most often repeated low-level data types (polygon vertices etc.) and the second approach for the main-level objects.

12.2 Typical file format header for main-level objects

Main-level objects are objects that inherit GEE_Geometry or GEE_Operator.

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

[1] The binary format is a must since any text-format would multiply both file size and read/write-times. At the moment largest files take more than 100 Mb and having that 100 Mb multiplied would quickly lead to files that do not fit on a single CD.

12.3 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

12.4 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.

Utility class GEE_ParamWriter is of use to anybody making type 2 file IO. It is used to nearly automatically read/write all sorts of variables to the file.

In header gee_macros.h there is macro that coverts four characters into an 32-bit integer — GEE_CONST. This macro is widely used to construct headers for low-level objects (section 12.3). To create 16-bit integer tags we use GEE_CONST2.

12.5 The Old Crystal File-Format

This section is here to remind that such a beast does exist. The Crystal file-format is an XML-based system that more or less directly dumps the contents of Crystal UI panels into the XML file. Needless to say XML is not the right format for storing large amounts of numeric data. Currently the Crystal-based animations are stored into such a file and the GEE-based stuff is stored separately into its own file.

Over time the old Crystal file format will disappear.

Chapter 13

Future Plans

Here is a list of things to be done (in no particular order). These are cursory (not necessarily trivially understood) descriptions of the nakkis[1] we have in front of us.

- Make WIMP widget groups rotatable
- Other cleaning in the WIMP
- Replace all old Crystal functionality with devastating GEE-based code
- Multicolor particle spray (new idea)
- Display selected and nearly selected faces/particles/vertices
- Make file merge work
- Conversion: Particles to metaballs
- Conversion: Metaballs to meshes
- Conversion: Meshes to particles
- Port the VR app to Linux once the new Linux PC -cluster system in HUT/TML is working
- Improve gallery
- More sophisticated undo/redo stack
- In addition to tubes and stripes make more generic polygon meshes (pancakes, balls).
- Now that OpenGL drivers support the OpenGL shading language (GLSL) push particle billoard calculations into the vertex processors. GLSL is also useful for other purposes.
- Preview when opening files (jpeg image).
- Improve everything

[1] nakki = task, sausage

Index