Appendix B

Tools used

This appendix describes some of the tools used during the research described in this thesis. The software was used on the departmental network of Sun workstations and on the Silicon Graphics Crimson VGXT and Silicon Graphics Indy workstations (running the IRIX version of Unix) in the Advanced Interfaces Group.

B.1 GLprof

GLprof is a tool supplied with the IRIX operating system for analysing the performance of graphics programs running on Silicon Graphics computers. GLprof consists of a number of different parts:

- A means of recording all the graphics library calls made during one frame.
- A simulator to simulate the actions of the graphics hardware given a set of graphics library calls recorded by the previous section.
- A method of displaying the results of the simulator.
- A method of displaying the image which would be generated by the stored graphics library calls.

To use glprof the user types at a shell:

```
glprof <program name> <program parameters>
```

where <program name> <program parameters> is the text which would normally be typed to start the application. *GLprof* then presents a window allowing the user to select a number of options, and perform any of the different functions that GLprof offers (see Figure B.1).

Figure B.1: The *GLprof* main window.

B.1.1 Recording of graphics library calls

Selecting the 'Capture' icon in the GLprof main window starts the application executing apparently normally. At any convenient point the user can capture all the graphics library calls made during the next frame by holding down the 'Pause' button. These calls are captured in a compressed format into a file which can be examined later. Also stored in this file is a series of timing markers which allow the effect of the host CPU to be incorporated into later simulations. These compressed files can (using a utility supplied) be converted into text. The figures on the number of library calls made in applications (see **Chapter 6**) were made by counting the number of library calls in these text files (after removing the timing information).

B.1.2 Simulation of graphics hardware

The simulator provided in the IRIX 5.2 version of *GLprof* can simulate the Silicon Graphics VGX, VGXT and Reality Engine graphics subsystems. No method is included for simulating any other systems.

The simulator produces two separate types of result. The first shows pipeline occupancy in the graphics system (see Figure B.2). The bars (from top down) are:

Figure B.2: A *GLprof* hardware activity display

- Host CPU busy

  This bar is solid when the host CPU is busy and not performing GL calls. This shows the overhead of the application code, the operating system and any other tasks running at the same time. Experience has shown that in many cases host CPU performance can be a major bottleneck in the applications being executed on the Silicon Graphics Crimson.

- GL operations

  This bar is solid when the host CPU is busy executing graphics library operations.

- Transform

  The transform bar is solid when the transform engine is busy.

- Clip

  This bar is solid when the geometry engine is busy clipping.

- Span

  This is solid when the span generator is busy

- Fill

  This is solid when the fill engines are busy. This appears to include the filling involved in clearing the screen/current buffer.

These bars are displayed twice - the top set is a magnified representation of an area of the bottom set of bars. The second form of output is a textual description of the operations made in that frame (see Figure B.3). The textual output is useful for finding out the scene complexity (in terms of the number of triangles passed to the rendering system). The last section of this display shows the distribution of the size of triangles. In many cases this shows that their have been many more small triangles than large triangles, however the large triangles take much longer to fill and can still dominate the usage of the filling engine.

B.1.3 Displaying the image generated by the graphics library calls

Glprof uses the program 'glview' to execute the graphics library calls and thus display the image. During the capture stage the Iris GL graphics library calls are partially converted into OpenGL library calls. Glview uses OpenGL to display the file. In conjunction with a modified OpenGL extension for an X server (described below) this allows the capture of information on the individual triangles used to build up the image.

B.1.4 Accuracy of *GLprof* simulation

One major concern is whether *GLprof* actually changes the timing characteristics of the application that it is attempting to measure. The application '*Jack*' (supplied on the Sili-

```
Profile of vcviz.dist iris -f
Trace taken: Thu Jun 23 12:02:43 1994
Simulation parameters:
Graphics: VGXT, 5 spans
Window (1281x481)
Total time 81.69 ms (12 fps w/ 1.65 ms margin)


---------- $main$ ----------
CPU:
 (64.7 ms)
GL Ops:
 4 loadmatrix (0.0672 ms)
 72 multmatrix (1.66 ms)
 70 popmatrix (0.721 ms)
 70 pushmatrix (0.721 ms)
 2 rotate (0.0434 ms)
 2 window (0.0674 ms)
 178 lmbind (2.05 ms)
 4 tevbind (0.207 ms)
Transformations:
 57 Non-lighted, Gouraud shaded Tri(s) (0.234 ms)
 270 Non-lighted, Gouraud shaded Tmesh(s) (0.529 ms)
 231 Non-lighted, Gouraud shaded, Rejected Tri(s) (1.06 ms)
 1170 Non-lighted, Gouraud shaded, Rejected Tmesh(s) (2.19 ms)
 1626 Lighted (1 infinite light), Gouraud shaded Tmesh(s) (3.58 ms)
  3178 Lighted (1 infinite light), Gouraud shaded, Rejected Tmesh(s)
(4.88 ms)
Clipping and culling:
 2538 Tri(s) rejected (0 ms)
Spans:
 615 Tri(s) 0-10 spans (total 2699) (0.135 ms)
 502 Tri(s) 11-50 spans (total 14311) (0.716 ms)
 202 Tri(s) 51-100 spans (total 13120) (0.656 ms)
 185 Tri(s) 101-200 spans (total 29460) (1.47 ms)
 86 Tri(s) 201-400 spans (total 23456) (1.17 ms)
 4 Tri(s) >400 spans (total 1608) (0.0804 ms)
Pixel Filling:
 Non-Z-buffered
 2 clear >10000 pixels (total 616161) (0.77 ms)
 2 zclear >10000 pixels (total 616161) (0.77 ms)
 Z-buffer, Gouraud Shaded
 381 Tri(s) 0-10 pixels (total 1858) (0.0485 ms)
 377 Tri(s) 11-100 pixels (total 17744) (0.361 ms)
 291 Tri(s) 101-500 pixels (total 72999) (1.43 ms)
 190 Tri(s) 501-1000 pixels (total 130338) (2.56 ms)
 210 Tri(s) 1001-10000 pixels (total 711667) (14 ms)
 22 Tri(s) >10000 pixels (total 433309) (8.5 ms)
```

Figure B.3: GLprof statistical output

con Graphics virtual reality demonstration CD-Rom) displays the amount of time that was spent on graphics and the amount of time spent on other aspects of its calculations. When run with *GLprof* the amount of time displayed by *Jack* corresponded within 10% to the time that *GLprof* determined. This shows that it is likely that *GLprof* makes only a small change to the characteristics of the application being measured.

B.2 A modified OpenGL extension for an X server

The OpenGL graphics library (as implemented on a workstation running the X windows system) consists of a library which the application uses and an extension to the X server (which actually displays the graphics). The source for a simple implementation has been licensed from Silicon Graphics by the Advanced Interfaces Group. With this source it was possible to build an X server which executes on a Sun workstation and displays the output of an OpenGL program running locally on a Sun or remotely on a Silicon Graphics workstation. '*GLview*' - the program *GLprof* uses to display captured frames - is one such OpenGL application, and it is thus possible to display a frame of an application which ran on a Silicon Graphics workstation with an X server running on a Sun.

During the course of this research the author modified the OpenGL extensions to store a list of all the triangles used to construct the image, into a file. This list of triangles is taken after transformation, lighting and clipping and so corresponds to the input to the lower levels of a graphics rendering pipeline. The file is a textual file consisting of one entry per vertex. Each entry is of the form:

```
Point: x y z
Colour: r g b
```

This file format (together with a small header listing the number of triangles, together with offset and scaling information) formed the input to many of the simulations carried out during this research. Figure B.4 shows the flow of information from GLprof, through the OpenGL server, and into a number of display routines and simulators described below.

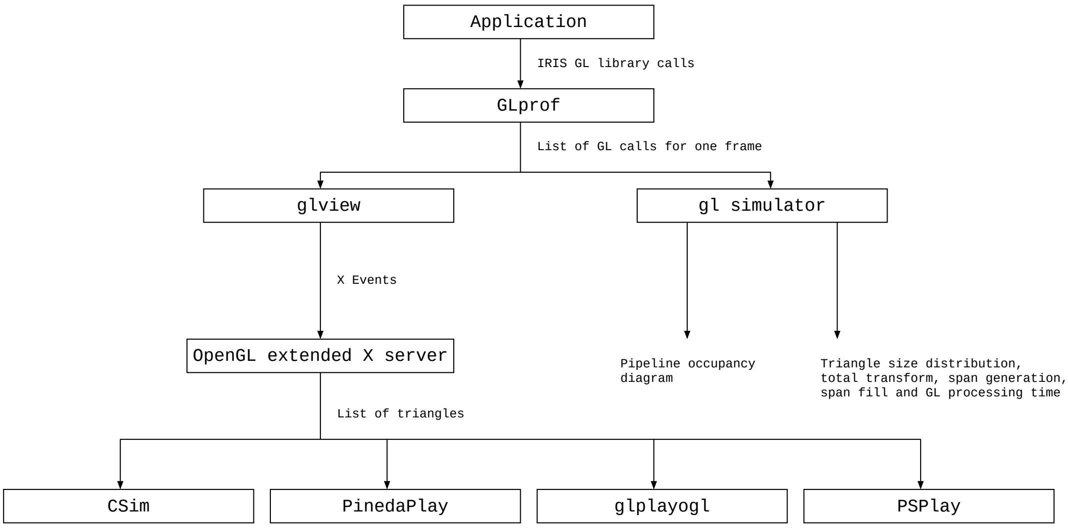

Figure B.4: Dataflow from application to simulators

B.3 Statistical processing

The list of triangles is in such a simple format that it is relatively easy to write programs to obtain statistics from the file. One program generates a list of the areas of the triangles from the file. This was used to generate the values for figure Figure 5.8.

B.4 GLPlayogl

When generating images from the triangle lists it was important to have some reference to compare the generated images with to check that they were correct. GLplayogl uses the OpenGL graphics library to display the scene - this provided an important aid to solving some simple errors which were made when initially creating the triangles file and later while debugging rendering simulators.

B.5 PSPlay

PSPlay displays the outline of the triangles, greatly magnified on a labelled grid (for example Figure B.5). At this level of magnification it is easy to see exactly which pixel vertices are covered (and thus which pixels should be filled). This was used as a debugging aid when writing PinedaPlay.

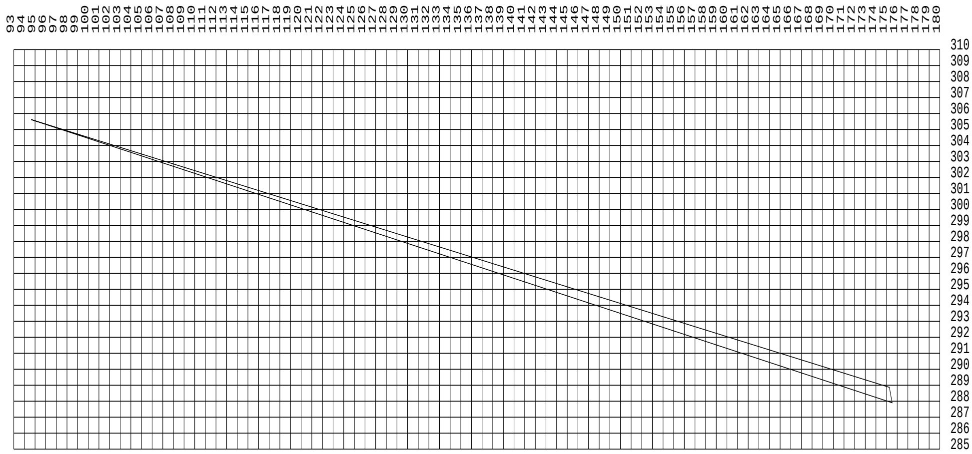

Figure B.5: Output of *PSPlay* for one very thin triangle

B.6 PinedaPlay

PinedaPlay displays the triangle file using Pineda's rasterisation algorithm **[Pine88]**. The output of *PinedaPlay* is an image file, a Z buffer file, statistics on depth complexity at each pixel and a polygon ID at each pixel.

B.7 CSim

CSim is a very simple, flexible discrete event based simulator written in C by the author for this research. CSim consists of a simulator core and a collection of 'components' which are joined via 'connections'. Components can place a value (either a simple integer or an entire C structure) on a connection at a given time - this combination of time, value and connection represent one event. Events are maintained on a time ordered queue by the simulator core. Connections are registered as either inputs, outputs or both input and output of any particular component.

At each stage in the simulation the current time is advanced to the time of the first event on the queue. Next each event at that time is examined. For each event at that time, the connection it references is set equal to the event value. As this is being done a set of components to which the modified connections are connected (as an input) is maintained. This is a true set in that no component is represented more than once. When all events have been processed in this way the input changed routine of each component in the set is called.

Each component is a collection of C functions which handle:

- initialisation

  Initialisation of data structures, registration of input/output connections and sub components.

- initial event generation

  This routine is called after initialisation of all components and is used by many components to initialise the value of their connections and possibly generate an initial event on some connections.

- input change

  The input change routine is called when any of the components inputs changes. The routine is called after the value of the connections has been updated. If multiple inputs change at the same time this routine is guaranteed to be called only one.

- finalisation

  The finalisation routine is used when the simulator is being shut down, either due to a lack of events or due to an explicit shut down request from one of the blocks. In this application the finalisation routine is often used to write out statistical information on the activities of the block.

One of the advantages of this simulator is that connections may hold arbitrarily complex objects in the form of C structures. In this application it is common for connections to hold entire triangles, spans or just simple integers.

For the task of simulating a simple rendering pipeline a number of components were created:

- A span generator

  The span generator takes axis aligned trapeziums, interpolates the end values and produces spans ready to be filled.

- A span filler

  This accepts the spans from the span generator and writes the generated pixels into a frame buffer. A frame buffer is held which stores for each pixel the colour, Z value, number of attempts at writing a pixel (Depth complexity), number of successful writes (after Z comparison), unique triangle number. The triangle number was initially stored as a debugging aid, but has since proved useful in the culling algorithm described in section **5.6**.

- A simple Processor

  A simple processor (and memory system) in the style of a Digital Signal Processor has been created to feed the span generator. In the present system being simulated the processor does some simple culling, splits the triangles into trapeziums and then calculates the values to be sent to the span generator. The processor is simulated as one component (with its memory system as a separate component). Internally to this component a full pipeline is simulated with interlocks between stages and protection against read after write dependencies on registers.

  Using the parser generator ‘bison’ a simple assembler has been written for the processor.

> The list of triangles to be rendered is fed to the simple processor via a simulated host which is mapped into the processors memory map. The host actually consists of another Unix process connected to the simulator via a pipe.

A FIFO is placed between the processor and the span generator, and another is placed between the span generator and span filler. The FIFOs are built as a separate component with an extra connection. This extra connection continually broadcasts an occupancy count which is used for gathering statistical information.

A 'trace' component was also built which monitors a number of connections and records any events (together with the time of the event) to a file. This file is in a format readable by '*Viewtrace*' (a part of the WorkView simulation package).

At the end of the simulation:

- The frame buffer dumps out all its sections into separate files (one for the image, another for Z etc.).
- The frame buffer writes information on the number of pixels written to a general information file.
- The processor writes information on instruction usage to the same information file.

The frame buffer code is shared with *PinedaPlay* (described above), and in this way utilities written to analyse the output files can be used on the output of either program.

The frame buffer saves its image in a raw format consisting of r, g, b triples (one byte between 0 and 255 each). This can easily be converted into a format such as 'ppm' which can be viewed using an image display program such as '*xv*'. The number of attempted writes to each pixel is also stored in a raw format - this information (after format conversion) was used to generate Figure 5.3.

B.8 AVS

AVS is a system for the visualisation of scientific data. Its main section consists of a network editor which allows blocks to be joined together in a type of flowchart representation. Each block represents one operation on a block of data (e.g. load data from file, display data, filter, convert to different data type). Figure B.6 shows a network which was used to display the saved information from the frame buffer. All the frame buffer information (except the image) is read in (using the ‘read field’ blocks) and the ‘attempted write’ field is displayed using an image viewer. This image viewer is connected to a block which allows the user to point to a pixel in the image viewer and display the value of all the fields at that point. Thus it is possible to point to a pixel and see the current Z value, number of successful and attempted writes and the polygon ID. This is extremely useful as a debugging aid.

Figure B.6: The *AVS* network used to display the Z buffer, triangle ID, and depth complexity information

B.9 Other utilities

A number of other small utilities were used to gather various pieces of information:

- countOcc

  Count occurrences of data values. When used on the triangle ID buffer this gives the visible area (in pixels) of each triangle.

- ImgDiff

  This takes as input two image files and produces another image file representing the difference. Each pixel of the output image is the absolute value of the difference between the pixels of the two input images. This was useful in checking that culling algorithms did not change the resultant image.

- xv

  xv is an image display program written by John Bradley which was used to display many of the images generated during this research.