Hacking Your Own Virtual and Augmented Reality Apps for Fun and Profit!

Tutorial Notes
Linux Conf Au 2004
Adelaide, South Australia

By Wayne Piekarski
wayne@cs.unisa.edu.au
http://www.tinmith.net/wayne

Wearable Computer Lab
School of Computer and Information Science
The University of South Australia

Abstract

Developing virtual reality and augmented reality applications has traditionally been very expensive due to the high cost of the equipment involved. In the last few years however, desktop computers have evolved to meet the demands of the gaming community and we now have cheap 3D graphics hardware and powerful processing power that rivals million dollar machines available only a few years ago.

This tutorial gives an introduction to some of the less mainstream possibilities that are available from the PC and web cam hardware currently sitting on your desk and currently existing free software toolkits. I will present some examples that perform real time 3D tracking of a user's hands and creating custom 3D input hardware. These applications are able to operate on mobile computers carried by a user with a head mounted display, but are equally useful on fixed desktops with monitors.

The purpose of this talk is to encourage developers to think "outside the box" by showing how it is possible to experiment and hack at home with exotic new input devices and user interfaces. I will cover some of the less used and understood subsystems available under Linux, including the DRI and OpenGL 3D interfaces in XFree86, accessing video cameras using Video4Linux and the 1394 Firewire drivers, using free libraries such as ARToolkit for 3D object tracking, and scene graph libraries for rendering 3D graphics.

One other important point to remember is that hacking doesn't always involve writing software, but includes opening up and modifying your hardware with power tools. You can make your own specialised input devices by chopping up existing cheap components such as USB mice. I will cover some of the possibilities for modifying existing laptops that you can wear on a backpack or within your clothing. Attendees will be able to inspect and try out the Tinmith backpack computer, a custom developed software and hardware system I have developed that provides mobile 3D modelling capabilities in outdoor environments.

License

1 Introduction

One of the guiding principles in this tutorial is to be "lazy like a fox". Most people are very busy and while it would be nice to spend a lot of time on doing something perfectly, we don't have a lot of time free but we still have projects we would like to build. This tutorial is designed to give a quick overview of useful technology that you can use to build projects at home right now, with the minimal amount of work possible. While some of these technologies are explained elsewhere, the documentation can be a bit lacking or non-existent. While some parts will be only an overview, I will focus in detail on the areas that have important tricks or features and may not be discussed in other areas. This tutorial is designed to give you useful tools so that you can begin hacking straight away. You can skip the many frustrating hours I have spent playing around with my computers and reading obscure documents and code trying to get things working.

While I try to avoid requiring too much previous knowledge, I will not teach things like basic C programming or OpenGL. These things are easily learned from many other excellent sources and so I want to focus my time on the more obscure areas. However, if you have any C programming experience then you should be able to follow what is happening and then you can catch up on things you missed after the tutorial.

We start the tutorial by briefly explaining how to configure a Linux distribution to contain the required packages that will be needed for development. Most modern distributions are suitable and contain pretty much everything you need but there are a few tricks (especially with Firewire) that I will go through that you must set up correctly and are poorly documented.

Next up is a detailed discussion of 3D graphics support with specific reference to Linux. While many books talk about OpenGL, there is almost no discussion of how 3D is implemented in Linux. I cover these important details and explain what the various libraries do and how they are used. I discuss in some detail the use of OpenGL to work around X11 limitations and its use for live video display. We finish off with a brief discussion of scene graph libraries and how they can be used to simplify application development.

The two video capture subsystems available under Linux are not commonly used by many, and have only small amounts of documentation that describe how they operate. I will go through in detail the use of V4L and Video1394 to perform video capture, including their various features, problems, and examples.

One of the most exciting uses of video capture is using it to perform real time vision tracking. I will demonstrate the use of a GPL library named ARToolkit which is able to extract full 3D information from paper markers. This library can be easily used to develop interactive 3D applications on a desktop without requiring expensive virtual reality equipment.

To finish off the tutorial, I will discuss some useful information for hackers who want to make their own hardware. Building your own hardware opens up a wide range of possibilities for exciting projects with your computers, such as attaching flashing lights and push buttons. I discuss the various interfaces available and how they can be used, including source code.

I look forward to talking to you all at Linux Conf Au 2004 in Adelaide, South Australia!

2 Base system install

In this tutorial we will work with some of the more obscure libraries available for Linux, and these libraries tend to be very dynamic and change often. As a result, I would advise you to install the latest distribution that you can get, and preferably including as many packages as possible. While it is possible to go and download each library as a tar ball and compile and install the sources yourself, this is quite time consuming and it is easier if a distribution can manage as much for you as possible.

2.1 Distributions

Previously I have used RedHat as my primary distribution as its installation was the easiest compared to many others. The problem with RedHat is that they do not tend to install the more obscure libraries that are available, and some of the packages available in previous releases are no longer available. Furthermore, older RedHat's did not have any network packaging capability such as Debian's APT, and so installing new RPMs was always a painful experience with unmet dependencies and so forth.

I have recently started testing Debian as a new distribution, and found that it was much more suitable for my development work because it has a massive number of packages – pretty much everything that you can ever think about is easily available. The APT program makes it very simple to download new packages and resolve dependencies, making building applications much easier. Debian is available in three versions: stable, testing, and unstable. The stable release is safe to use as it has been extensively tested, but the problem is that some of the libraries and tools are not the latest available. Later on we will talk about some of the problems with older libraries that are relevant to this tutorial. While it is possible to install the latest libraries from the testing and unstable releases, these have dependencies on other non-stable packages and so the system ends up performing a massive update when one package is installed. This results in a system which is no longer stable and for those who are not Debian gurus it can be a bit difficult to fix up problems that do occur. Another problem with Debian is that it tends to assume a reasonable amount of knowledge with Linux in order to configure the machine to your liking. Even though I have been using Linux since 1995 and am quite experienced, configuration files vary over time and between distributions, and I'd rather have something that makes simple tasks quick and easy to perform so we can focus on the real work.

2.2 Knoppix configuration

Based on the previous discussion, I have recently started using Knoppix as the base distribution for my work. The advantage of this distribution is that it is fully based on Debian, so it has all of the advantages of the package management system and the large amount of available software. However, Knoppix is designed for desktop end users and not system administrators, and so it comes with many of the tools I require preconfigured straight off of the CD. Knoppix is based on a mixture of testing and unstable packages which have been

selected by Klaus Knopper (the distribution's author) as being suitable for release, giving us a relatively bleeding endge system which is quite stable. Although it is designed to be used as a CD boot disk, it is easy to install to the hard drive and use just like any other distribution. So I have found that Knoppix is an easy way to get a base Debian system that is pretty much configured out of the box to do what I need with minimal work on my behalf. As an example, I installed Knoppix on my IBM Thinkpad with ATI Radeon 9000 and sound, video, and 3D acceleration worked straight away without having to do anything – this makes life much easier!

This tutorial is based on Knoppix v3.3 released in October 2003, although it should be relevant for most other distributions as well, particularly systems based on Debian testing or unstable. This tutorial however will give complete instructions for those using Knoppix, and you will have to work out the equivalents for your distribution. The standard Knoppix distribution comes with almost everything you need, but you will need to install a few extra packages to be able to build applications.

Coriander is a 1394 Firewire video preview program which is handy for debugging and controlling your cameras:

```
apt-get install coriander
```

The latest libDC and libraw development libraries are needed to compile your applications:

```
apt-get install libraw1394-5 libraw1394-dev
apt-get install libdc1394-dev
```

To develop your own OpenGL applications you should also install the GLUT toolkit:

```
apt-get install libglut3 libglut3-dev
```

To develop OpenInventor applications, you should install the latest COIN development libraries. The newer 40 version is better than the existing one because it contains support for loading VRML files:

```
apt-get install libcoin40 libcoin40-dev
```

Debian also comes with a nice program called auto-apt, which will attempt to install the correct libraries on your system automatically as you compile a program. Be warned though that sometimes it can install the wrong packages and mess up your system though, so carefully monitor the packages that it is going to install. To use it, simply install and use it:

```
apt-get install auto-apt
auto-apt update
auto-apt run make   # Run this command to build your program
```

2.3 Other configuration

If you are not using the recommended Knoppix system, it is still should be relatively simple to configure your system to get the right libraries installed. As a guide, make sure you have the following software installed in your favourite distribution:

Development tools (GCC, G++, make)

XFree86 (Server, libraries, and development files)

DRI support for XFree86 (3D support for your graphics chipset, GL libraries)

OpenGL libraries (GLUT, GLU, GL libraries and development files)

Kernel with all modules compiled (At least 2.4 is required)

Firewire support libraries (libDC, libraw, plus kernel modules)

2.4 Special Firewire configuration

For some reason, most distributions do not come with the Firewire devices (particularly video1394) properly configured. Also, there were two naming convention changes between kernel 2.4.18 to 2.4.19 and libDC v8 to v9. Before you begin using your system, if you are using libDC v8 I suggest that you upgrade it immediately. This version is quite old and has a lot of bugs when dealing with multiple cameras and extra features. Version 9 is much more reliable and so you really shouldn't be playing with the old library. The only catch is that distributions like Debian stable do not include v9, so you may have to make do with v8. Just be aware of it and if you have any problems make sure you either download new packages or compile up the source yourself.

In newer kernels 2.4.19 and later, the video1394 module uses character major 171 minor 16 devices, while older kernels use character major 172 minor 0. So make sure your devices are numbered like this accordingly. For newer libDC v9 code, the devices are stored in a directory /dev/video1394/* whereas in the older version only a single device is available at /dev/video1394. I have included a make_devices script with the examples which performs auto-detection, and also some common cases below:

Kernel 2.4.22 with libDC v9 (most common case)

```
mknod /dev/video1394/0 c 171 16
mknod /dev/video1394/1 c 171 17
mknod /dev/video1394/2 c 171 18
mknod /dev/video1394/3 c 171 19
```

Kernel 2.4.17 with libDC v9 (another common case)

```
mknod /dev/video1394/0 c 172 0
mknod /dev/video1394/1 c 172 1
mknod /dev/video1394/2 c 172 2
mknod /dev/video1394/3 c 172 3
```

Kernel 2.4.22 with libDC v8 (avoid this case)

```
mknod /dev/video1394 c 171 16
```

Kernel 2.4.17 with libDC v8 (avoid this case)

```
mknod /dev/video1394 c 172 0
```

The other devices such as /dev/raw1394 are typically included by default and have always worked for me without problems. More information on the firewire subsystem can be obtained from http://www.linux1394.org

3 3D graphics

One of the core things we will discuss in this tutorial is 3D graphics programming. Pretty much every example involves the display of graphics to the user, controlled via some kind of input mechanism. XFree86 is a graphical X display server used by Linux and many other operating systems to provide a standard way of drawing graphics to the display. Traditional X Windows programs are written as clients, where they run on a server machine somewhere and then send their display commands to an X server typically running on the user's desk. This architecture is very flexible and allows us to run applications remotely quite easily.

3.1 GLX and DRI

One limitation of the X design is that it always requires two programs to be running – the client and the server. Even when using shared memory or local communication mechanisms on a single processor, the client must still task switch with the server to render a display. Performing any kind of communication to another process involves the kernel performing switching and message passing, and both of these are very time consuming. When we need to render millions of graphical primitives to a display, the X protocol and kernel becomes a large overhead that prevents us from achieving the best performance. In these scenarios, being able to talk directly to the hardware is a must.

When Silicon Graphics (SGI) designed the 3D support into their workstations many years ago, they wanted to support the existing X protocol but extend this to support 3D primitives as well. They developed a protocol extension known as GLX which allows the encapsulation of 3D drawing commands over an X windows session. While GLX allows the transmission of 3D graphics over a network, there are overheads imposed by the network transport. One way to avoid transmitting the graphics commands repeatedly over a network is to use display lists, where the server caches geometry locally to improve performance. For geometry that is continuously changing however, direct hardware access is still needed.

SGI developed further extensions to their X server so that if it detected the client and server were on the same machine, it would allow direct access to the 3D hardware. The IRIX kernel was modified to allow user land applications to safely access the 3D hardware without risking system stability. The user land application then executes OpenGL function calls which are then used to directly write commands into the 3D hardware's command buffer. The video hardware then draws this to the display and there are minimal communication and switching overheads.

SGI later released some parts of the source code to GLX and their direct rendering extensions, which was then used as a foundation for the Direct Rendering Infrastructure (DRI) project. DRI is used to provide OpenGL direct rendering to hardware under Linux and XFree86, using both kernel modules to control access and XFree86 modules to provide the hardware interfaces. While SGI supported almost all of the OpenGL command set fully in hardware, most PC accelerator cards do not, instead relying on software emulation to fill in the gaps. The Mesa3D libraries are also integrated into XFree86 to provide a complete 3D rendering solution under Linux.

With GLX and DRI support under Linux, writing 3D applications using OpenGL is now very simple and easy to access by anyone with any PC and a cheap 3D accelerator card. The major difficulty is getting 3D support to work with your particular video card and distribution. The really nice part about the direct to hardware acceleration is that you can use it for writing fast 2D applications as well. Instead of using X server primitives, simply create an OpenGL window and do everything directly to the hardware using the driver features provided. A good example of this is supporting live video display: previously there were a number of extensions developed to X such as Xvideo (Xv), MIT shared memory extension (MITSHM), direct graphics architecture (DGA), etc, but none of them are standardised amongst all the drivers and still are not as efficient as direct to hardware under OpenGL. Since everyone building 3D hardware supports texture maps and OpenGL, it is a nice and portable way to write applications that are fast very easily.

3.2 3D hardware

Most 3D chipsets nowadays are quite powerful and able to pass the standard Quake3 test (ie, if you can play games on it then you should be okay). Before buying a computer or graphics card, it is a good idea to check out http://dri.sourceforge.net and http://www.xfree86.org to find out what the latest video card support is. Compiling your own XFree86 based on the source available is not for the faint of heart however, and so if you just want to get things working you just install a new distribution to get all the latest packages. If you want to keep things simple, take my advice and just install a new distribution and save yourself a lot of playing around to get it all working. If you install something like a latest Knoppix it will probably include the most recent XFree86 build as well. The other advice is that you can sometimes be better off buying a slightly older video card (don't get the latest bleeding edge one) because odds are the developers have had some time to get the drivers out for them. Since the card manufacturers do not typically provide drivers for video cards, the XFree86 developers can only begin thinking about a driver once the hardware is out.

3.2.1 Nvidia cards

Nvidia produce powerful 3D graphics hardware that is capable of performing a number of complex OpenGL functionality. While the TNT2 is a reasonably old 3D design, it is still very capable and able to run many of today's games at a reasonable frame rate and resolution. The TNT2 is great for most development and you can get them for free from people throwing them away.

The GeForce2 is a few generations ahead of the TNT2 and is capable of rendering much more complicated models. A nice feature of the GeForce2 is it is powerful enough to support real time video texture mapping – you can capture video data and load it into a texture in real time, and then render it onto a polygon. This ability is what separates cheap low end cards from the more expensive higher end cards. The GeForce2 is also a very cheap card to purchase as it is not very new, but still an excellent card. The thing to realise with 3D hardware is that if you have a quality card, you do not need the latest release to get good performance – they are all quite good and you only notice the difference on the most demanding of games. For everything else it does not matter. GeForce2s and above are available in many laptops as well, particularly larger desktop replacement units – I think this

is because they are not a very power efficient design, although Apple use them in most of their laptops though, so who knows.

Nvidia cards are interesting to run under Linux. Nvidia does not publically release any documentation about their cards, and so there are no GPL drivers in XFree86 to support the 3D acceleration features. However, Nvidia do provide their own binary 3D driver that reliably supports every 3D card they have ever made. The binary driver is wrapped up so that it can compile against whatever distribution and kernel you are using, and I have tried it under both RedHat and Knoppix with no problems. The binary driver shares much of its code base with the Windows Nvidia driver, and so the performance is excellent and all features of the card such as dual head are nicely supported on even the latest hardware. I have used the Nvidia drivers for a number of years and never had any problems with them, although you do read a few people on Slashdot complaining about them occasionally.

As stated previously, the drivers for Nvidia are not GPL and so there is no source code available. When I have presented at conferences previously, you always get one or two people who tell you off for using binary only code and so forth. I have been developing 3D applications for a number of years, mostly on mobile laptops, and only in the last year or two was it possible to get any decent 3D chipsets at all. So when we purchased our Dell 8100 laptop with 3D support, Nvidia was the only powerful hardware which had working drivers available. The equivalent Radeon chips still had very beta and buggy drivers, and were not useable for our work. The people who fund our work want to see demonstrations, and so we had no choice but to use the Nvidia drivers. In the next section I will discuss the current ATI cards though, which have now matured to a useable level.

3.2.2 ATI cards

ATI are the main competitors to Nvidia and produce 3D graphics hardware with similar capabilities. I do not have as much experience with using these cards since I have only started using them recently, so I cannot give as much advice on their usage. From what I have read, a Radeon chipset is slightly better than a TNT2, and a Radeon 7500 is slightly better than a GeForce2, so if you want to do intensive operations like live video texture mapping then go for at least a Radeon 7500 or above. Other older cards like Rage128 are many generations old and you should probably avoid these if possible, especially the Rage (Mach64) chipset which is even older and less supported. On desktops, Radeon cards are very cheap to buy, and Radeon chips are common in many laptops now too.

Getting a Radeon 7500 and up in a laptop will require a bit more looking around though, and be careful with your purchase. I have found many resellers will incorrectly tell you a laptop has a 7500 when in fact it does not. The best place to check is http://www.tuxmobil.org and read the feedback people post before purchasing.

The DRI project directly produces their own drivers for the ATI Radeon cards, supporting all models except for the latest bleeding edge hardware just released. My laptop (IBM Thinkpad T40p) uses an ATI Radeon FireGL 9000, and this worked straight off the Knoppix CD which is very nice and impressed me greatly. Please note that this is my first real experience with using Radeon hardware and so I do not know as much about them as their Nvidia counterparts. However, when the first generation of laptops arrived with 3D chipsets, the Radeon drivers were not sufficiently mature enough at the time to use for my mobile applications. The drivers seem to have improved a lot since then though and so I purchased

my new laptop with an ATI chipset. An interesting thing I noted when looking for a small and power efficient laptop is that most manufacturers tend to go with ATI for their laptops, while the power hungry and larger models tend to use Nvidia. This probably has to do with the ATI design being much more efficient in this area.

Recently it seems like there have been some problems with ATI releasing driver specs for their latest cards, with DRI only supporting Radeons up to 9200. I am not sure what is going to happen for newer cards than this. ATI have however released a binary only driver which is able to support Radeons from 8500 to 9800. I do not have any experience with this driver so I cannot comment on its stability, although from what I have heard I do not believe it is as well supported as the equivalent Nvidia driver.

3.2.3 Other cards

If you are not using one of the big two graphics chipsets (Nvidia or ATI) then support varies depending on the chipset that you have. While the chipsets by Intel that are integrated into motherboards are okay for simple 3D applications, they are not powerful enough to handle the live texture map example described previously.

If you are out to buy a video card for your computer, I would recommend you stick with Nvidia or ATI – a lot of people use them, lots of testing has been done, and you can get help on various forums. I would steer clear of other brands because they are typically much cheaper and more cut down.

Many years ago there used to be a project called Utah-GLX which was designed to provide 3D on very old cards such as ATI Rage and 3dfx Voodoo. This support works quite well on these chipsets, but the hardware is so old that it really isn't suitable for the work we will perform in this tutorial. If you have a desktop then I suggest you get another card like the ones recommended above, although if you have a laptop you don't really have a choice and so you can use this to get it working. I personally have used Utah-GLX with XFree86 v3.3 on an ATI Rage Mobility laptop and got it to work quite well with standard OpenGL applications, although the live video texture map slowed performance down to about 2 frames per second.

3.3 Programming OpenGL

This tutorial will not cover too much about how to program OpenGL programs themselves. We will focus on the Linux specific issues and leave you to find out more about OpenGL from other places. The best place to learn OpenGL is from what we graphics people call "The Red Book". Pretty much everyone I know who does GL coding learned how by reading this book, and it is very well written with many excellent examples.

```
OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL
Version 1.2 (3rd Edition)
By Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner, OpenGL Architecture Review Board
```

If you Google for 'opengl red book' then you will find many references to it, and there are even some online copies on the web that you can read yourself. There is another reference manual called "The Blue Book", but this one is just a dump of all the man pages and is not very useful. If you use Google or visit http://www.opengl.org/developers/documentation/ there is a lot of free stuff on the web you can use to lookup what the functions do. The red book also comes with a wide range of examples that are available online at http://www.opengl.org/developers/code/examples/redbook/redbook.html.

There are many free tutorials available on the web. While many of these are written by Windows people, they are mainly written to use the GLUT toolkit which means the source code is portable across most platforms, and so are still useful. Of all the web sites I've seen, probably the most useful is Nate Robbin's http://www.xmission.com/~nate/tutors.html site. Nate has built example tools which allow you to interactively tweak OpenGL function parameters to see what will happen without having to write your own demo application. These tools are very useful to work out what values to use very quickly and easily.

3.3.1 Libraries

When you write applications for OpenGL, there are a number of libraries which you may or may not use depending on what kind of functionality you need.

libGL is the core OpenGL functionality, with all functions named something like gl[NAME]. On the original SGI implementation, every one of these functions was fully implemented in hardware for optimised performance. This library provides primitive rendering, texture mapping, and matrix transformations. When you want to perform hardware accelerated rendering under X11, your driver (DRI, Nvidia, etc) will provide a complete libGL shared library object for you to use. If you do not have one then a software emulation like Mesa will be used.

libGLU is the OpenGL utility library and sits on top of the core GL functions, with all functions named glu[NAME]. There are some functions for assisting with camera placement, and to handle advanced functionality such as tessellating triangles. A triangle tesselator is a program that takes an arbitrarily sized polygon and breaks it up into triangles for you. I have found that the standard GLU tessellator functions provided by Mesa are very buggy and fail to successfully tessellate any kind of complex triangle. To get around this, I compiled up a copy of the SGI tesselator which has been released as open source, and this works much more reliably.

libGLUT is the OpenGL utility toolkit and sits on top of the code GL functions. While the GL functions allow code to render to the hardware, no provision is provided to configure a window on the display or interact with the rest of the window system. The GLUT library is very nice because it provides a set of commonly used functions for opening the display and handling user input. The GLUT library is available on most architectures (Linux, Windows, MacOS, etc) and so if you write your code to be portable and only use GL, GLU, and GLUT calls then it will work on any of these platforms. In general, if you are able to write your application using only GLUT calls, do it because it will save you a lot of grief. You can do everything yourself but it can be quite painful and there isn't too much documentation for it.

GLX is the interface that we use if we want to write OpenGL applications specifically for X servers. While the GLUT toolkit is nice, it cannot open multiple windows and does not provide the ability to access X server specific structures (because it is designed to be generic across platforms). So if you are going to write an application optimised for best performance with your own custom software architecture, GLX is definitely the way to go. Note that there are other ways of embedding OpenGL into your applications. GUI toolkits such as Qt, GNOME, and Motif all provide widgets that you can write OpenGL to. Note that these widgets were written using the GLX interface to fit with the rest of the toolkit. The strange part with GLX is that it is very hard to find much information about this, all the docs you read

either tell you to use GLUT or use an existing widget. I have included an example below which opens up an X window but also configures it for OpenGL drawing as well.

```
/* Global context information */
Display *__display;               /* Pointer to the X display connection */
Window __window;                  /* Handle to the X window to draw in */
int pixels_x = 640;               /* Width of the display in pixels */
int pixels_y = 480;               /* Height of the display in pixels */

/* Temporary variables needed within this function, but not kept */
XVisualInfo *vi;
GLXContext   context;
Colormap     cmap;

/* Open a connection to the X server */
__display = XOpenDisplay (NULL);

/* Handle the case of not being able to make the connection */
if (__display == NULL)
  gen_fatal ("Could not open connection to X server on path [%s] - check the DISPLAY variable",
XDisplayName(NULL));

/* Get a visual from the display which meets our requirements */
static int attribute_list [] = {
  GLX_RGBA,          /* We want a true colour visual */
  GLX_DOUBLEBUFFER,  /* Double buffering is required */
  GLX_DEPTH_SIZE, 1, /* Minimum one bit for depth buffer needed */
  GLX_RED_SIZE,   1, /* Minimum one bit per plane for RGB values */
  GLX_GREEN_SIZE, 1,
  GLX_BLUE_SIZE,  1,
  None };            /* This must be at the end of the list */

vi = glXChooseVisual (__display, DefaultScreen(__display), attribute_list);
if (vi == NULL)
  gen_fatal ("Could not get RGBA double buffered visual from the X server");

/* Now create a GLX context on the server - do not share lists (NULL), and use direct
   drawing with the server when possible (GL_TRUE) */
context = glXCreateContext (__display, vi, NULL, GL_TRUE);

/* Create a color map, we need to do this to create a new window */
cmap = XCreateColormap (__display, RootWindow(__display, vi->screen),
                        vi->visual, AllocNone);

/* Create a window which is the size we would like */
XSetWindowAttributes attr;
attr.colormap = cmap;
attr.border_pixel = 0;
__window = XCreateWindow(__display,
                         RootWindow(__display, vi->screen),
                         0, 0,
                         pixels_x, pixels_y,
                         0, vi->depth, InputOutput, vi->visual,
                         CWBorderPixel | CWColormap,
                         &attr);

/* Setup the window title */
{
  XTextProperty x_window_name;
  char *title = "WINDOW TITLE GOES HERE";

  XStringListToTextProperty (&title, 1, &x_window_name);
  XSetWMName (__display, __window, &x_window_name);
  XSetWMIconName (__display, __window, &x_window_name);
}

/* Configure the window to produce exposure events */
XSelectInput (__display, __window, ExposureMask);

/* Map the window to the display, waiting for it to appear before continuing, if we do
   not do this then our application may fail on slow or laggy X servers! */
XMapWindow (__display, __window);
while (1)
  {
    XEvent x_event;

    /* Wait for event to occur */
    XNextEvent (__display, &x_event);

    /* Check to see if event was Expose */
    if ((x_event.type == Expose) && (x_event.xexpose.window == __window))
      break;
  }
/* Reconfigure window to produce no events */
XSelectInput (__display, __window, 0);

/* Get the context, and make it the active one for OpenGL commands. This function
   allows us to control which window all the GL commands will go to. This allows
   us to have multiple windows if we wanted. */
glXMakeCurrent (__display, __window, context);

/* Add event listening to the connection so we know when things happen in the server */
XSelectInput (__display, __window, X_INPUT_MASK);

/* Flush everything out to the server now */
XFlush (__display);
```

```
XSync (__display, False);

/* Set up the graphics viewport - use the entire window allocated */
glViewport (0, 0, pixels_x, pixels_y);

/* Set the display to be 0,0 in the top left, and X,Y at the bottom right */
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluOrtho2D (0, pixels_x, pixels_y, 0);

/* No modelling transformations required for now */
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();

/* Continue with the rest of our OpenGL code now */
```

After this code you can then execute standard OpenGL commands. The next thing to do is flip the display when we have completed drawing because we are using double buffered mode. To do this, we have to make sure to tell GLX to flip over the buffers:

```
/* Swap the buffers – this does a flush automatically */
glXSwapBuffers (__display, __window);

/* Clear the new buffer for drawing */
glClearColor (0, 0, 0, 1.0);
glClear (GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);

/* Flush the GL pipeline to make sure any errors get picked up */
glFlush ();

/* Check for any errors that occurred */
GLenum glerror = glGetError ();
if (glerror != GL_NO_ERROR)
  gen_fatal (“OpenGL error code %d detected, graphics system has failed - %s”, glerror, gluErrorString
(glerror));
```

3.4 Live video display

OpenGL is a very powerful graphics library that is capable of performing pretty much everything you could ever imagine. Since you can perform anything Xlib can do, but with arbitrary transformations and viewpoints, you can write your applications in pure OpenGL like I have been doing lately. You can mix Xlib and OpenGL commands but you must ensure you flush the pipeline so they can be kept synchronised properly. Back to using OpenGL though, the Red Book goes into a lot of examples showing the kinds of things you can do with OpenGL, and so here we will cover an example of something a bit different and less mainstream.

As described previously, there are a number of X extensions for displaying live video, but there is no real portable interface supported by all video cards. OpenGL supports two different ways to send large amounts of pixel data to the card very quickly. The first method is to use the glDrawPixels() function call. This method takes in an image and maps it directly to the pixels on the display, and in theory should work quite well. In practice, this function causes the rendering pipeline to stall because it must flush all existing operations currently in progress. This function is not normally used and so hardware manufacturers typically do not optimise it either. One thing that is highly optimised is texture mapping however, and so this is the recommended way to draw images quickly in OpenGL. So we load the texture into the video card, and then draw polygons with the image as its texture. By drawing a square onto the display we can achieve an object that looks just like the texture was copied to the display, but the operation takes advantage of the optimised texture rendering hardware. Using texture maps has the following advantages over glDrawPixels and standard X windows methods:

- The application can send data directly to the hardware with no overheads
- It is much faster than glDrawPixels() because the pipeline is heavily optimised for it

- Images can be cached within the video card so when they are reused they do not need to be copied across the AGP bus again
- Any linear transformation such as scale, rotate, and shear can be performed in the hardware for free with no performance penalty
- Video data can be supplied in RGB, YUV, greyscale, or any other format and OpenGL will manage any conversions automatically
- Video can be mapped onto any 3D polygon, so you can have a virtual TV set with live video within a 3D world as you move around

So as we can see, using OpenGL for video display is both easy to use and very powerful. Applications such as MPlayer and ARToolKit have display code that is capable of using this rendering technique. One catch with using textures in OpenGL is that there is a restriction that the texture must be a power of two in each dimension. So if you have a 320x240 image from your camera, you must supply to OpenGL a texture image which is rounded to 512x256. By forcing this requirement OpenGL is able to accelerate texture performance further. While this limitation may seem difficult, the important thing to realise is that a scaled image like this is only required for the first frame! We don't really want to implement our own function to pad an image as each frame comes in from the camera, because we aren't gurus at writing optimised image handling code. Instead, OpenGL supplies a function glTexSubImage2D() which allows us to replace the existing image with a sub-image that does not have to be a power of two. This function then magically copies the image data over and fills it into the texture correctly, and we will assume that the people who wrote this function did a good job of it. It would be nice if all the texture functions didn't have this restriction but we will have to live with these decisions, I am sure they were made for good reasons. With the above exceptions explained, we are now ready to explain the process of mapping live video to a textured polygon. We will explain video capture in a separate section later on, that is another problem of its own.

Let us assume for this example we have a 320x240 input stream which is 24-bit RGB formatted. The first step is to perform a one time initialisation to get started, and then we will explain the rendering part which is repeated every time the frame is redrawn. I have written a demo program that draws some polygons with live texture mapped video, and it is included in the demo section of my area on the conference CD.

3.4.1 Initialisation

We will assume that everything else in OpenGL has already been configured before this point. The only major trick is to remember to turn on an appropriate texture mode somewhere during startup, so we do this:

```
/* Replace mode ensures the colours do not affect the texture */
glTexEnvi (GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

The first thing we must do is turn on texturing and then make sure we turn it off once we are done here because otherwise it will affect any other primitives drawn afterwards:

```
/* Tell the video card to turn on the texturing engine */
glEnable (GL_TEXTURE_2D);
```

We now need to create the padded buffer which is rounded to the nearest power of two (512x256). The contents of the buffer are unimportant but I have initialised it to all 0xFF which will make it white if rendered. This buffer is supplied to OpenGL so it can initialise

itself, and we can destroy it after because OpenGL makes a copy of the buffer rather than only a reference:

```
GLuint texid;
int width = 320;
int height = 240;
int round_width = 512; /* Power of two of width */
int round_height = 256; /* Power of two of height */
int bpp = 3;
int round_bytes = round_width * round_height * bpp;
char round_buffer [round_bytes];
memset (round_buffer, 0xFF, bytes);
```

Now we need to initialise a texture map in the video card's memory. Note that the card allocates a handle called texid that we will need to keep for later use when we want to draw with the particular texture. Whatever you do, make sure you do not run this code more than once per texture (ie, don't put it in the render loop) because you will run out of memory. Think of the glGenTextures() as a kind of malloc() call that needs to be freed up later:

```
/* Tell OpenGL the length of each row of pixmap data in pixels */
glPixelStorei (GL_UNPACK_ROW_LENGTH, round_width);

/* Allocate a new texture id */
glGenTextures (1, &texid);

/* Set the new texture id as the current active texture */
glBindTexture (GL_TEXTURE_2D, texid);

/* Set some parameters for the texture. We want it to be tiled and also to not perform any special
filtering to improve performance */
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Now we will load in the blank image we previously created. An interesting thing to note here is the use of GL_RGBA8 even though we specify GL_RGB as the format. What does this mean you may ask? The second GL_RGB is the format of the input image from the camera and tells OpenGL how to interpret the data we have supplied. The first GL_RGBA8 is the internal format to use on the video card however. GL_RGBA8 represents the image with transparency support and 32-bit padding internally in the video card memory, and gives the best performance and texture quality on hardware I have tested it on. You can use others but watch out for performance or quality problems:

```
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA8, round_width, round_height, 0, GL_RGB, GL_UNSIGNED_BYTE,
round_buffer);
```

Now we finish off and disable texturing and we can do any other initialisation work:

```
glDisable (GL_TEXTURE_2D);
```

3.4.2 New video frame

When a new video frame arrives, we need to capture the video frame, enable texturing, and then pass the image on to the video card:

```
/* Capture a frame of data */
int width = 320;
int height = 240;
char *in_data = video_capture_function ();

/* Enable texture mapping */
glEnable (GL_TEXTURE_2D);

/* Activate the previously created texid */
glBindTexture (GL_TEXTURE_2D, texid);

/* Load in the new image as a sub-image, so we don't need to pad the image out to a power of two.
   Note that some libraries return video in BGR format, so you may need to replace GL_RGB with
   GL_BGR if you get weird looking colours. */
glPixelStorei (GL_UNPACK_ROW_LENGTH, width);
glTexSubImage2D (GL_TEXTURE_2D, 0, 0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, in_data);

/* Disable texturing */
glDisable (GL_TEXTURE_2D);
```

For those of you who may be considering the use of threads to keep the video capture and render loops separate, be warned that this is dangerous. Most Linux libraries (including

OpenGL and Xlib) are not thread safe by default and you will cause big problems if you try to make calls from two threads simultaneously. The best way to implement threading is to have the frame capturing code copy the new images into a memory buffer, and then have the rendering loop be the only code which makes OpenGL calls. You can try to implement locking but these introduce performance overheads because either the kernel has to get involved, or you have to waste CPU cycles waiting for spin locks to clear.

3.4.3 Render

The render code may be run even when there is no new camera data available. The user might want to change their viewpoint and so we may need to refresh the display even if the video has not changed. To do this, we do a standard OpenGL texture render operation:

```
/* Turn on texture mapping */
glEnable (GL_TEXTURE_2D);

/* Activate the previously created texid */
glBindTexture (GL_TEXTURE_2D, texid);

/* Any polgons we draw from now on up to glDisable() will have the texture mapped to their surface */

/* Draw a square of size 1.0x1.0 with the video exactly mapped to fit it, taking into account
   the padding that was required to make the image a binary power of two in size. */
double scale_width  = width  / round_width;
double scale_height = height / round_height;
glBegin (GL_QUADS);
glTexCoord2f (0.0, 0.0);                          glVertex2f (0.0, 0.0);
glTexCoord2f (0.0, scale_height);                 glVertex2f (0.0, 1.0);
glTexCoord2f (scale_width, scale_height);         glVertex2f (1.0, 1.0);
glTexCoord2f (scale_width, 0.0);                  glVertex2f (1.0, 0.0);
glEnd ();

/* We can draw any other polygons here if we wanted to as well, this is good because we can keep
  the same texture loaded into memory without requiring a new one to be swapped in. */

/* Turn off texture mapping so that other polygons are rendered normally */
glDisable (GL_TEXTURE_2D);
```

3.5 Scene graphs

OpenGL is designed to render primitive shapes such as lines and polygons with texturing and lighting effects. OpenGL is a very low level graphics toolkit and while it is possible to write complete applications using it, the programmer has to supply much of their own code to handle common tasks. You can think of OpenGL as being equivalent to Xlib in terms of functionality, and most people do not write their applications directly using it. Instead they use a higher level toolkit such as Qt (KDE) or GTK (GNOME) which provides more useful functionality such as widgets.

The most powerful high level programming library for OpenGL would have to be Open Inventor, again developed by SGI. This library was originally designed back in 1992 to provide a more functional environment for programming complex 3D applications. It provided features such as a powerful object model in C++, a scene graph, user interface widgets, object selection, engines for animated objects, and a file format which forms the basis for VRML. Even though it is quite old, the design of the toolkit is still excellent and there is nothing that even compares in terms of functionality and design – the people at SGI who designed Inventor and GL really knew what they were doing.

To provide a more detailed description, a scene graph is a description of an environment that also contains relationships between various objects. To represent a human in a scene graph, the body of the human would be the root node, with the head, arms, and legs attached to it. The nose and ears are attached to the head, and the hands and feet are attached to the arms and

legs respectively. A scene graph is able to calculate transformations so that if you move the body, the rest of the parts will move with it. If you rotate the head for example, the nose and ears will move to match this motion. Scene graphs are very useful when representing complex mechanical systems and make rendering them very simple. Another bonus from using a scene graph is that the renderer is able to cache the results of previous runs to improve performance on the renders later on. OpenGL supports a number of features such as display lists and vertex arrays which can be used to improve performance.

3.5.1 Coin3D

SGI have recently released the source code for Inventor to the public, and groups have taken this code and cleaned it up. A commercial group have developed their own version of Inventor named Coin3D from scratch and released it as GPL code to the public. The free version may be used in GPL applications at no cost but commercial applications without source code require a license fee to be paid. A number of my colleagues who use Inventor all swear that the Coin version of Inventor is much better than the SGI sources (less bugs, less problems, etc) and so you probably should have a look at Coin first. There is a book called "The Inventor Mentor" by Josie Wernecke which describes all of the functionality of Inventor, but unfortunately other documentation seem to be a bit hard to come by. Coin have a complete generated set of documentation available on their web site, but it is more to be used as a reference and there are no examples. You can however download the collection of the examples from the Inventor Mentor book, and these are probably the most useful.

The Coin libraries have a number of nice features apart from the standard Inventor object collection. It has the ability to read in VRML files and natively support them in the scene graph, so you can use it as a generic VRML parser. Secondly, it is possible to use only the rendering code (and not the user interface widgets) and use Coin as your renderer in your application. This requires a bit of messing around but is possible and means you can use Coin instead of other libraries such as OpenVRML. When I was building VRML support into my application, I found that OpenVRML was not documented enough to be able to work out how to embed it into my existing scene graph, whereas Coin worked very easily.

To use Coin to read in a VRML file, use the following code:

```
SbViewportRegion *inventor_viewport;
SoGLRenderAction *inventor_render;
SoSeparator *inventor_root;


/* This is my special hacked callback function which forces the solid flag to be false to ensure that
   most of our VRML objects are rendered properly */
SoCallbackAction::Response adjust_vifs (void *, SoCallbackAction *, const SoNode *node)
{
  /* We can safely cast the object because Inventor wouldn't have put us here otherwise */
  SoVRMLIndexedFaceSet *vifs = (SoVRMLIndexedFaceSet *)node;

  /* Set the attributes we need - just turn off solid and that ensures we have no backface culling */
  vifs->solid = false;

  /* Done, tell Inventor its ok to continue */
  return (SoCallbackAction::CONTINUE);
}


void init_coin (char *infile)
{
  /* Initialise Coin */
  SoDB::init ();

  /* Create an Inventor container for everything */
  inventor_root = cnew SoSeparator;
  inventor_root->ref ();

  /* Control extra attributes of the object - make sure solids render right */
  SoDrawStyle *draw = cnew SoDrawStyle;
  draw->style = SoDrawStyle::FILLED;
```

```
    inventor_root->addChild (draw);

    /* The Coin documentation says we must use this combination to get two sided lighting and no backf
  ace culling */
    SoShapeHints *shape = cnew SoShapeHints;
    shape->shapeType.setValue (SoShapeHints::UNKNOWN_SHAPE_TYPE); // The Coin docs say this is important
    shape->vertexOrdering.setValue (SoShapeHints::CLOCKWISE);     // Clockwise ensures that cones and
  spheres render right
    inventor_root->addChild (shape);

    /* Now add a child to the root which will hold the VRML object */
    SoSeparator *container = cnew SoSeparator;
    inventor_root->addChild (container);

    /* Read in the specified input file */
    SoInput input;
    if (!input.openFile (infile))
      gen_fatal ("Could not open Inventor input file %s", infile);

    /* Parse the specified input file */
    SoSeparator *read_obj = SoDB::readAll (&input);
    if (read_obj == NULL)
      gen_fatal ("Could not parse Inventor input file %s", infile);
    input.closeFile ();

    /* Add the new object to our container */
    container->addChild (read_obj);

    /* Now we need to walk the tree and look for any VRML geometry nodes and force the backface culling off
       because the Inventor SoShapeHints does not control VRML nodes! */
    SoCallbackAction action;
    action.addPreCallback (SoVRMLIndexedFaceSet::getClassTypeId(), adjust_vifs, NULL);
    action.apply (inventor_root);

    /* Set up a viewport based on the display, I have no idea why we have to do this, but if
       you get the value wrong then the image does not render properly. However, if you do this
       then all is good and you have no problems, so I'm just going with it because it works */
    inventor_viewport = cnew SbViewportRegion (DISPLAY_WIDTH, DISPLAY_HEIGHT);

    /* Setup a rendering object for OpenGL */
    inventor_render = cnew SoGLRenderAction (*inventor_viewport);
  }
```

To get Coin to render the VRML object, set up your transformations correctly and then call the following code:

```
  void render_vrml (void)
  {
    /* Save rendering state because it could be changed inside OpenInventor */
    glPushAttrib (GL_ALL_ATTRIB_BITS);
    glMatrixMode (GL_MODELVIEW);
    glPushMatrix ();

    /* Render the Inventor scene here */
    inventor_render->apply (inventor_root);

    /* Restore attributes off the stack */
    glMatrixMode (GL_MODELVIEW);
    glPopMatrix ();
    glPopAttrib ();
  }
```

Make sure then any VRML files you supply have all the lights and viewpoints removed from them, otherwise Coin will try and reset these and will cause undesired effects that you have not planned for.

3.5.2 Other scene graphs

There are a number of projects that aim to provide a scene graph for use on top of OpenGL (or other 3D graphics libraries). Apart from Coin, there are a number of other libraries that provide this functionality. It should be noted that Coin provides more than just a scene graph, and can be used to write complete applications on its own. OpenVRML (www.openvrml.org) is designed to be a parser for VRML files so you can integrate it into your existing applications. I did not end up using this library because it was not as well supported as Coin and I could not embed an OpenVRML node within my existing scene graph – the examples I found were for an older version of the API. Others such as OpenSG (www.opensg.org) and OpenSceneGraph (www.openscenegraph.org) are also available, although I have no experience with either of these libraries.

If you are writing a complex 3D application, you should consider using a scene graph to help make your job easier. You could write your own like I did back in 1999 when there was nothing else you could use on a laptop, but this is painful and very tedious, and the result is usually not as good as what others have achieved.

4 Video capture

There are two ways of capturing video under Linux as of the time of this writing: Video4Linux (V4L) and 1394 Firewire (video1394). Video would have to be the weakest point in Linux right now, with little documentation and very low level hardware interfaces that make programming difficult. There is currently no frame work of similar completeness as something like DirectShow for Windows, and is an area where major development is required. In this tutorial I will attempt to demystify some of the features of video under Linux, and give some examples of how to use it. The conference CD includes a demo program which takes these code snippets and integrates them into a single example so you can see it all being used.

4.1 Video4Linux

Video4Linux (V4L) was the first API developed to provide a common interface for video capture devices. Most devices that support video under Linux such as PCI capture cards and USB cameras support this interface. Applications such as xawtv support V4L input, and mplayer supports the ability to write output to a V4L loopback device so other applications can read from it. The API is very simple and all processing is performed in the kernel, and there is no user-land library to support it. The API provides for open() and close() calls, and ioctl() is used to control settings and request image frames. If the camera does not provide the image in the right format (YUV instead of RGB for example) then you will need to implement a conversion routine yourself. So writing code using this API is very low level and "to the metal" programming, and you have to do a lot of playing around yourself if you need anything different done. V4L was originally developed based on the interface from the BTTV driver, and my understanding is that it was never really intended to be an extensible API for the future.

Recently, a new video API called Video4Linux2 (V4L2) has been developed to support more of a complete framework and fix limitations of the original V4L. The API has been under development for a while and is available in the 2.5 and 2.6 kernels. Applications written for V4L will still work under new V4L2 drivers because the older ioctls are still supported with a compatibility layer. It should be noted though that not all of the V4L drivers have been converted over to the new V4L2 API, and so if you write a V4L2 application it will not work if there is only V4L driver support. I have used the V4L interface for a number of years and will probably keep using it unless a new need arises. In this section we will focus only on V4L programming as it is currently the one which is the most supported.

Here is some example code which opens up a V4L device and prepares it for video capture operations. Note that lots of the code sets up things like channels, video formats, resolutions, etc:

```
#include <linux/videodev.h>

/* V4L data structures */
struct video_capability dev_info;
struct video_channel    channel_info;
struct video_picture    picture_info;
struct video_mbuf       mbuf_info;
struct video_mmap       mmap_info;
```

```
/* Actual memory map of video data */
char *memory_map;

/* Configuration variables */
int width, height, channel, fd;
double brightness = VALUE;
double contrast = VALUE;
double color = VALUE;


void main (void)
{
  /* We need to pass in the appropriate device name, depending on your distribution and configuration */
  open_v4l ("/dev/video");
  open_v4l ("/dev/video0");
}



void open_v4l (char *device)
{
  /* Open up the video device */
  fd = open (device, O_RDWR);
  if (fd < 0)
    {
      fprintf (stderr, "Could not open video device %s for read/write", device);
      exit (1);
    }

  /* Get the capabilities of the video source */
  ioctl (fd, VIDIOCGCAP, &dev_info);
  fprintf (stderr,  "Device = %s, Name = %s\n", device, dev_info.name);
  fprintf (stderr,  "Channels = %d, Width = %d..%d, Height = %d..%d\n",
           dev_info.channels, dev_info.minwidth, dev_info.maxwidth, dev_info.minheight,
dev_info.maxheight);

  /* Check the values to make sure they are sane */
  width  = dev_info.maxwidth;
  height = dev_info.maxheight;
  fprintf (stderr,  "Setting resolution to X=%d, Y=%d\n", width, height);
  if ((width <= 0) || (height <= 0))
    {
      fprintf (stderr, "Device %s width/height values (%d, %d) are not valid, must be positive", device,
width, height);
      exit (1);
    }

  /* Set the channel to the A/V inputs */
  channel_info.channel = channel;
  if (channel_info.channel >= dev_info.channels)
    {
      fprintf (stderr,  "Adjusting channel value from %d to %d to put it within the valid range\n",
channel_info.channel, dev_info.channels - 1);
      channel_info.channel = dev_info.channels - 1;
    }

  /* Grab the information for the above selected channel */
  ioctl (fd, VIDIOCGCHAN, &channel_info);

  /* Set the mode to PAL and print out debugging */
  channel_info.norm = 0; /* ----- Was 0 for PAL ----- */
  fprintf (stderr,  "Channel %d, Name = %s, Tuners = %d, Mode = %d\n",
           channel_info.channel, channel_info.name, channel_info.tuners, channel_info.norm);

  /* Set the channel to the one we want */
  ioctl (fd, VIDIOCSCHAN, &channel_info);

  /* Set the picture parameters */
  picture_info.brightness = int (32767 * 2.0 * brightness);
  picture_info.hue        = 32767;
  picture_info.colour     = int (32767 * 2.0 * color);
  picture_info.contrast   = int (32767 * 2.0 * contrast);
  picture_info.whiteness  = 32767;
  picture_info.depth      = 24;
  picture_info.palette    = VIDEO_PALETTE_RGB24;
  fprintf (stderr,  "Bright = %d, Hue = %d, Colour = %d, Contrast = %d, White = %d, Depth = %d\n",
           picture_info.brightness, picture_info.hue, picture_info.colour, picture_info.contrast,
           picture_info.whiteness, picture_info.depth);
  ioctl (fd, VIDIOCSPICT, &picture_info);

  /* Get memory map information */
  ioctl (fd, VIDIOCGMBUF, &mbuf_info);
  fprintf (stderr,  "Memory Size = %d, Frames = %d\n", mbuf_info.size, mbuf_info.frames);

  /* We need at least two frames for double buffering */
  if (mbuf_info.frames < 2)
    {
      fprintf (stderr, "%d frames is not enough to support double buffering, at least 2 is required",
mbuf_info.frames);
      exit (1);
    }

  /* Open up the memory map so we can use it */
  memory_map = (char *)mmap (0, mbuf_info.size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  if (memory_map == MAP_FAILED)
    {
      fprintf (stderr, "Could not mmap() %d bytes from the device %s", mbuf_info.size, device);
```

```
        exit (1);
      }

    /* Setup structure so the capture calls can use it */
    mmap_info.frame  = 0;
    mmap_info.width  = width;
    mmap_info.height = height;
    mmap_info.format = picture_info.palette;
  }
```

An interesting problem with V4L is that it does not seem to always support non-blocking I/O operations! I tried this with my CPIA based camera and any non-blocking calls will always block. You can open the device with O_NONBLOCK and try to change it with ioctl() calls but nothing works. This is unfortunate because it prevents us from writing single threaded applications that deal with many I/O sources – instead we have to use threads which introduce a lot of problems that we'd rather avoid if at all possible.

To start the capture process, we need to tell V4L to begin capturing frames and get ready for I/O operations (so this is still initialisation code):

```
void start_capture (void)
{
  /* Set to the first frame */
  mmap_info.frame = 0;

  /* Start the capture for the first frame */
  ioctl (fd, VIDIOCMCAPTURE, &mmap_info);

  /* Start the second frame as well */
  mmap_info.frame = 1;

  /* Start the capture for the second frame */
  ioctl (fd, VIDIOCMCAPTURE, &mmap_info);

  /* We will use frame zero as the start now */
  mmap_info.frame = 0;
}
```

The following code captures a frame into a buffer, and will block in the ioctl() until it is available:

```
void capture_next (void)
{
  /* Flip the frame values, the next operation will be on the next frame */
  mmap_info.frame = 1 - mmap_info.frame;

  /* Release the previously used frame returned in getFrame() */
  ioctl (fd, VIDIOCSYNC, &mmap_info.frame);

  /* Start this frame off for capturing the next frame */
  ioctl (fd, VIDIOCMCAPTURE, &mmap_info);
}
```

To retrieve a frame from the buffers that we maintain, we do the following:

```
char *get_frame (void)
{
  /* Return back a pointer to the current memory buffer */
  return (memory_map + mbuf_info.offsets [mmap_info.frame]);
}
```

So if we want to have a processing loop that reads in frames from the camera, we do the following:

```
while (1)
{
  /* Grab a pointer to a video frame */
  capture_next ();
  char *imgdata = get_video_frame ();

  /* Render the video data to the display */
  render_data (imgdata);
}
```

So this is the V4L API in action, it is relatively straightforward to use once you have an example to look at and modify. If you don't want to go to the trouble of writing your own code for this, an easier way can be to use the video library present inside the ARToolKit (we talk more about ARToolKit later).

4.2 1394 Firewire

Firewire is currently supported reasonably well under recent Linux 2.4 kernels and is able to handle devices such as hubs, hard drives, scanners, and even cameras. The cameras are the most interesting for this tutorial and so we will only talk about these. In the Firewire standard, there are two categories of cameras which are defined so that all cameras can be controlled in a consistent fashion without requiring new drivers. Digital Cameras (DC) is designed to control web cam type devices and typically operate at 640x480 with either YUV compression or raw RGB data. Digital Video (DV) is designed to interface with hand held video cameras and passes raw DV video which is recorded to tape. While the kernel supports generic 1394 protocols, it is the responsibility of user-land libraries to implement these other protocols using the base functionality. LibDC1394 implements interfaces to DC specification web cameras, while LibDV1394 implements interfaces to DV specification hand held video cameras.

At the start of this document I describe how to install the libdc code and some problems there are with older versions. It is very important that you put the latest 0.9 version on because it fixes many bugs that were present in the previous versions, and there were also some API changes. I have tested libdc with a number of cameras made by Point Great Research and ADS Technologies and it has always worked without any problems. I tried to use libdv on a Sony camcorder but was not able to get video that was not corrupted and have not tried to work much on it to investigate the problem. This tutorial will focus on DC specification cameras and the use of libdc.

Firewire cameras are much nicer than what you can achieve using USB1.0, because there is a lot more bandwidth available. The cameras are typically able to output resolutions up to 640x480, with frame rates up to 30 fps in YUV and 15 fps in RGB mode. The nice part about RGB mode is that the image data can be used directly from the camera without performing any extra image conversions at all, reducing the burden on the CPU further. So I have used Firewire cameras for the past two years for my research and the software support is much more reliable than the USB CPIA V4L driver which caused lots of kernel problems.

One problem I've noticed with Firewire is that if you continuously restart your application that uses a Firewire camera it will eventually just not find the camera and you need to restart the system. Rather than doing this, a much easier way is to reload the modules. I wrote a script called 1394-reset which performs this and will usually fix up any problems currently happening, as well as start up the modules if they aren't already running. See the attachments for a copy of this script, but you basically have to reload the modules in the correct order and with the right arguments:

```
rmmod raw1394
rmmod video1394
rmmod ohci1394
rmmod ieee1394
modprobe ohci1394      # attempt_root=1       (might be needed on some older 2.4 kernels)
modprobe video1394
modprobe raw1394
```

In some older kernels, there was a bug that required the extra parameters attempt_root=1 when loading ohci1394.o – newer kernels do not support this and seem to not have the bug any more. You may need to add this if you are having problems, and if you use the attached 1394-reset it contains the necessary logic to deal with this problem.

To test firewire cameras, there is a tool called gscanbus which draws a nice connectivity graph showing where all the devices are plugged into relative to the main 1394 controller.

Coriander is an excellent tool which you can use to debug DC specification cameras, and it will allow you to graphically adjust all the control and view live video streams.

Programming libDC is a bit of a nightmare however. I have managed to work out how to program everything else in this tutorial, except for libDC. There is no documentation covering the library at all, and the header files only have the barest number of comments. How anyone managed to write applications that use it is beyond me, because I've never found any info on it. However, the ARToolKit library discussed in the next section does provide an interface to libDC (as well as libDV) and so I ended up using this to integrate DC camera support into my code. ARToolKit is a modular set of libraries, with interfaces to V4L, libDC, and libDV, and a configure script to switch between them. The rest of the ARToolKit vision tracking code then uses these generic interfaces, and it is possible to easily use them from external programs. The video interface libraries are not thread safe and are not very clean, but you can write a wrapper around them to make them fit into the rest of your application.

To open up the video device and prepare for video capturing using the ARToolKit, perform the following steps:

```
/* Make sure ARToolkit/include is in your –I path to gcc */
#include <AR/video.h>
#include <AR/config.h>
#include <AR/ar.h>

/* Include internal ARtk structures */
AR2VideoParamT *artk_params;

/* Configuration variables */
int width, height;


void init_artk_capture (char *device)
{
  /* Get ARtoolkit to open up the device */
  artk_params = ar2VideoOpen (device);
  if (artk_params == NULL)
    gen_fatal ("Could not open up a camera device with ar2VideoOpen");

  /* Store in width and height info */
  ar2VideoInqSize (artk_params, &width, &height);

  /* Get ARtoolkit to start the capture */
  ar2VideoCapStart (artk_params);
}
```

To capture a frame of data and then retrieve it, perform the following steps:

```
char *get_video_frame (void)
{
  /* Use ARtoolkit to get the next frame started */
  ar2VideoCapNext (artk_params);

  /* Return back a pointer to the current memory buffer */
  return ((char *)ar2VideoGetImage (artk_params));
}
```

So this is a brief explanation of the libDC library and how it can be used in your applications. If you want to actually know how to program it, I would suggest that you have a look at lib/SRC/VideoLinux1394Cam/video.c which has the implementation inside ARToolKit. It is quite complex but if you want to study it, it is there.

5 Vision tracking

This chapter will talk about some of the possibilities for performing vision tracking using the previous capture and display code as well as some special purpose libraries which you may not know about. Vision tracking allows us to implement all kinds of neat applications without requiring any expensive hardware add ons.

5.1 ARToolKit

In the augmented reality research area, there is a commonly used vision tracking library named ARToolKit which has been used to implement a number of prototype applications. The ARToolKit was first released in 1999 by Hirokazu Kato and Mark Billinghurst at the University of Washington HIT Lab. The source code is released under the GPL so that others can easily deploy it within their applications and make their own improvements. The sources are also portable and work under Linux, SGI, and Win32 environments.

The ARToolKit libraries were developed to support the tracking of simple paper based fiducial markers. Applications can then overlay 3D objects over the top of these markers and then viewed on a display device. The toolkit generates a 4x4 matrix which includes both the rotation of the marker as well as its position relative to the camera that is capturing the scene. It is possible to find the coordinates of the camera relative to the marker by calculating the inverse of this matrix. The important thing to realise is that while you may not understand how a 4x4 matrix works, they are commonly used in many graphics rendering libraries (such as OpenGL) and so you can just copy the matrix from ARToolKit straight into OpenGL with no extra work. Note that in C you can represent a 4x4 array in row-column or column-row format, so a conversion may be required for this depending on the library you use.

5.1.1 Installation and compiling

If you have previously installed all the packages necessary for this tutorial, the installation should be a breeze. Simply download a copy of the toolkit (make sure you get the Linux version), then extract it into any directory you like. Next, run ./Configure which will ask which video capture library you want to use (it supports V4L, DC, and DV interfaces). After that, just type make and it should build without any problems. Note that the ARToolKit sources will be configured to use libDC v8 but if you have v9 installed like I recommended, you will need to install the fixed version of ARToolKit I have supplied.

The next step is to print out some marker patterns that the tracker knows about. Go to the patterns directory and print out at least pattHiro.pdf, and the rest of them if you want to play with all the demonstration applications. Once you have the toolkit compiled, you can run it by going into the bin directory and running the demo simpleTest. Take the pattHiro.pdf marker on paper and place it on your desk, and then point your video camera toward the marker. You should notice that once the marker is in full view the software will overlay a 3D model directly on top of the object. And that's it, we are ready to play with vision tracking!

5.1.2 ARToolKit internal operation

The ARToolKit is divided up into a number of separate libraries but provides a complete solution for video capture, vision detection, overlay, and display. This is nice because we have the ability to plug in and remove libraries as required, for example if we want to supply our own capture code or integrate the tracking output into our own scene graph. In demos which completely use the ARToolKit (such as *simpleTest*), the video is first captured by *libARvideo*. Next, recognition of fiducial markers and calculation of camera-space transformations is performed in *libAR*, which is then used to render the final scene using the camera calibration frustum in *libARgsub*. You can see all the various libraries that are available by looking in the lib/SRC directory:

```
AR                      Single marker recognition library (most apps will need to use this)
ARMulti                 Use multiple markers attached to an object to improve recognition
Gl                      Source code for libARgsub which are support routines for display of output
VideoLinux1394Cam               Capture library for 1394 based DC compliant web cameras
VideoLinuxDV            Capture library for 1394 based DV compliant video cameras
VideoLinuxV4L           Capture library for Video4Linux API under Linux
VideoSGI                Capture library for Silicon Graphics (SGI) machines
VideoWin32              Capture library for Win32 API machines (download other archive if you want
this)
```

5.1.3 Example code

The following is a cleaned up excerpt from the source code in examples/simple/simpleTest.c. Everything you need to know about ARToolKit is pretty much contained within simpleTest.c, so if the documentation (which is a bit out of date) does not explain enough then look in here. Unfortunately the comments are a bit lacking but it is straightforward to follow the code. In this excerpt, we will skip the initialisation part and look mainly at the loop which processes each frame and decides what to do with it:

```
/* main loop */
static void mainLoop(void)
{
    ARUint8         *dataPtr;
    ARMarkerInfo    *marker_info;
    int             marker_num;
    int             j, k;

    /* grab a video frame */
    if( (dataPtr = (ARUint8 *)arVideoGetImage()) == NULL ) {
        arUtilSleep(2);
        return;
    }
    if( count == 0 ) arUtilTimerReset();
    count++;

    argDrawMode2D();
    argDispImage( dataPtr, 0,0 );

    /* detect the markers in the video frame */
    if( arDetectMarker(dataPtr, thresh, &marker_info, &marker_num) < 0 ) {
        cleanup();
        exit(0);
    }

    arVideoCapNext();

    /* check for object visibility */
    k = -1;
    for( j = 0; j < marker_num; j++ ) {
        if( patt_id == marker_info[j].id ) {
            if( k == -1 ) k = j;
            else if( marker_info[k].cf < marker_info[j].cf ) k = j;
        }
    }
    if( k == -1 ) {
        argSwapBuffers();
        return;
    }

    /* get the transformation between the marker and the real camera */
    arGetTransMat(&marker_info[k], patt_center, patt_width, patt_trans);

    draw();

    argSwapBuffers();
```

```
}
```

The arVideoGetImage() call is used to capture an image frame using the ARToolKit capture library. You don't need to mess around with the previous capture examples if you do not want to. In fact, if you want to play with 1394 cameras (either DC or DV) I suggest you use the ARToolKit capture code because it is very tricky to write your own.

argDrawMode2D() configures the display for drawing and then argDispImage() then sends the image to the display. The display function uses similar texturing techniques that we described earlier, so if you are want you can use these functions without writing your own.

The next step is that arDetectMarker() is called which does vision detection on the image to extract out the marker features. This function is the main functionality of ARToolKit and can be quite intensive, so make sure you do not call this function unnecessarily. The thresh value is a cutoff used to determine the difference between black and white pixels on the markers and you may need to tune this for various lighting.

On completion of the vision tracking code, an array of markers is returned back in marker_info along with a count of the number of markers detected. The next video frame is then cued up using arVideoCapNext() and the code then continues to look at what markers were found. The code works through the list and finds which of the detected markers matched the expected template. The function arGetTransMat() is called which extracts out a 4x3 transformation matrix (this is similar to a 4x4 but has the bottom row cut off because it is always 0, 0, 0, 1) and stores the result into patt_trans. The final step is to call the draw() function which makes a call to argConvGlpara() to load this 4x3 matrix onto the OpenGL stack, and a cube is then placed on top of the marker. argSwapBuffers() is then used to flip the display over and make it active.

5.1.4 Quick and dirty 4x4 matrix tutorial

In 3D graphics, 4x4 matrices are used to represent both position and rotation of an object relative to some kind of origin. The matrix contains 16 values that should be stored with as much precision as possible (float or double) to ensure that the transformation is represented accurately. Firstly, a matrix that performs no transformations at all is called the identity matrix (I) and looks like this:

```
| 1 0 0 0 |
| 0 1 0 0 |
| 0 0 1 0 |
| 0 0 0 1 |
```

If you have an object and want to move it in the (X, Y, Z) direction from the origin, then you use a matrix that looks like this:

```
| 1 0 0 Tx |
| 0 1 0 Ty |
| 0 0 1 Tz |
| 0 0 0  1 |
```

If you want to scale an object around the origin along each of its three axes, then you use a matrix that looks like this:

```
| Sx 0  0  0 |
| 0  Sy 0  0 |
| 0  0  Sz 0 |
| 0  0  0  1 |
```

Performing a rotation about an axis is a bit more trickier, and you can use the following three matrices to rotate about X, Y, and Z:

```
c = cos(theta)
s = sin(theta)

 Rotate Z Axis      Rotate X Axis       Rotate Y Axis
| c -s  0  0 |     |  1  0  0  0 |     |  c  0  s  0 |
| s  c  0  0 |     |  0  c -s  0 |     |  0  0  0  0 |
| 0  0  1  0 |     |  0  s  c  0 |     | -s  0  c  0 |
| 0  0  0  1 |     |  0  0  0  1 |     |  0  0  0  1 |
```

If you look at the above matrices, you will notice the trend that the far right column is reserved for translations, while the top left 3x3 cells are reserved for rotation and scaling. While OpenGL understands these 4x4 matrices very easily, it is a bit more difficult for humans to interpret these values easily. If you want to build a quick and dirty tracking system that returns only the position of the marker relative to the camera, you can safely extract out the far right column and store the values separately. To do this, ARToolKit uses array[row][col] notation, so to extract out the position of the marker relative to the camera's coordinates, you can do this:

```
x = matrix [0][3]
y = matrix [1][3]
z = matrix [2][3]
```

To find the position of the camera relative to the marker's coordinates, you will need to invert the matrix (beyond the scope of this tutorial) and then perform the above extraction. If you want to do things like extract out the rotation angles from a matrix, this is quite complicated and I recommend that you search for Ken Shoemake's QuatLib, which appears to be included with another library you can search for called VRPN.

For more detailed theory on matrix transformations, please check out the book by Foley, Van Dam, Feiner, and Hughes. *Computer Graphics - Principles and Practice.* Addison-Wesley. This book has everything you will ever need to know on computer graphics, and more!

5.1.5 Uses for ARToolKit

There are a lot of interesting uses you can put the ARToolKit to. If you do a Google search for ARToolKit you will find links to hundreds of pages where people describe the work they have done with it, so it would be a good idea to go and check these out to get some ideas. The point of this tutorial is not to tell you what you can do with it, but more to show you how to use it so you can then go and develop really neat user interfaces and applications that we can check out at the next Linux conference.

For my Tinmith mobile modelling system, I have used the ARToolKit to track my hands and uses a set of gloves with metallic contacts and fiducial markers. The user can reach out and grab and manipulate objects in a 3D environment without requiring a keyboard or a mouse. For this all I did was use the ARToolKit to capture the location of the hands relative to the camera, and then feed the 4x4 transformation into my scene graph library. We have also built a simple tracking system that uses shoulder mounted cameras to look at fiducial markers placed on the ceiling to work out where in a room you are located. Both of these projects are described in detail (with papers to read) on my web site at http://www.tinmith.net.

5.2 Others

There are a number of other free vision tracking libraries that are also available for use in free software projects that may be of use to you. While ARToolKit is designed to be used for full 3D tracking, there are a number of other simpler cases such as tracking simple shapes which you may want to implement.

One library which you may want to investigate is Intel OpenCV, which is available under the BSD license. I have not used this library extensively, but the documentation describes in great detail some of the types of vision tracking that it can perform. It also has the ability to use compiled binary libraries from Intel (Intel Performance Primitives) which are optimised to use the various features of the latest Pentium processors. I believe these libraries are binary only because GCC may not be able to optimise its code in the same way that the dedicated Intel compiler can.

If you are interested in vision tracking you probably want to check out OpenCV, I have heard lots of vision tracking research people talk about it.

6 Hardware development

The most important thing about hacking to remember is that it doesn't just involve writing software. There are many other exciting things you can do with your computer as well, such as opening it up and making modifications with power tools. I am not just talking about making modifications to your case, but other cool things like making your own input devices and enhancing the things you already own.

6.1 Interfaces

There are currently a number of different interface standards for PCs that allow us to plug in devices to extend the functionality of a computer. This section goes through most of the common types of technology that are easily available to most of us. The focus here is on which interface is the most appropriate to use while still keeping things relatively simple. I will not describe things like building your own ISA or PCI card because this is beyond the capabilities of the average person.

An excellent reference site for all things related to interfacing with hardware is at http://www.beyondlogic.org – pretty much everything you could want to know is all located here.

6.1.1 Headers

For all of these examples, it is assumed that you have included the appropriate headers. You may need to tweak these slightly because they seem to vary across systems, and if your compiler can't find a function then look up the man page and see what header files it says are required.

```
/* Includes for open() */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/* Includes for close(), read(), and write() */
#include <unistd.h>

/* Includes for errors */
#include <errno.h>
#define error_string()  (strerror(errno))
#define gen_fatal(format, args...) fprintf (stderr, __FILE__, __LINE__, __FUNCTION__, format, ## args),
exit(-1)

/* Includes for sockets */
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

/* Includes for select */
#include <sys/time.h>

/* Includes for terminal/device stuff */
#include <termios.h>

/* Includes for directory entry stuff */
#include <dirent.h>
```

6.1.2 Parallel

Probably the simplest way to connect devices up to your computer is via the parallel port, commonly used for printing and available on pretty much all computers. The printer port contains 25 pins (DB-25), of which 8 are used for transmitting data along with another 4 or 5 used for sending flow control signals. Years ago before Ethernet became common you could buy "LapLink" cables that would transfer files between PCs. The original parallel port was designed to allow 8-bit output, but no input was supported and so you have to use the spare control pins to be able to read data back in. Newer enhanced parallel ports available on most PCs nowadays are bidirectional, allowing you to send and receive 8-bit data and use the control pins as well. A parallel port can achieve much higher speeds than a serial port but has no hardware assistance and requires the CPU to constantly control it.

The parallel port is nice because you can connect simple switches and LEDs directly up to the port without having to design much of a connection circuit for it. I have included in the examples folder a GPL'd program called ledcpumeter by Mike Romberg. This program allows you to hook 8 LEDs up to your parallel port and use the CPU load to vary the speed of the flashing. You can create cool case mod effects like Knight Rider style swooshing lighting for your computer. The parallel port is based on +5V voltages relative to ground, and many can be turned on and off under software control. Note that only a very limited amount of current can be drawn from each pin, otherwise you could fry your parallel port. The ledcpumeter shows that you can drive an LED off each pin, but any more than this and you will start to have random glitches and other problems. If you want to control devices with higher current draw, you should consider using an amplifier circuit to provide the current required. I will not describe amplifiers here because I have not used them; my applications have only required small amounts of current.

The main limitation of the parallel port is that it is very CPU intensive to operate. In order to receive data from the port, you can use either polling or interrupts. The polling method relies on spinning in a tight loop reading the status of the port and capturing data from it. Polling is very intensive on a CPU and will convert your shiny new Pentium-IV processor into a 486 because it must spend its entire time checking for data and if other applications are running we might miss data from the port. The alternative method is to use interrupts, so that when data arrives the kernel schedules the application to read the incoming byte. The kernel has large overheads for processing interrupts, and so if you need to read a couple hundred or thousand bytes per second then the machine will also grind to a crawl. If you really need to use the parallel port, you should consider using something like Real Time Linux or even DOS which does not impose as many overheads and is designed for these tasks. The standard Linux kernel is not really designed for these kinds of operations and there are ways we can avoid using the parallel port. So use the parallel port for tasks you will only do occasionally.

Writing data to the parallel port is quite easy, but requires us to directly access the I/O address of the printer port. Make sure you are running the parport.o module and find the address that the port is located at. Parallel devices are typically named /dev/lp0, /dev/lp1, and so forth, and you can find out their address ranges by looking in the file /proc/ioports. Typically, ports are located at addresses 0x3bc, 0x378, and 0x278. Before you can write to the port hardware, you must get permission from the kernel otherwise it will terminate your application because you normally can't do this. Windows 2000 and XP completely prohibit these kinds of operations (even as the administrator) and you must write a kernel level driver to support this access.

Under Linux you can write a user land application for simple hacks, and if you get serious you may want to consider writing a kernel module (especially if security is a concern).

To play with the parallel port, we do the following simple code steps:

```
/* Get permission to do I/O operations from the kernel */
/* ioperm (base_port, number_of_bytes, on_or_off_flag) */
if (ioperm (0x378, 3, 1) != 0)
  generate_error ();

/* Write some bytes to the port now */
/* outb (value, base_port) */
outb (0xFF, 0x378);
```

6.1.3 Serial RS-232

The RS-232 serial port is another commonly available port available on most computers these days. The connector can be either 9 or 25 pins (DB-9 or DB-25) and is capable of sending data in both directions one bit at a time. On most modern PCs with good serial buffering, transfer rates of up to 115,200 bps are possible. I prefer using the 9-pin connectors because they have the same capabilities as the larger 25-pin versions but are a lot more compact and easier to work with. When implementing a serial cable, you must connect pin 5 (ground) on both ends and appropriately join up pins 2 and 3 which are the transmit and receive lines. To build a cable that connects two computers together, you would do 5-5, 2-3, and 3-2. For an extension cable you should do 5-5, 2-2, and 3-3 instead. To support flow control, the appropriate pins must also be connected to pass this information on. You can generally avoid this because most simple devices that we will build do not support it or run so slow that it is not required. So all of the cables that I use for my work only have 3 wires and you can make things a bit more light weight this way. For more information on making up more complex serial cables, do some Google searching for schematics or go to the Beyond Logic site mentioned earlier.

While serial ports transfer data at a slower rate than parallel ports, they are much easier and more efficient to program for. PCs use internal chips called UARTs which are responsible for taking data and converting it into a stream of bits for the serial port lines. The CPU does not need to concern itself with controlling the signals directly, since the UART handles this in the background. Very old serial UARTs available in the days of the 486 and earlier had very small buffers and required constant attention from the CPU to ensure that the UART was always provided with data. This meant that if you tried to run under a multitasking system with an old UART you could get errors. Newer machines use a 16550 UART which supports a much larger buffer so that the CPU can do other things and clear the buffer less often. The benefit of bigger buffers is that the CPU is interrupted less often, and can grab larger chunks of data when it does happen. They are also more friendly with non-realtime operating systems such as Linux, which thrive on using big buffers to give flexibility in scheduling tasks. So from a CPU and operating system point of view, serial ports are definitely the preferred way to interface hardware to your machine.

The problem with serial ports is that they require more logic in your hardware device so it can communicate with the computer. While parallel ports allow you to directly connect components to the pins, with serial you will have to add something in between to collect the RS-232 protocol signals and reconstruct the original bytes. The RS-232 standard also uses -12V to ground which makes it more difficult to generate without some kind of dedicated hardware to support it. I have found the easiest way to build serial port circuits is to use a Basic Stamp II microcontroller. You can write very simple basic like programs and it uses a

compiler to write the necessary machine language and upload it onto the chip. The BS2 is not very fast and are a bit pricey, but everything is integrated onto a single chip so all you need to do is apply power and you are ready to go. Unfortunately the development tools are Windows only, but are very simple to use and require no assembly language or hardware experience. I managed to learn how to use a BS2, write useful apps, and build a working hardware device in a day – that's how easy it is. Other more adventurous types may wish to investigate the usage of other microcontrollers but you will need to use assembly language or low level C code, and acquire the appropriate hardware to flash your images over to, and build a development board to supply necessary regulators and glue logic. Programming for BS2s is a whole area in itself and there are tons of docs and examples available on the Internet via a Google search.

The Tinmith system provides gloves that are used to control the system. The finger presses are detected by a BS2 sending +5V pulses up each finger tip to see if they are pressing against other surfaces. The BS2 listens to see if the +5V signal is visible on the palm or the thumb pads, and if so it then transmits a single byte via the serial port to the host computer. It performs this test in a tight loop and can poll each finger on both hands more than 30 times per second, which is adequate for my application. The beauty of using a microcontroller is that tight an inefficient loops can be offloaded from the main CPU into a small and inexpensive device designed specifically for the task. The BS2 runs for days on end from a single 9V battery to perform a task that would max out a Pentium CPU and a parallel port. By sending serial characters, the CPU is only bothered when something like a finger press happens, which happens only very rarely when you are a computer.

Programming a serial port requires a bit more setup than a parallel port, but the kernel provides nice abstractions so that you can treat the serial port more like a file. The kernel provides devices /dev/ttyS0, /dev/ttyS1, /dev/ttyS2, etc which map to COM1 to COM3 under DOS. If you have USB based RS-232 serial ports they may map to the existing device naming or an alternative /dev/ttyUSB0, /dev/ttyUSB1, etc (more on these devices later). By having a file device we can use the standard read()/write() style operations that we are used to dealing with normally. When dealing with serial ports, we need to decide on a baud rate (300 – 115,200 bps) and the number of bits (7 or 8) in each character sent. The next thing we need to think about is the type of interface we want. By default, the kernel mangles lots of the characters on the serial port, and waits for lines delimited by carriage returns as an extra buffering mechanism. The default values for a serial port are confused and varying, so the safe bet is to reprogram all the values on the port to known sane values so that you do not get any surprises when you move to another machine. There is an FAQ on programming the serial port under Linux but it can be a bit hard to understand if you are not familiar with all the terminology used such as cooked mode, canonical mode, etc. The example code below will fully configure a serial port for reading and writing, turn on non-blocking mode, and allow you to chose between character or line based buffering:

```
/* Set these variables to configure the code */
int baud = 38400;       /* Set for 38400 baud */
int bits = 8;           /* Set for 8 bits per byte (7 is also possible) */
bool line_mode = false; /* Set for single byte mode, there is also line based reading too */

/* Open the device with non-blocking and no controlling TTY attributes */
fd = open (device, O_NONBLOCK | O_NOCTTY | O_RDWR);
if (fd < 0)
  gen_fatal ("Could not open the serial device %s with attributes set - %s", device, error_string());

/* Work out the constant to use for the baud rate from the input value */
int baud = -1; /* Keep the compiler happy, it can't seem to detect that this
                  value is initialised correctly */

switch (baud_rate)
  {
  case 300:    baud = B300;  break;
```

```
    case 1200:   baud = B1200; break;
    case 2400:   baud = B2400; break;
    case 4800:   baud = B4800; break;
    case 9600:   baud = B9600; break;
    case 19200:  baud = B19200; break;
    case 38400:  baud = B38400; break;
    case 57600:  baud = B57600; break;
    case 115200: baud = B115200; break;
    default:
      gen_fatal ("The baud rate value %d given is not a supported serial port rate", baud_rate);
      break;
    }

  /* Decide on a bit flag */
  int bitflag;
  if (bits == 7)
    bitflag = CS7;
  else if (bits == 8)
    bitflag = CS8;
  else
    gen_fatal ("The number of bits %d must be 7 or 8", bits);

  /* Reprogram the serial port of the device - do it differently depending on the mode */
  struct termios new_serial;
  clear_memory (&new_serial, sizeof (new_serial));
  if (line_mode == true)
    {
      new_serial.c_iflag = IGNPAR | ICRNL; /* Ignore parity errors, convert cr -> nl */
      new_serial.c_oflag = 0;              /* Raw output */
      new_serial.c_lflag = ICANON;         /* Set canonical mode (line based) */
      new_serial.c_cflag = bitflag | CREAD | CLOCAL | baud; /* Enable 8-bit, no rts/cts */
    }
  else
    {
      new_serial.c_iflag = IGNPAR;         /* Ignore parity errors */
      new_serial.c_oflag = 0;              /* Raw output */
      new_serial.c_lflag = 0;              /* Set no processing */
      new_serial.c_cflag = bitflag | CREAD | CLOCAL | baud; /* Enable 8-bit, no rts/cts */
    }

  /* Flush away all data not yet read or written */
  if (tcflush (fd, TCIOFLUSH) < 0)
    gen_fatal ("tcflush() call on port %s, fd %d failed - %s", device, fd, error_string());

  /* Set the serial port to use the specified settings */
  if (tcsetattr (fd, TCSANOW, &new_serial) < 0)
    gen_fatal ("tcsetattr() call on port %s, fd %d failed - %s", device, fd, error_string());

  /* Debug */
  Fprintf (stderr, "Serial port %s, fd %d reset to baud %d, line %d", device, fd, baud_rate, (line_mode ==
  true));
```

Non-blocking mode allows us to check if there is data on the serial port even when nothing is available. Normally in Unix, if you read from a device but there is no data available the kernel will not return control back to you until something arrives. This is nice if you are waiting on the console with “Press any key to continue” but not useful if you have other useful work to be done instead. The above code is set for non-blocking but you can stop this by removing the O_NONBLOCK flag. Also note that I have disabled all flow control so if you need this you will need to put in the appropriate flags in c_cflag to activate it.

When using this code, you will need to use read() and write() calls to interact with the serial port. Using the standard I/O library (stdio.h) is not advised because it adds extra buffering that you probably don’t want. If the buffer in the kernel contains no data when you read, or is full when you try and write, the kernel will return an EAGAIN error in non-blocking mode, and block otherwise. Here is an example of some code that reads the serial port with non-blocking support:

```
  char buffer [1024];
  int result = read (fd, buffer, 1023); /* Leave space for a terminating \0 at the end */
  if (result < 0)
    {
      if (errno == EAGAIN)
        {
          /* No data available, go and do something else */
        }
      else
        {
          gen_fatal (“Could not read from serial port with an error - %s”, error_string());
        }
    }
  else
    {
       /* Clean up the buffer by adding a terminator at the end. If we don’t add this we could seg fault
          the program and potentially introduce a security hole! */
       buffer [result] = ‘\0’;
```

```
        /* Print out the final result now */
        fprintf (stderr, "Data package of [%s] with %d bytes received\n", buffer, result);
    }
```

Sometimes you may want to quickly dump out the serial data arriving on a particular serial device but you don't want to have to write a C program to do it. By default if you just run cat on the device name you probably won't get anything, unless a previous application has left the serial port set up for you. Here is a simple shell example you can use for many devices that don't require flow control:

```
# Set this to your device
DEVICE=/dev/ttyS0

# Set this to the baud rate
BAUD=38400

# Configure the port with the settings
stty raw –crtscts $BAUD < $DEVICE

# Sit forever waiting for new data to arrive
cat $DEVICE
```

6.1.4 USB serial

Many modern computers nowadays are coming out with only "legacy free" connectors such as USB and Firewire. Parallel and serial connectors are a very old and clunky way of attaching devices to a computer, with inefficient CPU controls and slow transfer rates. They are also not expandable, and so once you have plugged devices into your ports there is no way to add new ones without putting in another ISA or PCI card into your machine. If you have a laptop then you are particularly stuck because you can't add extra cards at all. This was particularly a problem with my old mobile backpack systems, which included only a single serial and parallel port.

I firstly experimented with PCMCIA cards that provide serial ports, which seem to have excellent driver support under Linux since v2.2 (the cards that I have tried anyway). The problem with these cards is that they are expensive, very fragile and break easily, and that there are only two PCMCIA slots available. When you run out of ports you are back to the old problem of before.

The really elegant solution to the problem is to use USB. I have been waiting for something like USB to appear for years and now that it has arrived it has solved all of my interfacing problems with the mobile backpack. The best part about USB is that you can use hubs to connect up as many devices as you want (127 actually). While a PC should make available about 500 mA of current at 5V, this runs out quickly when you add devices and so you can add external power supplies if needed.

The first neat hack you can do with USB is leach power off the bus. I have seen lots of devices do this such as USB powered fans, coffee mug warmers, and phone chargers. One project I used this for was to USB power an Ethernet hub so I can take it travelling with me without bringing along a power supply. The first trick is to make sure that you get one which runs off 5V directly, otherwise you will need a converter which is a hassle. I then cut off the USB cable from an old mouse and found the two wires that supply the power (USB has four wires – one for power, one for ground, and two for bi-directional data). Once you have the two wires you then just add a plug and voila it can now power the Ethernet hub. So USB provides a nice way for distributing power and data from a single source, whereas before you needed to run them separately with all the problems described before as well.

The easiest way to take advantage of USB is to connect all your legacy devices up with converter dongles. Be warned though that all dongles are different and there are no standards for how a serial port converter must be implemented. If you are buying a dongle to use under Linux, you must be very careful to ensure you get the right one otherwise it will not have any drivers. I can recommend the Keyspan series of RS-232 adaptors and have used them for a number of years. Hugh Bleming wrote an excellent kernel driver for them with documentation provided by the manufacturer, so it would be good to support a company that provides information to developers. Keyspan make devices that have one, two, or four serial ports integrated into a single unit, making it easy to connect up a large number of devices while using a minimal number of hub ports.

Another interesting device is the FTDI FT8U232AM chip, which provides a serial interface but as a single chip solution. This device is designed to be integrated into existing devices and USB enable them with a minimal amount of work. Rather than redesigning your circuit around a new microcontroller, you can simply add this on with minimal changes. One catch to the device is that it uses TTL (0 to 5V) rather than RS-232 (-12V to 0V) and so a Maxim MAX232 converter or similar chip is also needed to use it. We are currently investigating the use of these chips in our backpack to help miniaturise some of the components and cut down the amount of cabling by using USB to power the devices where possible. The nicest part about this chip is that it is fully supported with a Linux kernel driver as well. I have also seen some pre-packaged USB to serial converters in the shops which had FTDI visible through the clear case, so these should also work. More information on the FTDI chips can be found at http://www.beyondlogic.org/usb/ftdi.htm.

FTDI and others also make parallel interfaces (such as the FT8U245AM) but I have not used these and I cannot comment on if they are useful for controlling external devices directly such as the ledcpumeter program.

6.2 Cheap hacks

The best circuit designs are the ones where you don’t have to do much work. There are many cases where you can completely avoid building your own circuit and instead can just modify something you already have. For example, lets say you want a device such as a handheld plastic gun which contains a trigger as well as some buttons on the side. The easiest way to implement this is to use an existing USB mouse (which are incredibly cheap) or you may even have one which has broken wheels inside making it useless. Simply open up the mouse and throw away all the plastic. Then desolder the micro-switches and run wires off to the switches you provide to control the device instead. So within 10 minutes (most of it desoldering) we have turned a mouse into a perfect USB button box. The next step is to buy something like a toy plastic gun or whatever physical prop you want, and open it up. Now you can glue the circuit from the mouse into the gun, embed switches into the casing, and now you are all done. Reading the plastic gun in software is really easy, because it will appear just like a mouse and so you can use it with all your existing games that already can handle mouse input. This method does have limitations for some cases however. If you want to have more buttons than the mouse supports (typically 3-5) then you will have to think of something else. If the mouse is also being used to control say a GNOME or KDE desktop, button presses from the gun will be mixed up with the real mouse device. While it is also possible to address each USB mouse as a separate device, this becomes a bit tricky especially if you are trying to use it

in someone else's existing application. In these cases, it is probably best to look to one of the previous options.

6.3 Other interfaces

One of the problems with the interfaces provided by modern PCs is they are becoming harder for the average hobbyist to interface to as their speed increases. Standards such as Firewire and USB2.0 require very careful attention to detail even when creating cables to avoid building up interference. In order to communicate the complex amounts of information required, specialised chipsets are typically used but these are complicated in themselves. These chipsets are normally used in the millions by designers at large companies rather than hobbyists who would only buy in quantities of one or two.

For interfaces that are internal to a PC, an ISA card (the original PC interface from the XT) is probably the simplest interface and yet is still quite difficult. The next problem is that this bus has been phased out of new computers and so it will become harder to find motherboards that actually support it. The reason that developing ISA cards is complicated is because you must build a circuit to interact with the bus, and if you make a mistake then you can either destroy your motherboard or at least crash your machine. The next step is to write a kernel level driver to interface to the card to allow user applications to access it. These things are all quite difficult and if you can avoid it by using a serial or parallel port, then for simple applications you should avoid it.

7 Conclusion

In this tutorial, we have covered a wide range of topics that are useful for people who want to hack up their own interactive 3D environments at home. We worked through the following topics: configuring a distribution, OpenGL implementation under Linux, using OpenGL to display video, capturing video using both V4L and video1394, performing vision tracking using ARToolkit, and developing custom hardware devices as interfaces. While I gave a few examples of things that I have built in the past, the next step is for you to go forth and take this knowledge and build other cool devices. You will have to think of your own ideas though, and I look forward to hearing about them on Slashdot or the next LCA in the near future some time. If you are interested in research work then perhaps consider studying at university or if you've done that then consider postgraduate education where you can apply your knowledge to developing really interesting new ideas. So have fun with your computer and explore some of the exciting possibilities that are available to you.

Good luck and don't fry your computer!

regards,

Wayne

8 References

Wayne Piekarski Home Page
http://www.tinmith.net/wayne

Project Tinmith
http://www.tinmith.net

School of Computer and Information Science, University of South Australia
http://www.cis.unisa.edu.au

Wearable Computer Lab, University of South Australia
http://wearables.unisa.edu.au

ARQuake Project
http://wearables.unisa.edu.au/arquake

DRI Project
http://dri.sourceforge.net

XFree86
http://www.xfree86.org

Linux on laptops
http://www.tuxmobil.org

OpenGL documentation
http://www.opengl.org/developers/documentation

OpenGL red book examples
http://www.opengl.org/developers/code/examples/redbook/redbook.html

Nate Robbin's OpenGL examples
http://www.xmission.com/~nate/tutors.html

Coin Inventor
http://www.coin3d.org

OpenVRML
http://www.openvrml.org

OpenSG
http://www.opensg.org

OpenSceneGraph

http://www.openscenegraph.org

Beyond Logic hardware interfacing
http://www.beyondlogic.org

ARToolkit home page
http://www.hitl.washington.edu/artoolkit

Mike Romberg's LED CPU Meter download
http://www.ibiblio.org/pub/Linux/system/status