Image Processing with OpenGL and Shaders

Nov 22, 2010 By George Koharchik
in Graphics HOW-TOs

Using OpenGL and GLUT, you can increase the speed of your image processing by using the power inside your system's GPU.

Video games have been making full use of GPUs for years. Now, even nongraphical products (like Matlab) are starting to take advantage of the number-crunching abilities of GPUs. You can do the same.

This article discusses using OpenGL shaders to perform image processing. The images are obtained from a device using the Video4Linux 2 (V4L2) interface. Using horsepower from the graphics card to do some of the image processing reduces the load on the CPU and may result in better throughput. The article describes the Glutcam program, which I developed, and the pieces behind it.

The Glutcam program takes data from a V4L2 device and feeds it into OpenGL for processing and display. To use it, you need the following pieces:

- A Video4Linux2 (V4L2) device.
- OpenGL 2.0 (or better).
- The GLUT or FreeGLUT library.
- The GLEW library.

First, let's look briefly at the individual pieces and then look at the complete Glutcam program.

The Video4Linux2 Device

Video4Linux2 is the interface between Linux and many video devices, including tuners and some Webcams. (Some devices still are using the older V4L1 drivers.) Bill Dirks started the Video4Linux2 API in 1998. It was merged into the kernel in 2002 in version 2.5.46. The V4L2 API controls the imaging device. V4L2 allows the application to open the device, query its capabilities, set capture parameters and negotiate the output format and method. Two of those things are especially important to us: the output mechanism and output format.

The output mechanism moves video data from the imaging device into your application. This is less trivial than it sounds, because the driver has the data in kernel space and your application is in user space. There are two methods for making this transfer. They are the “read” interface, which is the normal read() function reading from a device, and the “streaming” interface.

Because speed is important, the streaming interface is preferred, as it is significantly faster than the read interface. The read interface is slower, because it needs to copy all the data from kernel space into user space. The streaming interface, on the other hand, can eliminate this copying in one of two ways: the interface can give the user-space application access to buffers in the driver (called memory mapping), or the application can give the driver access to buffers allocated in user-space (called user pointer I/O). This saves the overhead of copying the image every time one is available. The drawback to the streaming interface is complexity: you have to manage a set of buffers instead of just reading from the same buffer each time via the read function.

However, the complexity is worth it. The single biggest speed-up I got developing Glutcam was switching from read to streaming.

The second item to watch is the output format. Not all devices can supply all output formats natively. Some use in-kernel software to translate between the formats the device provides and what the application asks for. When there's a difference between what the device can supply and what the application asks for, a translation has to be done. The kernel driver does that translation and takes some CPU time. The strategy is to move as much of that processing onto the GPU as possible. So, pick a format that minimizes in-kernel processing.

The main formats are greyscale, RGB and YUV. Greyscale and RGB are familiar to all. YUV (sometimes referred to as YCbCr) separates the brightness information (also called luminance and symbolized as Y) of each pixel from the color information (called chrominance and symbolized as U and V). YUV evolved when early black-and-white television started including color, but still had to be compatible with black-and-white receivers. There is a whole family of formats in the YUV category based on how color is stored. For instance, the human eye is less sensitive to color than to the brightness of a pixel. This means there can be a brightness for each pixel, and the color values can be shared between adjacent pixels—a natural form of compression. Frequently used terms include the following:

- YUV444 (there is a Y, U and V for each pixel).
- YUV422 (Y for each pixel, U and V shared between adjacent horizontal pixels).
- YUV420 (Y for each pixel, U and V shared between adjacent horizontal and vertical pixels).

There is a lot of variation in formats for how the information is stored in memory. One of these variations is whether the components are “planar” or “packed”. In planar formats, all the Ys are stored together, followed by blocks of Us or Vs. The packed formats store the components mixed together.

Glutcam accepts RGB, greyscale, planar YUV420 and packed YUV422. Some sources supply only JPEG; Glutcam does not speak JPEG.

OpenGL

V4L2 covers how we get the data in; to display the data, we use OpenGL. OpenGL is a real-time graphics library that SGI first released in 1992. Brian Paul released the Mesa implementation of the OpenGL API in 1995. Mesa allowed OpenGL to run completely in software. SGI gave the OpenGL sample implementation an open-source license in 2000. You may have OpenGL on your machine from Mesa, or you might be able to get it from your graphics card vendor. Vendors frequently release OpenGL libraries that take advantage of their graphics hardware to run faster than a software-only version. In 2006, control of the API passed to the Khronos Group, a nonprofit technology consortium. Today, you can find OpenGL in everything from supercomputers to cell phones.

OpenGL contains two processing paths: the fragment processing path and the pixel processing path. The fragment processing path is the best known one. You feed in information about lights, colors, geometry and textures, then OpenGL generates an image. The second path is the pixel processing path. The pixel processing path starts with pixels and textures and lets you operate on those to generate an image. An optional part of OpenGL, called the ARB Imaging Subset, builds some common image processing operations into OpenGL. Even without the imaging subset, plenty of operations are built in for scaling, warping, rotating, combining images and overlays. OpenGL also contains the compiler and runtime environment for shaders written in the the GLSL shader language. This brings us to the next piece, shaders.

A shader is a small program that runs (ideally completely) on the GPU. Shaders can be written in several languages. Microsoft has High Level Shader Language (HLSL). In 2002, NVIDIA released Cg (C for Graphics). OpenGL has GLSL (OpenGL Shading Language). A shader can answer one of two questions:

1. Where do I draw? (A vertex shader.)
2. What color is this part of the image? (A fragment shader.)

This article concentrates on GLSL fragment shaders.

Shader programs allow the programmer to modify the “fixed functionality pipeline” of OpenGL. Fragment shaders take inputs from OpenGL textures and shader variables (Figure 1). Shaders have a C-like syntax with the same basic types and user-defined structures.

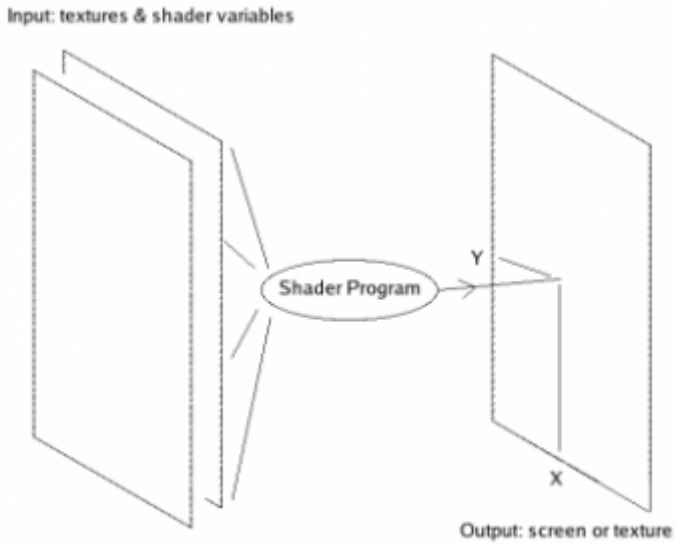

Figure 1. Fragment Shader Programs

A GLSL fragment shader looks something like this:

```
void main(void)
{
    // Making an assignment to gl_FragColor sets the color of
    // that part of the image. This function assigns the color to be
    // full red at full opacity.  The arguments to vec4 are
    // the components of the output color (red, green, blue, alpha).
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

GLSL starts with C, then adds support for vectors and a type to access textures. Textures are read using the “Sampler” types. Because Glutcam uses two-dimensional arrays, we will be using the Sampler2D type.

Putting functionality into OpenGL and shaders has many advantages, including:

- Graphics cards are (often) less expensive than CPUs and their capability is increasing faster than CPU capability.
-

Shaders can increase throughput while preserving the CPU for other tasks.

- You don't need to write code to interpolate values between pixels. The OpenGL texture mechanism will interpolate for you. This is important if your output image is not the same size as your input image.

Many image processing algorithms fit this pattern:

```
for y in height
    for x in width
        pixel(x, y) = something(input_image, x, y, ...);
    end
end
```

Which fits well with OpenGL and the shader mechanism (see below).

In the shader method of posing the problem, the input_image is a texture. OpenGL and the shader mechanism take care of the loops for you. Your fragment shader just does the "something(input_image, x, y, ...)" part. Note, however, that iterations of the loop can't communicate with each other. With that limitation, the iterations of the loop can be done in parallel: conceptually, there is an instance of your program running for each pixel in the output (Figure 2).

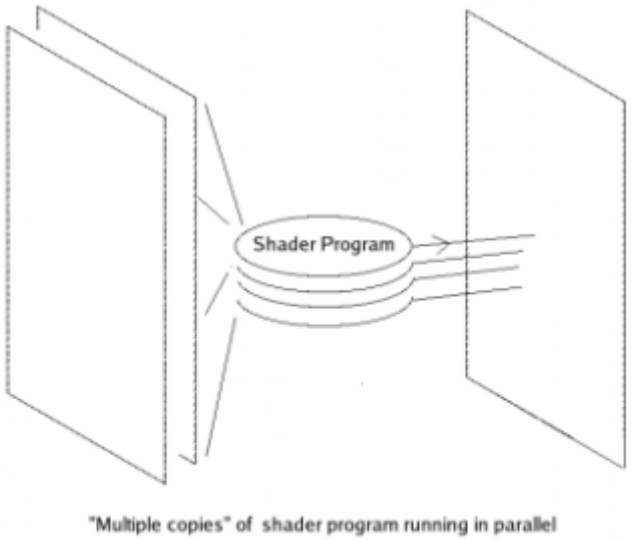

Figure 2. Multiple, Parallel Fragment Shaders

This has advantages but also limitations. Your GPU is a finite resource, and GLSL has limitations. These limits include:

- Arrays can be only one-dimensional: arrays of two or more dimensions have to be unwrapped.
- No bit-fields or bit-level operations. (OpenGL 3 relaxes that constraint.)
- No pointers.
- No extended precision (your GPU may support 64- or 128-bit pixels, but check before counting on it).
- No print statements. Your program's only output is the color of that pixel.
- Data must fit on the card. If you have very large textures, you might have to divide them up and process them serially or distribute them across more than one GPU.
- Your program must run within the time allowed. If your program runs too slow, your output frame rate will drop.
- Not all combinations of inputs are hardware-accelerated. Try to stay on the hardware-accelerated path.

If your program doesn't fit in these limits, it either will fail to compile, or it will compile and run partly on the CPU instead of completely on the GPU.

If your problem doesn't fit these parameters, try related methods like NVIDIA's CUDA or the newly arriving Open Computing Language (OpenCL). The General Purpose Graphics Processor Unit (GPGPU) page shows some of the tools and techniques available.

OpenGL Utility Toolkit Library (GLUT) or FreeGLUT

By itself, OpenGL is a display-only system. It doesn't handle window creation or placement, or processing keyboard and mouse events. The GLX layer of your X sever handles those (see the man page for glXIntro for more details).

GLUT encapsulates all those functions, and you organize your code into callbacks for things like drawing the window and processing keyboard and mouse events. The GLUT library handles all the lower-level functions. It also gives you an easy way to define menus. In addition to GLUT, there is the less-restricted implementation FreeGLUT, (I refer to them both as GLUT here.) Glutcam uses FreeGLUT to handle all those details.

GLUT is one of the simpler OpenGL GUI tools, but it certainly is not the only one. Take a look at the OpenGL Web site under coding resources for a selection of choices.

OpenGL Extension Wrangler Library (GLEW)

Because multiple groups support OpenGL, new features are not just added simultaneously in all implementations. Most new features start off as extensions proposed by members of the OpenGL Architecture Review Board. As these extensions are examined and tried out, some are accepted by more vendors. Some make it all the way into the core OpenGL specification. The OpenGL extensions mechanism lets your program check at runtime to see what features your OpenGL library supports and where those routines are located. The GLEW library handles all that. GLEW also supplies the glewinfo program that will tell you about your OpenGL implementation and what features it supports.

Glutcam

Now, we get to the Glutcam program. Glutcam connects a V4L2 device to an OpenGL window and uses a GLSL fragment shader to process the image (Figure 3). Ideally, we'd conserve CPU by using the GPU for all pixel-level operations. At a minimum, that's translating the camera-format pixels to RGB for display. Because I wanted to do more than just that color-space translation, Glutcam also does edge detection. Edge detection is a classic image processing operation. You can use the menu to select whether the edge detection is done on the CPU or the GPU. You can compare the results by looking at Glutcam's frame rate and use the `top` command to see the difference in CPU load.

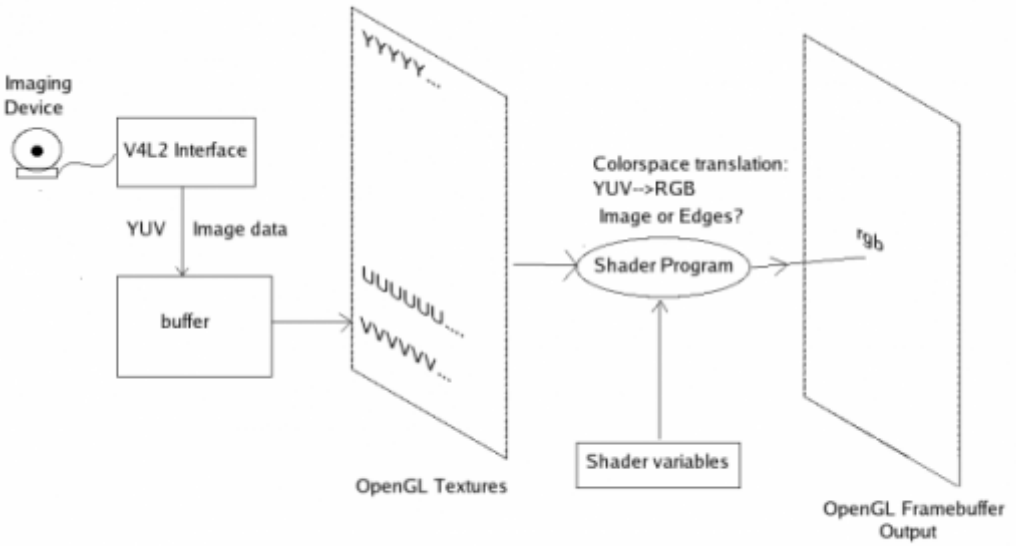

Figure 3. Data Path from Camera to Screen

In the best case, you'd tell the imaging device to produce an image in a format your shaders know how to handle, then put the image data into one or more OpenGL textures. When you draw, the data passes to your fragment shader. If you've chosen no edge detection or CPU-based edge detection, the shader converts it to RGB for display. If you've chosen GPU-based edge detection, the shader runs the edge detection algorithm on the image and displays the result.

Glutcam uses Laplacian edge detection. The CPU option implements this as a convolution with the kernel:

```
-1  -1  -1
    -1   8  -1
    -1  -1  -1
```

Which is to say, you compute the output value for each pixel by taking the value of corresponding input pixel, multiplying it by 8, then subtracting off the values of the neighboring pixels. In the places where the pixel has the same value as its neighbors, this makes the output pixel 0 (black). So edges (where the colors change abruptly) show up. Places where the colors change slowly are dimmed. Kernels and convolutions are work horses in image processing. You can use the same algorithm and just change the values in the kernel to get lots of effects, such as sharpening, blurring and smoothing.

In the shader, this takes the form of pulling a value from the input texture, multiplying it, then subtracting off the values of the adjacent texels (texture elements), and using the result as the output value. It looks more like algebra, but it has the same effect.

To compile Glutcam, make sure you have FreeGLUT/GLUT, GLEW, OpenGL 2.0, V4L2 (not V4L1) and all development packages installed. Untar the source, and see the directions in the Makefile. The sources are attached to this article. See the file at the bottom of the page.

Once built, invoke it with:

```
glutcam [-d devicefile][-w width][-h height] \
        [-e LUMA | YUV420 | YUV422 | RGB]
```

If invoked without options, Glutcam will display 320x230 greyscale test patterns.

To connect to a V4L2 device on /dev/video1 and to get a 640x480 image in YUV420 format, use:

```
glutcam -d /dev/video1 -w 640 -h 480 -e YUV420
```

On start up, the program tells you about its environment—that is, the V4L2 device it is talking to and what OpenGL it is using to display the data. You should see a startup plume something like what is shown in Listing 1.

Listing 1. Sample Output Messages from Glutcam

```
$ glutcam -d /dev/video0 -w 640 -h 480 -e YUV422
Info: '/dev/video0' connects to UVC Camera (046d:08c5)
      using the uvcvideo driver

Device has the following capabilities
```

```
Device: 'UVC Camera (046d:08c5)' Driver: 'uvcvideo'
Device supports video capture.
Device can NOT supply data by read
Device supports streaming I/O
Device does NOT support asynchronous I/O
Device has the following controls available using device file /dev/video0
 Control Brightness:integer 0 to 255 in increments of 1
 ...
Source supplies the following formats:
[0] MJPEG
[1] YUV 4:2:2 (YUYV)
device supports -e YUV422
Warning: VIDIOC_S_CROP cropping not supported: Invalid argument

Opengl information
Opengl vendor 'NVIDIA Corporation'
 renderer 'Quadro FX 1500/PCI/SSE2'
 version '2.1.2 NVIDIA 169.09'
Supported texture units: 4
...

Shader information:
8 active vars
0 'chroma_texcoord_offsets'
...

Press a key in the display window
```

For your machine, you'll have to figure out what device file your source uses (for example, -d /dev/videoN), what size images it likes to produce (-w, -h) and what format it generates (-e).

Press a key in the Glutcam window to start displaying what your Webcam sees. Figure 4 shows an example.

Figure 4. Screen View

If you have the imaging subset, you can turn the histogram on or off with the toggle histogram option (Figure 5). The histogram shows the portion of pixels at each brightness value when they're first passed in to OpenGL. The CPU-based edge detection will influence the histogram (since it changes the input to the shader) but shader-based edge detection will not.

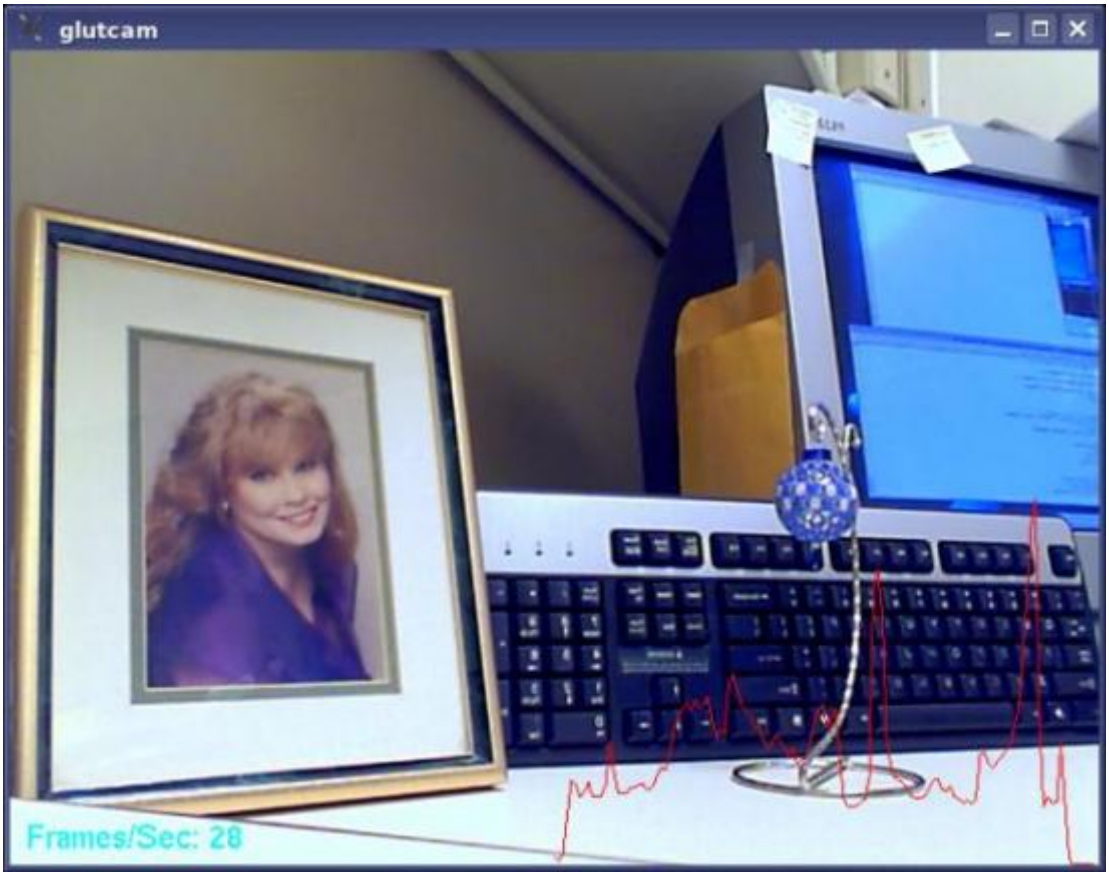

Figure 5. Screen View with Histogram

Right-click on the window to bring up a menu (Figure 6). You can choose one of the image processing algorithms to see the edge detection at work (Figure 7). Try both choices for edge detection (CPU and shader), and see the results on the frame rate and CPU usage in the `top` command.

When the CPU has to touch every pixel (when doing the histogram or doing edge detection on the CPU), frame rates drop significantly (by 50–70%), and CPU usage rises significantly (by 100–200%). But, don't generalize too much from these results. Both hardware and OpenGL implementations will vary. Recall that software-only implementations of OpenGL will be slower. Glutcam will run on Mesa 7.6-01; so you will see the output, but you probably won't see the frame rate.

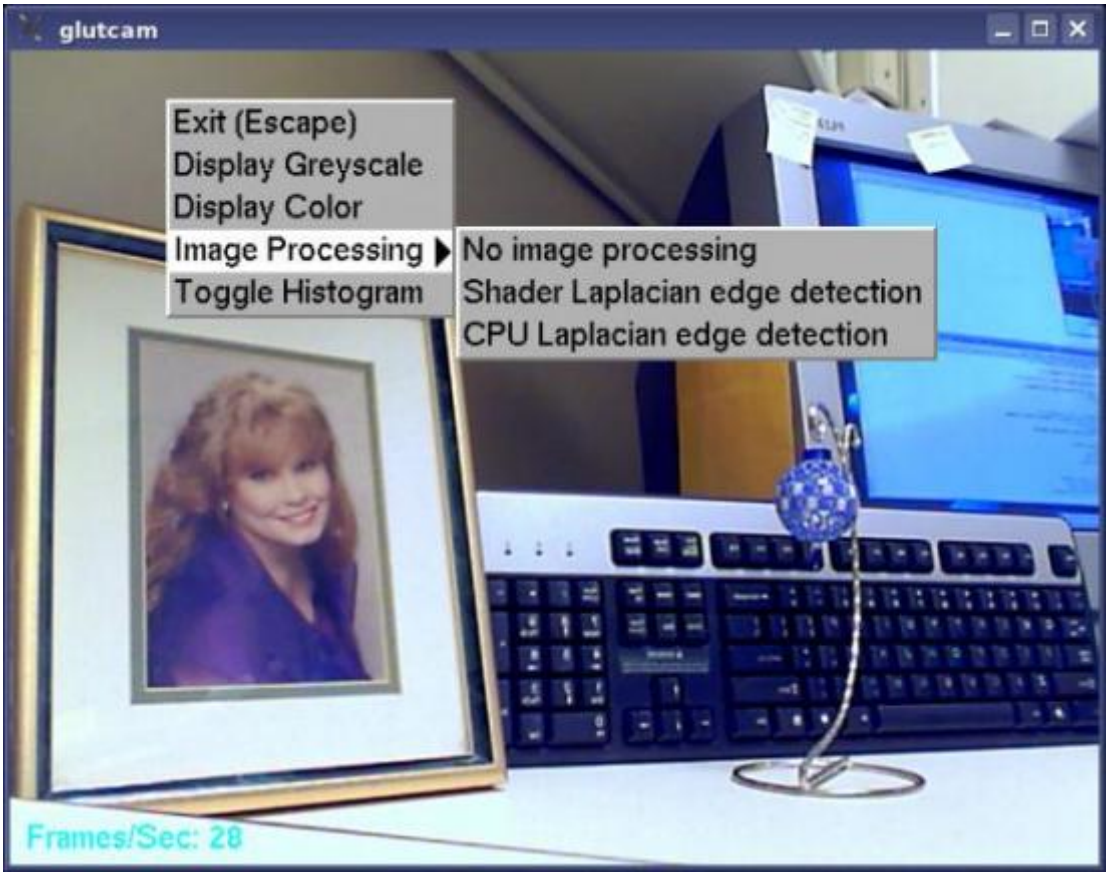

Figure 6. Glutcam Edge Detection Menu Options

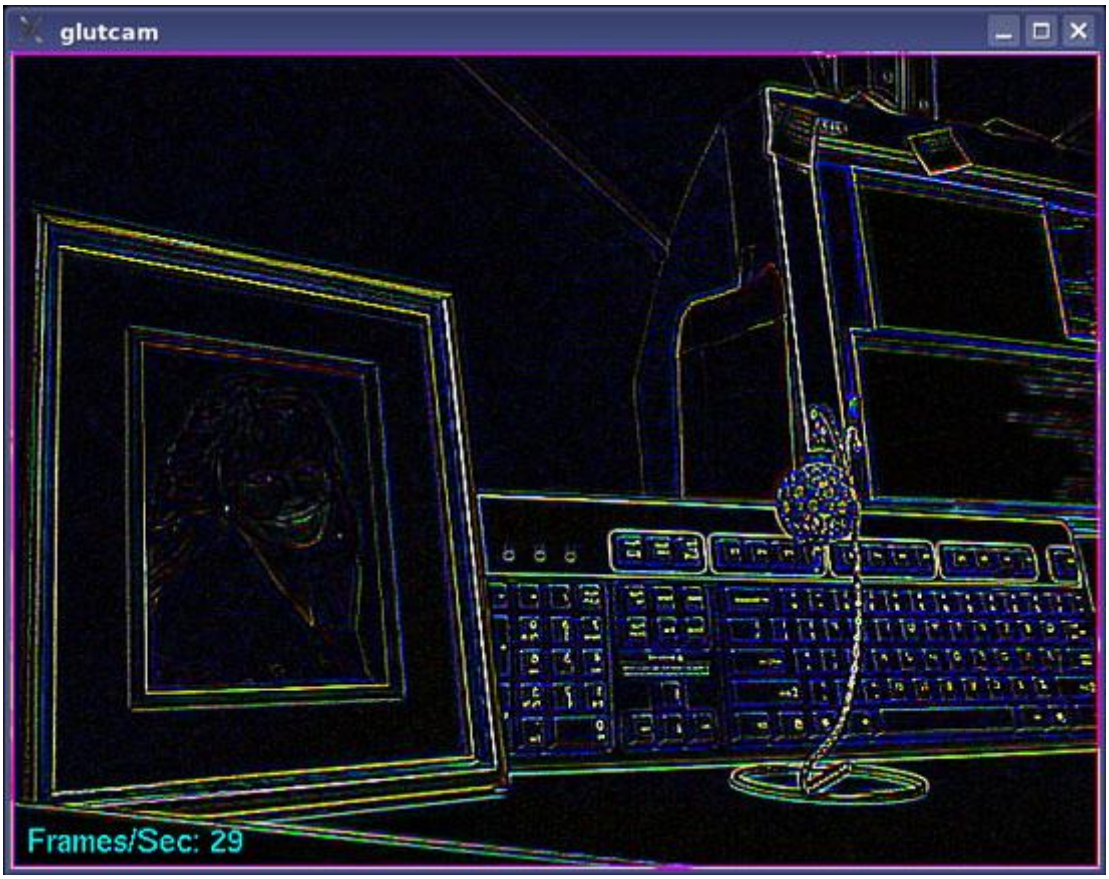

Figure 7. Glutcam Edge Detection

Finally, here are some shader development tips (and superstitions):

- Use the `top` command to keep an eye on your program's CPU usage.
- Start with a simple shader and add complexity gradually. Rethink your approach when your frame rate drops or your CPU usage spikes. This may mean you're running on the CPU instead of the GPU.
- Watch the compile time logs from OpenGL when it compiles the shader. Warning level and quality of error messages can vary a lot. Watch for warnings that your shader runs partially on the CPU.
- Watch the number of texture accesses your code makes. On the oldest machine in my tests, a 3x3 matrix for the edge detection was the largest I could use without the frame rate dropping. On the newer machines, operations with a 5x5 matrix worked well.
- Profile the CPU part of your application before getting too deep. You want the CPU to hand off the work to the GPU, and the GPU to have enough time to do the work. So if you speed up the CPU portion, it may give the GPU a bigger increment of time to work in. If you are aiming for 60Hz, you want 16 milliseconds of work on the GPU and 16 milliseconds of time to do it in.
- Use a test pattern stored in memory to see if your bottleneck is getting the image or processing the image once you have it.
- If you've synced the refresh rate of the monitor with the OpenGL buffer swap, see if reducing the monitor's refresh rate increases your frame rate. Use the `xrandr` command to show the resolutions and refresh rates available to you.
- Minimize the number of state changes you make to the OpenGL variables passed to the shader. These have to be passed to your GPU, just like the image data.
- Use the `xmag` command to see the numeric value of a pixel on the screen.

- Try different screen resolutions and OpenGL window sizes. Some may be accelerated while others are not.
- When using a histogram, one with fewer bins will slow down the process less than one with more bins.
- Be aware that your input source may have an associated rate. For instance, the pwc driver module has the “fps” parameter.
- Simplify the math in your shader.
- The GLSL compiler optimizes aggressively. This includes optimizing out parts of your shader that it determines unnecessary. Keep an eye on the warnings you get when you try to set the value of a shader variable. If you get an error, the variable may have been optimized out. If all variables give you errors, the entire shader may have failed to compile.
- Write your shader to make it easy for the GLSL compiler to optimize. For instance, hard-coding loop bounds instead of passing them in from OpenGL can speed up your shaders.
- Watch out for texture interpolation if you're using a packed format. If you have a planar format, interpolating between two adjacent values (say two lumas) gives you a legal value. Interpolating between a luma and an adjacent chroma will not give you the result you're looking for.
- Keep an eye on shader development and performance analysis tools. There are some interesting things in the pipe.

Conclusion

I saw a question posted in the Usenet newsgroups back in the 1990s (when 486s were state of the art). The poster wanted to get a coprocessor to speed up raytracing. The best suggestion anyone could come up with was to reprogram the digital signal processors on a soundcard. The thought of using a soundcard to get better graphics tickled me. We're now in a time that is starting to be the mirror of that situation. With programmable graphics cards and related items like CUDA and OpenCL, you can use graphics hardware to accelerate other computation.

Resources

Sources file: **10575.tar** Glutcam Source:

OpenGL Web Site: **www.opengl.org**

Lighthouse 3-D OpenGL Tutorials: **www.lighthouse3d.com/opengl**

General-Purpose Computation Using Graphics Hardware: **www.gpgpu.org**

Video4Linux2 Spec: **v4l2spec.bytesex.org**

Video Codecs and Pixel Formats: **www.fourcc.org**

FreeGLUT Home Page: **freeglut.sourceforge.net**

GLEW Home Page: **glew.sourceforge.net**

Linux USB Web Site: **www.linux-usb.org**

OpenGL Shading Language by Randi Rost (second edition, Addison Wesley, 2006)

OpenGL Programming Guide by the OpenGL Architecture Review Board (sixth edition, Addison Wesley, 2008)

OpenGL Reference Manual by the OpenGL Architecture Review Board (fourth edition, Addison Wesley, 2004)

OpenGL SuperBible by Wright, Lipchak, Haemel (fourth edition, Addison Wesley, 2007)

Open Computing Language: **www.khronos.org/opencl**

Luc Saillard's PWC Page: **www.saillard.org/linux/pwc**

The Video4Linux2 API: an Introduction: **lwn.net/Articles/203924**

George Koharchik happily acknowledges those who reviewed this article: Mike Menefee, Rick Still, Jim King and Candy Koharchik. George can be reached at **g-koharchik@raytheon.com**.

Attachment	Size
10575.tar	450 KB