My Robot Blog

Friday, 25 October 2013

Pi Eyes Stage 3

For the past few days I've been working on getting a pair of raspberry pi camera modules working and accessing their data in a c++ program ready for image processing. Last night I got to the first version of my camera api which can be found here. So far I can:

- Start the camera up
- Have a callback triggered once per frame that gets past the buffer + its size
- Shut the camera down

Very soon I'll get to work on converting the YUV data the camera submits into nice friendly RGB data, and get downsampling of the images going. Both will either need to be done using mmal, or through my own GPU code if they stand a chance of being usable for real time processing. Once they're going I'll be in a great position to get more complex stuff like feature analysis working.

However, while I've got a host of ideas of how to move forwards, the first thing to do is get the output from the camera rendering on screen so I can actually see it in action. As such my next goal is to get opengl going, and render the output from the camera callback to the screen. Initially it'll look like garbage (as it'll actually be yuv data), but it'll be garbage that changes when things move in front of the camera! Once it's working I'll be in a position to do things like downsampling, rgb conversion etc and actually see the results to verify they're functioning correctly.

Getting OpenGL going

I've not endeavered to get opengl working on the pi yet, but there's a couple of examples called hello_triangle and hello_triangle2. On looking at them, hello_triangle2 is both the simplest and most interesting as it uses shaders to do its rendering. I start by copying the graphics init code from hello_triangle2, and then add begin/end frame calls that basically clear the screen and swap the buffers respectively. This rather uninteresting photo is the result:

OK so it's not much, but crucially it shows opengl is operating correctly - I have a render loop that is clearing the screen to blue (and all the while I'm still reading the camera every frame in the background).

Shaders, Buffers and Boxes

I'm not gonna mess with fixed function pipelines and then have to go back and change it to shaders as soon as I want something funky - this is the 21st century after all! As a result I need to get shaders working which I've never done in opengl. From the example it basically seems to be a case of:

- Load/set source code for a shader
- Compile it, and get a shader id
- Create a 'program', and assign it a vertex shader and a fragment shader

So I come up with this code inside a little GfxShader class:

```
bool GfxShader::LoadVertexShader(const char* filename)
{
    //cheeky bit of code to read the whole file into memory
    assert(!Src);
    FILE* f = fopen(filename, "rb");
    assert(f);
    fseek(f,0,SEEK_END);
    int sz = ftell(f);
    fseek(f,0,SEEK_SET);
    Src = new GLchar[sz+1];
    fread(Src,1,sz,f);
    Src[sz] = 0; //null terminate it!
    fclose(f);

    //now create and compile the shader
    GlShaderType = GL_VERTEX_SHADER;
    Id = glCreateShader(GlShaderType);
    glShaderSource(Id, 1, (const GLchar**)&Src, 0);
    glCompileShader(Id);
    check();
    printf("Compiled vertex shader:\n%s\n",Src);

    return true;
}
```

Email me

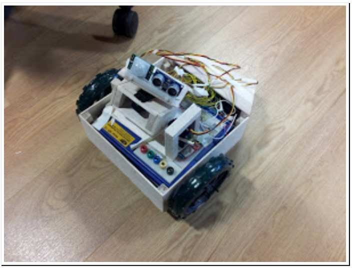
wibble82@hotmail.com

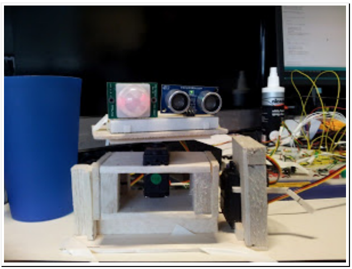
Follow me by Email

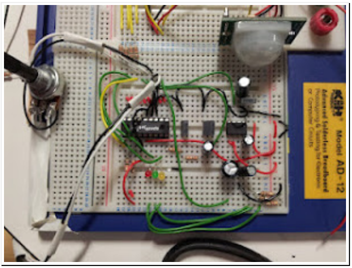
Email address... Submit

Links

- Happy Robot Games: No-Stick Shooter







About Me

Chris Cummings

I'm a games programmer and I like to program stuff and make robots

View my complete profile

Blog Archive

That just loads up a file, and fires it through the open gl code to create a shader program. Next, I knock together the simplest vertex shader and fragment shader I can think of:

SimpleVertShader.glsl:

```
attribute vec4 vertex;
void main(void)
{
    vec4 pos = vertex;
    gl_Position = pos;
};
```

SimpleFragShader.glsl:

```
void main(void)
{
    gl_FragColor = float4(1,1,1,1);
};
```

And now it's time to try and render a triangle using those shaders!!! Please note - at time of writing I still don't know if this is going to work, or if those shaders are entirely wrong... Unless I've missed something, it appears the old way of specifying vertices 1 by 1 isn't present in OpenGLES2 (although it's **very** possible I've missed something), so I'm gonna need to create me a vertex buffer. I knock together these bits to create and draw it...

Create it...

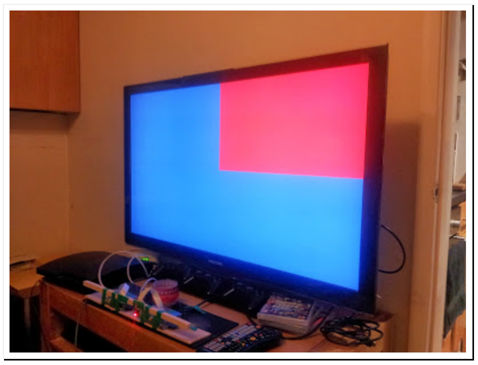
```
    //create an ickle vertex buffer
    static const GLfloat quad_vertex_positions[] = {
        0.0f, 0.0f,    1.0f, 1.0f,
        1.0f, 0.0f, 1.0f, 1.0f,
        1.0f, 1.0f, 1.0f, 1.0f,
        0.0f, 1.0f, 1.0f, 1.0f
    };
    glGenBuffers(1, &GQuadVertexBuffer);
    glBindBuffer(GL_ARRAY_BUFFER, GQuadVertexBuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(quad_vertex_positions), quad_vertex_positions, GL_STATIC_DRAW);
    check();
```

Draw it...

```
    glUseProgram(GSimpleProg.GetId());
    printf("gl error: %d\n",glGetError());
    check();
    glBindBuffer(GL_ARRAY_BUFFER, GQuadVertexBuffer);
    GLuint loc = glGetAttribLocation(GSimpleProg.GetId(),"vertex");
    glVertexAttribPointer(loc, 4, GL_FLOAT, 0, 16, 0);
    glEnableVertexAttribArray(loc);
    check();
    glDrawArrays ( GL_TRIANGLE_STRIP, 0, 4 );
    check();
    glFinish();
    glFlush();
    check();
```

But glUseProgram is giving me errors so its thinking hat time....

And... an hour of fiddling later I've discovered open gl doesn't return an error if the shader compiling or linking into a program fails. Instead it returns happy success, unless you specifically ask it how things went! Having fixed some compile errors in my earlier shaders I run it and am presented with my first quad:



And after adding offset and scale uniforms and passing them into this draw function...

```
void DrawWhiteRect(float x0, float y0, float x1, float y1)
{
    glUseProgram(GSimpleProg.GetId());
    check();

    glUniform2f(glGetUniformLocation(GSimpleProg.GetId(),"offset"),x0,y0);
    glUniform2f(glGetUniformLocation(GSimpleProg.GetId(),"scale"),x1-x0,y1-y0);
```

Pi Eyes Stage 1

Update on the pi bot simulator

► September (2)

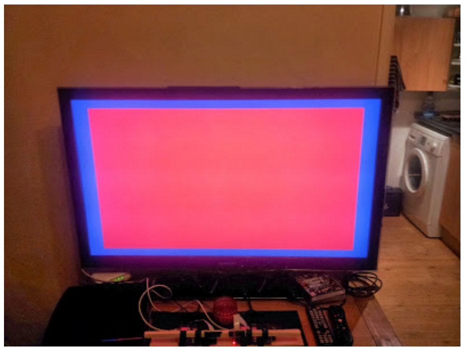
► February (1)

► January (3)

► 2012 (33)

```
    glBindBuffer(GL_ARRAY_BUFFER, GQuadVertexBuffer);
    check();

    GLuint loc = glGetAttribLocation(GSimpleProg.GetId(),"vertex");
    check();

    glVertexAttribPointer(loc, 4, GL_FLOAT, 0, 16, 0);
    check();

    glEnableVertexAttribArray(loc);
    check();

    glDrawArrays ( GL_TRIANGLE_STRIP, 0, 4 );
    check();

    glFinish();
    check();

    glFlush();
    check();
}
```

Hmm - not sure what the glFlush does yet. One for later though. The point is I can make a box anywhere I want:

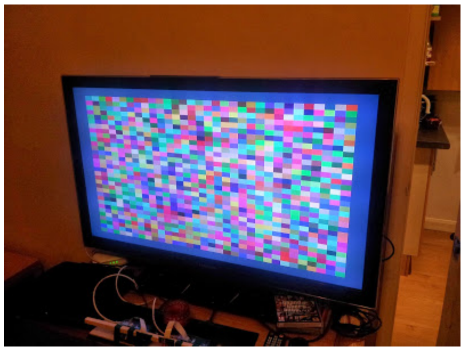


OK, it's.....

Texture Time

My ultimate goal here is to get the camera texture on screen, which will involve filling an open gl texture with data each frame and then displaying it on a quad like the one above. Before getting that far I'm just gonna try filling a texture with random data each frame and seeing where that gets me...

...half an hour later... well having grasped opengles2 a bit better, that was actually fairly easy. We have a 32x32 random texture (and a code base that's getting messier by the second):

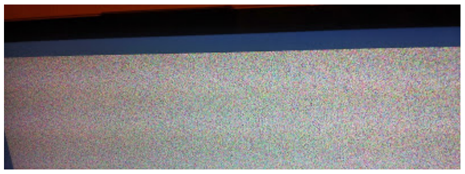


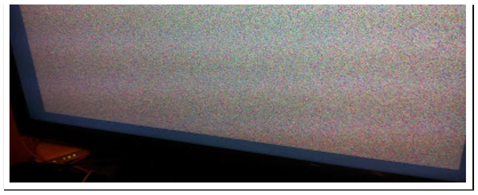
Woohooo!

From camera to texture

This is it folks. If I can get from camera output to something on screen at a decent frame rate then it paves the way for much wonders on this raspberry pi. I'll start with a hacky not-thread-safe approach which will also waste a bit of cpu time doing extra memcpys and generally be bad. But quick to write.

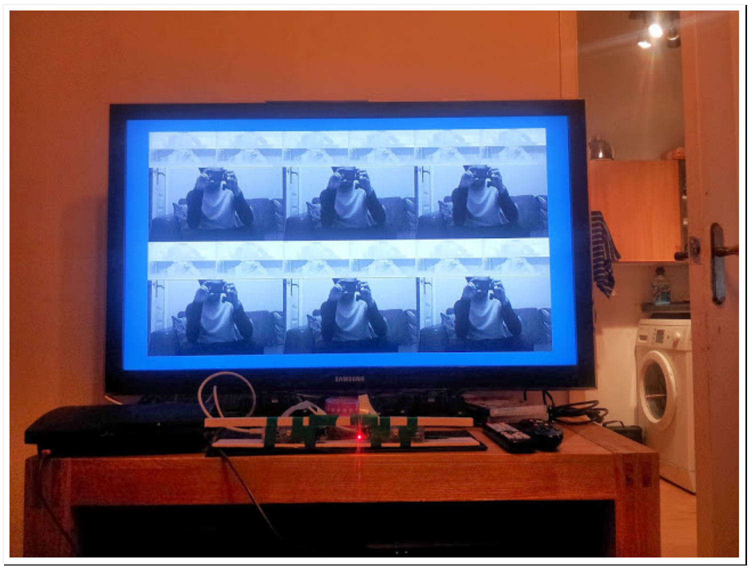
So we've got a callback in the app that is submitting new frames from a separate thread, and a call on the main frame to render a texture on screen. I just need to get the data from the thread into the texure and all will be well. I start by bodgily setting my earlier 'random texture' to be exactly the right size for a 1280x720 camera image, resulting in something a little more 'trippy':





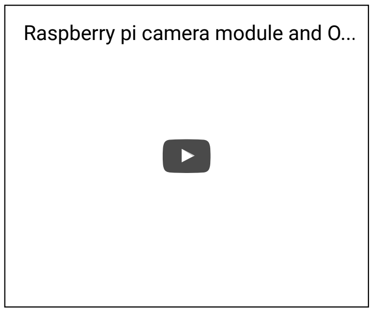
Now to try regenerating that random data each frame - works although very slow. Not even worth uploading the video I made of it really!

However, I now have code that generates some new data, pumps it to open gl and then draws it on screen. All I need to do now is use my camera feed as the 'new data' instead of random numbers. I write a new function that takes a pointer to a buffer and copies it into the block of memory I was previously filling with random numbers. Remember my camera data is still in YUV so it'll not fill a full RGB texture (and will look weird), so I make it keep copying until it fills the buffer - this gives me a good measure of performance. A bit of jiggery pokery and...



Eureka!!!

At 1080p the memcpys involved (probably 2 - one from camera -> scratch space, another from scratch space -> open gl texture) are heavy enough and it hits about 10fps. But at 720p (still easily enough for some tasty image fun) it's in at around 25fps. With a little clever engineering I can remove 1 of those copies, so it'll hit a solid 30fps. Here's a video to show it in action:



Pretty tasty yes? Although please note - when I say 'copies it into the cpu' I mean 'into cpu accesible memory'. One doesn't make sense, the other does...

All code is here, although it's in a bit of a state right now so don't take anything as 'the right way to do it' - especially the graphics bits!

http://www.cheerfulprogrammer.com/downloads/pi_eyes_stage3/picam_rendering.zip

Next Steps

Now that I can see what I'm generating (and can do it at an appreciable frame rate) I'm going to look at using the mmal image resizer component to create downsampled versions of the feed (for different granularity in image processing) and in theory do the rgb conversion (if the documentation is telling the truth...).

Firs though, I need to order a takeaway.

Posted by Chris Cummings at 20:06 G+1 Recommend this on Google

7 comments:

chad 14 November 2013 at 11:45

Chris,

really struggling to dupe this process!!!
no surprise there then.

what is excellent is that afaict,
you are not relying on an external build or ESlib
and the code is succinct.
which really helps matters.

for this page 'only' please could you provide code for each step?
or perhaps comment code to indicate how to achieve results?
add a simple triangle example?

very hard to understand later steps, without the basics...

Reply

Chris Cummings 14 November 2013 at 12:16

Hey Chad. Have you seen my latest 2 posts? The first supplies the finished camera API, which internally has a lot of comments to descirbe what it's doing. The last one is an adaption to using the GPU for image processing which is a little less commented but still fairly useable.

Reply

chad 14 November 2013 at 15:40

Chris,
yup looked through them, and want to get there,
but need to start at the beginning...

would really appreciate expansion of this page with code samples...
ie simple shader rect, random texture,
whatever is easy for you, and helps one understand your route to here.

Reply

chad 15 November 2013 at 11:09

for 'trippy'
picam.cpp comment out:
// WriteToRandomBuffer(buffer,buffer_length);
graphics.cpp uncomment:
for(int i = 0; i < sizeof(GRandomTextureBuffer); i++)
GRandomTextureBuffer[i] = (unsigned char)rand()%256;
~:"

Reply

Chris Cummings 15 November 2013 at 11:12

Hi Chad

I'm a little busy to revisit this section right now, as I'm working on the updated version of the api which will be more gpu focused. I will make sure that one contains some simpler samples and a better commented code base in general though.

Is it the use of egl/opengl that you're struggling with? If so I'd recommend searching for egl tutorials online - there's loads of really good ones. I actually came at this with a very minimal understanding of opengl and by following some online tutorials was able to put together the code above. Make sure you look for egl2 - that's the version that uses shaders, rather than old school egl1 which is a fixed function pipeline and much less powerful.

fyi - the c++ api contains graphics.cpp, which contains all my actual egl code. It's not the greatest code in the world but is a fairly short piece of work that could probably be understood very easily once you'd looked at a few egl2 tutorials.

Reply

chad 15 November 2013 at 12:01

for 'Texture time' in graphics.cpp change 1280, 720 to 32, 32
three times in total
~:"

Reply

chad 15 November 2013 at 12:08

excellent, please continue providing comments and examples along the way, really helps. Thank You!

excellent, please continue providing comments and examples along the way, really helps, Thank You!

will check egl2 tutorials presently...

Reply

Enter your comment...

Comment as: Unknown (Goo ▾ Sign out

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)

Simple template. Powered by Blogger.