- „Hello World“ in OpenGL -

Goal:
In this exercise we set up the OpenGL working environment, we learn how to control X11 windows on both host and target using Eclipse, and we work on our first simple OpenGL program

Exercise 15.1: Set up your Work Environment

OpenGL programs require various libraries, which we first have to install both on host and on the target. We assume, that on the target we have already X11 installed (this is true, as long as we use our original target image).

Using **`sudo apt-get install`** we first install the following packets:

```
freeglut3 freeglut3-dev libglew1.10 libglew-dev
libglu1-mesa libglu1-mesa-dev libgl1-mesa-glx
libgl1-mesa-dev libfreetype6-dev
```

Exercise 15.2: Compilation for and Execution on the Host

Now you create a new C-project called **`ogl`** as a Makefile project in Eclipse. Create the usual folders **`src, Debug_with_Cross\ GCC und Debug_with_Linux\ GCC`** in your project directory (Note: the spaces in the directory names have to be masked with “\”, if you use the console). Copy the files from the quick start folder of exercise 15 in your **`src`** folder. Doublecheck the Makefile (hint: compare to the Makefile we have used in exercise 12). Create your own make target called **`rect02_host`** to build the executable **`rect02`** for the host. Construct an appropriate run-configuration in Eclipse and execute the program locally on the host.

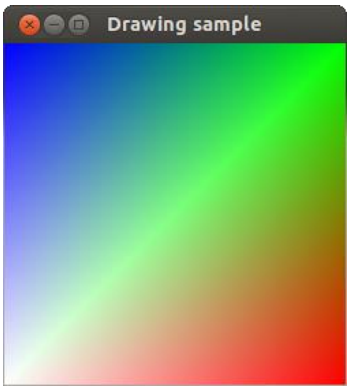

Fig.: Rectangle with Color Gradient in OpenGL / GLUT

Exercise 15.3:And now with Cross-Compilation and Execution on the Target

First we need to understand the display management on the X window system. For this, we install **`xterm`** and some smaller X11 applications on the target (and, if necessary, on the host), which can be found in the **`x11-apps`** package.

In putty we open a session on our target with X11-forwarding activated (Category -> Connection -> SSH -> X11 -> Enable X11 forwarding). In the putty terminal we start on the target an X11 application, e.g. **`xclock`** or **`xterm`**. We notice, that the window of our application does not open on the target as expected, but locally on the host, i.e. the window is provided by the local X11 server running on the host.

But how can we start an X11-application on the target with the window opening on the target? For this, we need a feature called display forwarding in X11, which also includes keyboard, mouse and stdin! This can be accomplished using the command line parameter **`–display`**, e.g. **`xterm –display :0 .`**

To enable cross compiling and cross linking the target executable, the linker needs access to the OpenGL libraries and headers on the target. So we need, as already done previously, to **export** the root directory of the target, which get **mounted** on the host in **`/mnt/rootfs`**. Again, this requires the following steps on the target:

a) Check / install the packages **`nfs-common`** and **`nfs-kernel-server`**
b) Now in file /etc/exports add a line, which will export the root directory to your host (adapt to the correct IP address of your host. **Note**: no space in front of “(“ !)

```
/       <myhost>(rw,sync,no_subtree_check)
```

Also
c) Check, whether this directory gets exported correctly using:

```
showmount –e localhost
```

d) if not, then restart **`rpcbind`** and afterwards **`nfs-kernel-server`**.

Now we mount the exported root directory of our target into the file system hierarchy of our host under **`/mnt/rootfs`**. First we create the directory using **`mkdir`** . Then we mount using:

```
sudo mount <myraspi>:/ /mnt/rootfs
```

(adapt the IP address or host name to your target). Now the host is able to access the content of the root directory of the target in **`/mnt/rootfs.`**
Hint regarding NFS-Client running on Ubuntu: When using **`mount.nfs`** pls. make sure to add the Option **`–overs=3`** . Without it, NFSv4 is used by default, which is based on the Kerberos authentication system. This might result in extremely long timeout intervals (>4min) when connecting.

After these preparations we are now ready to create another make target in Eclipse named **`rect02_target`**, which is used to build the ARM-based executable for the target. To allow the linker to find the target libraries, we again use the **`sysroot`** option. Where do we have to insert the parameter **`-sysroot=/mnt/rootfs`** into our Makefile?

After creating the executable for ARM (and checking with **`file`**), we need an appropriate remote run configuration in Eclipse. To make sure that the X11

application window gets opened on the target rather than locally on the host, we insert the argument

```
-display :0
```

into the parameter tab of our remote run configuration.

Now execute the program both on host and target. Compare the output on the screen. What do you notice? How can we improve the behavior of our program when resizing the window? Hint: reshape callback!

Add. question (bonus): what needs to be done to have the X11 window of the application appear on the host(!), when remote-executed from within Eclipse? What is the correct display number (hint: use **`xauth list`** to figure it out)? Where do we have to specify it in the remote-run config?