Part I
Computer graphics

The computer graphics part is written in C and compiles using make. Most of the code is contained in the file `assignment.c`, with the exception of the math library given for excercise 1.

The code draws a square and a pyramid. The pyramid rotates around an invisible origin, as well as rotating around its centerpoint. The square rotates around the pyramid, always having the same face towards the pyramid. Pressing the space button will change the center of the camera, from either being centered on the square, the pyramid or the invisible origin.

1 Rendering using scenegraph traversal

The scenegraph is constructed in a very similar fashion to the scene graph given in excercise 2. Each node in the graph is a struct. This node contains a (single) transformation matrix (`transform`), a function pointer to a function that *draws* that node (`draw`), an id (`object_no`), an int containing the number of children (`children_count`), an array with pointers to the child nodes (`children`) and finally a pointer to a parent (`parent`).

The id is not used and is a remnant of the scene graph given in the excercise. Its use is replaced by a pointer to a function. I felt this made the `traverseGraph()`-function easier to read and easier to extend.

Nodes are created with the `make_node()` function which takes the address of a function as a parameter. Children are added to nodes with the `add_child()` function and removed with the `remove_child()` function. The two functions `destroy_node()` and `destory_node_rec()` both delete a node (from memory), where the latter also deletes a nodes children.

Objects are then rendered by traversing this scene graph such that children of a node position themselves *relative* to their parent. The OpenGL API provides a stack that saves transformations as the tree is traversed. As the traversal goes down the tree a transformation matrix is *pushed* on the stack and as the traversal goes up the tree a transformation matrix is *popped* off the stack.

This traversal of the scene graph is done by calling the function `traverseGraph()`. The function calls itself recursively and takes a `node` pointer as a parameter, and for each invocation of the function, the transformation matrix found in the node is pushed onto the stack, the node is drawn, the function is recursively called on the nodes children and finally the transformation matrix is popped of the stack.

2 Extracting transformations from scene graphs

2.1 The frame-to-canonical matrix

Extracting transformations from a scene graph is implemented in the function `frame_to_canonical()`. It takes a pointer to a node as a parameter and traverses 'up' the scene graph until it reaches the root. At each node along the way it computes the inverse of that nodes transformation matrix and multiplies it on an intermediate matrix. This intermediate matrix will contain the complete inverse transformation matrix.

Say that a node is positioned such that its complete transformation matrix looks like

$$\mathbf{M} = \mathbf{M_0 M_1 M_2}$$

To get the position of the camera in world coordinates the inverse transformation is needed, that is:

$$\mathbf{M^{-1}} = \mathbf{M_2^{-1} M_1^{-1} M_0^{-1}}$$

And this is what the function `frame_to_canonical` computes.

2.2 The camera

The camera is added as a regular node, in the scene graph, but without any pointer to a draw function (i.e. the pointer is 0). The inverse matrix, i.e. the position in world space, is then acquired using the `frame_to_canonical()` function explained above. The position of the 'real' OpenGL camera is then set to this matrix in the `display()` function.

3 Per-pixel lighting

3.1 Positioning of light sources

I was unable to selectively position any light sources using OpenGL, but it should be fairly straightforward in theory. You can create one or more nodes representing one or more lightsources in the same manner as the camera from the precious section and use the function `frame_to_canonical()` to get the position of a light source in world space. This positions of the light sources would then need to be passed on to the shaders and retrieved in a similar fashion as it is done below. How to 'pass on' the positions eluded me though.

3.2 Per-fragment lighting

Per-fragment lighting is achieved using vertex and fragment shaders in OpenGL. Shaders are implemented using the skeleton from excercise 2, where the vertex and fragment shader-files have been modified appropriately.

The shading model used to implement per-fragment lighting is Phong shading using equations 10.5 and 10.6

$$c = c_l max(0, \mathbf{e} \cdot \mathbf{r})^p$$

$$\mathbf{r} = -\mathbf{l} + \mathbf{e}(\mathbf{l} \cdot \mathbf{n})\mathbf{n}$$

where c is the final color of the pixel, c_l is the specular color of the surface, $\mathbf{e}$ is the unit vector towards the eye, $\mathbf{n}$ is the normal unit vector of the surface and $\mathbf{l}$ is the unit vector towards the light source.

In the vertex shader the surface normal `n` and the vertex position `v` are transformed into eye coordinates and passed on to the fragment shader. In addition the normal vector is normalized.

In the fragment shader the direction of the light, as seen from the surface, is calculated as $\mathbf{l} = \mathbf{l_e} - \mathbf{v_e}$, where $\mathbf{l_e}$ is the position of the light and $\mathbf{v_e}$ is the position of the vertex, both in eye coordinates. The direction of the eye is just the direction of the vertex reversed, i.e. $\mathbf{e} = -\mathbf{v_e}$. These vectors, as well as $\mathbf{r}$, need to be normalized before using them. With these two vectors $\mathbf{r}$ is acquired using equation 10.6 and the specular shading is calculated using equation 10.5. The specular color (c_l in equation 10.5) is given by the variable `gl_FrontMaterial.specular` and the power p by `gl_FrontMaterial.shininess`.

Prior to adding the Phong shading there has also been added some ambient shading and some Lambertian shading (given by $max(0, \mathbf{l} \cdot \mathbf{n})$).

Part II
Image processing