TNCG14 – Advanced Computer Graphics Programming
1. OpenGL

January 12, 2009

1 Introduction

This is a quick introduction to OpenGL with corresponding lab assignments. The example code can be downloaded from the lab section of the course homepage:
`http://staffwww.itn.liu.se/~jimjo/courses/TNCG14-2009`.

1.1 Reading this compendium

The aim of this compendium is to introduce the reader to OpenGL. Both exercises and tasks should be carried out. We presume that you have good knowledge of object oriented programming, data structures, 3D-graphics and linear algebra from earlier attended courses. A good idea is to have a 3D-graphics book nearby when reading this compendium. We will refer to books or websites specifically when needed but in general the following book is useful:

- *OpenGL Programming Guide, Third Edition*, Woo, Nieder, Davis et. al., Addison Wesley, 1999

This book is considered a framework within its field.

We also recommend the following websites on the topic:

www.opengl.org Here can you find information on OpenGL as well as links to related sites.

www.sgi.com/software/opengl SGI's site about OpenGL contains links and information about several packages like OpenGL Optimizer, Volumizer etc.

devcentral.iftech.com A really good site with tutorials in C/C++ and programming languages.

nehe.gamedev.net Here can you find excercises in OpenGL for most operating systems.

1.2 OpenGL

Modern 3D graphics requires hardware supported programming. OpenGL is the software interface to this type of graphics hardware. An important feature of OpenGL is that it is a state machine, signifying that some calls to the API changes states in the OpenGL context that will affect subsequent calls to the API, for example for rendering to the context. These are materials, colours, model transforms, view and specularity, just to mention a few.

OpenGL is also window-system independent, it does not have any functions to create and handle windows, but only handles rendering to a rectangular OpenGL context area on the screen.

2 GLUT

In order to quickly get started with OpenGL and be able to easily handle windows in different operating systems, eg. Linux, one can use a library called "OpenGL Utility Toolkit" (GLUT). This is a platform independent library that contains functions for creating windows, handling in-data from mouse and keyboard, creating simple menus and drawing some simple geometric objects. GLUT helps to avoid the often troublesome and not standardized procedure of connecting OpenGL with the respective windows handling system. There are several functions one must learn to use in order to create a window using GLUT. The most fundamental functions that should be used are:

```
glutCreateWindow(char *string)
glutDisplayFunc(void (*func)(void))
glutMainLoop(void)
```

These can make a window frame appear on the screen, nothing more. A complete but still very simple program includes the following functions:

```
glutInit(int *argc, char **argv)
glutInitDisplayMode(unsigned int mode)
glutInitWindowPosition(int x, int y)
glutInitWindowSize(int width, int height)
glutReshapeFunc(void (*func) (int w, int h) )
```

Some of these functions are self explanatory, for example `glutInitWindowPosition()` specifies the position of the window's top left corner and `glutInitWindowSize()` specifies the size of the window. The coordinates specified in `glutInitWindowPosition()` are calculated in pixels from the screen's top left corner (positive x-coordinates to the right and y downward).

The function `glutInit(int *argc, char **argv)` initiates the GLUT-library, creates a connection between OpenGL and the window system and handles some arguments of `main`. The first lines in a program look like:

```
int main(int argc, char **argv) {
  glutInit(&argc, argv);
  ...
}
```

Observe that the first parameter should be a pointer.

The function `glutInitDisplayMode(unsigned int mode)` is the first that needs a more detailed description; this function specifies which attributes we want the window to have. The argument to the function is a bitmask, i.e. every bit of the argument decides if a certain attribute should be on or off. GLUT has several predefined bitmasks and we just have to use two of them to begin with, GLUT_RGBA and GLUT_SINGLE. GLUT_RGBA specifies which colour model we will use and GLUT_SINGLE specifies that we will use single buffering.[1] This looks like:

```
glutInitDisplayMode(GLUT_RGBA|GLUT_SINGLE);
```

Now it's time to create the actual window with the attributes specified above. We do this with `glutCreateWindow(char *string)`. This is very straightforward. You simply call the method and send along a string that will be shown in the title row of the window. Example:

```
glutCreateWindow("Brocolli");
```

A function must be registered that is responsible for actually drawing what we want to draw in the window. This is done by:

```
glutDisplayFunc(void (*func)(void))
```

[1] Single- and double-buffering will be explained in a later chapter.

Don't be scared if the argument list seems complicated! The declaration `void (*func)(void)` means that `glutDisplayFunc()` takes a pointer to a function as its argument. This function does not take any arguments and does not return anything. A suitable name for such a function could be `display()` and must be declared as:

```
void display(void) {
  ...
}
```

When you have written the display function you have to register it as the display callback function by writing:

```
glutDisplayFunc(display);
```

`display()` will now be called everytime the program decides that a window has to be redrawn. Finally, in order for the window to be shown on the display, call `glutMainLoop()`. This is the function in GLUT that is responsible for sending events to the right callback function. The function doesn't return.[2] A simple program that only creates a window could look like this:

If no code is written in the display function, as in the example, it will not be all that fun to look at the window. Only a window frame with the correct dimension will be drawn.

It's no harder than this!

3 Double buffering and anti-aliasing

To do a simple animation is easy. Explanations of the functions that are used in the example below will come in a later chapter. For now you will get a simple program that you can change and play with a bit. The following code is found in the file `wcube.c`. Test run and study the code. Compile with `make wcube`. Notice that the image is flickering, this is because we actually look at the image while it is being drawn. One way to solve this it to use two buffers, one to draw in and one that is being shown. When the new image is ready, we change between the shown (Front buffer) and the one we draw in (Back buffer), which is exactly what the function `glutSwapBuffers()` does. This is called double buffering.

Activate double buffering in the above example by exchanging `GL_SINGLE` for `GL_DOUBLE` in the call of `glutInitDisplayMode()`. Run the program again. Change the program so that you get a smooth animation, i.e. change the number of degrees the cube is rotated for each image in the function `anim`.

As you notice the lines become 'jagged' in the above example. To create smooth lines anti-aliasing has to be activated. To do that add the following to the function `init`.

```
glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
glEnable(GL_LINE_SMOOTH);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_BLEND);
```

The first line asks OpenGL to draw smooth lines as well as possible, one can also choose as quickly as possible. The second line activates smooth lines. In order for anti-aliasing to work, pixels that are not on the line should be mixed with the background. This is what "blending" accomplishes. This can also be done for polygons, but it takes a little more work so we won't do it in this lab.

We can easily create a sort of illumination effect by redrawing the wire-cube a few times with different line thicknesses and transparencies. Change the function `display` by replacing `glutWireCube(1)` with the following code.

[2] A program can be terminated by calling `exit()` somewhere in the code, for example if the key Escape is pressed. You can also terminate the program by `<Ctrl>+c` in the window from which it was started.

```
glColor4f(0.8, 0.3, 0.1, 0.3);
glLineWidth(12);
glutWireCube(1);

glColor4f(0.8, 0.8, 0.1, 0.4);
glLineWidth(6);
glutWireCube(1);

glColor4f(1, 1, 1, 1);
glLineWidth(0.6);
glutWireCube(1);
```

4 Primitives in GLUT

OpenGL in itself doesn't have any object like primitives such as sphere, cubes, etc. GLUT, however, provides a few pre-made primitives, or simple objects:

```
glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);
glutWireSphere (GLdouble radius, GLint slices, GLint stacks);
glutSolidCube(GLdouble size);
glutWireCube(GLdouble size);
glutSolidCone(GLdouble base, GLdouble height, GLint slices,
              GLint stacks);
glutWireCone(GLdouble base, GLdouble height, GLint slices,
             GLint stacks);
glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius,
               GLint nsides, GLint rings);
glutWireTorus(GLdouble innerRadius, GLdouble outerRadius,
              GLint nsides, GLint rings);
glutSolidDodecahedron(void);
glutWireDodecahedron(void);
glutSolidOctahedron(void);
glutWireOctahedron(void);
glutSolidTetrahedron(void);
glutWireTetrahedron(void);
glutSolidIcosahedron(void);
glutWireIcosahedron(void);
glutSolidTeapot(GLdouble size);
glutWireTeapot(GLdouble size);
```

Replace `glutWireCube()` in `wcube.c` with a few of the functions above. Experiment with different settings for number of slices, stacks, sides and rings (`slices`, `stacks`, `nsides` and `rings`. How will a sphere with three slices and two stacks look? Draw multiple objects at the same time and draw them in different colours. You can draw them on top of each other.

If you choose the solid objects you will see the silhouettes of the objects. That's because we haven't defined any light sources and object materials yet, we will return to that later.

5 Transforms and function names

In the example above there was a cube rotating around its own origin. Two functions were used to transform the cube. `glTranslatef()` and `glRotatef()`. The camera was not transformed and will then reside in the world's origin. In order to get the entire cube visible we moved it in the world using `glTranslatef()`.

Table 1: Function suffix, C type och OpenGL type

Suffix	Data Type in C	OpenGL type
b	signed char	GLbyte
s	short	GLshort
i	int	GLint, GLsizei
f	float	GLfloat, GLclampf
d	double	GLdouble, GLclampd
ub	unsigned char	GLubyte, GLboolean
us	unsigned short	GLushort
ui	unsigned int	GLuint, GLenum, GLbitfield
	void	GLvoid

Since C doesn't support function overloading, OpenGL has no overloaded functions and the argument types are part of the function name. (Which in turn is exactly what C++ does automatically, but these names are hidden from the programmer and C++ uses a bit more cryptic names.) The OpenGL library makes these function name suffices explicitely visible and we therefore find both a `glTranslatef()` and a `glTranslated()`. They only differ by suffix to indicate double precision floating point `d` and single precision floating point `f`. In table 1 a complete list of types and their suffices are shown.

In addition, many functions take different number of arguments, like `glVertex*` which may take both 2, 3, and 4 arguments. Furthermore, some functions take an array (vector) as input which is indicated with a `v`. The argument to that function is then a pointer to a variable of the specific type.

Usually, OpenGL man pages or reference manuals do not list all possible combinations of these functions. A more compact form is used which indicates the possible combinations one can use:

```
glFunction{234}{sifd}{v}(TYPE x, TYPE y);
```

Function:	e.g. Translate, Rotate, etc
234:	One digit indicating the number of arguments the function can take, i.e. 2, 3 or 4.)
sifd:	Choose one of the letters for corresponding type.
{v}:	The function is available in a version that takes a pointer to the specified type, this pointer shall then point to an array containing the number of elements as determined by the digit.
(TYPE x,...)	The parameter `x` must be of type TYPE, as defined by the type suffix. For instance GLshort, GLint, GLfloat, GLdouble, ...

You may also encounter `glFunction*()` which means it is of no importance which function you use, just use one of them.

As an example we will look at functions to transform objects:

```
void glRotate{fd} (TYPE angle, TYPE x, TYPE y, TYPE z);
void glTranslate{fd} (TYPE x, TYPE y, TYPE z);
void glScale{fd} (TYPE x, TYPE y, TYPE z);
```

According the definition above you may use `GLdouble` or `GLfloat`. If you wish to use `GLfloat` you must call the function

```
glTranslatef(xTrans, yTrans, zTrans);
```

where `xTrans`, `yTrans` and `zTrans` is of the type `GLfloat`. It is also possible to use standard C types, but different systems might have different adaption of these.

The function `glTranslate*()` takes three arguments that specify a translation along the axis. `glScale*()` also takes three arguments but they specify scaling along the three dimensions. `glRotate*(TYPE angle, TYPE x, TYPE y, TYPE z)` takes four arguments, the first argument specifies the angle of rotation in degrees and the last three arguments is a vector around which to rotate. Rotating 45 degrees around the Y-axis is thus written as

```
glRotated(45, 0, 1, 0);
```

You must pay attention to the order of application of the transforms in order to get understandable results. A translation and then a rotation is not the same thing as a rotation and then a translation. Perhaps a brief reminder is in place.

You can view transforms in (at least) two ways. The first conception is that all transformations are applied to a an object in relation to a fixed world coordinate origin. The other conception is that the transforms are done in a local movable coordinate system that is transformed in world coordinates. There is no practical difference, just a matter of viewpoint. When you write the code the transformations will be added in the same order regardless of your viewpoint.

5.1 Fixed origin

If you wish to use the Fixed Origin model you must consider that the model always is transformed according to the origin. The object pivot point will be at the origin. A translation along the x-axis and then a rotation will have the object to rotate as if it was attached to a stick and this stick is rotated the origin. If you use this model you must apply the transforms in the inverse order in your code. That is how the transform matrices are computed.

5.2 Transforming a local coordinate system

Using the other model, it might be easier to understand what is happening. The conception is that each transformation transforms a local coordinate system within the world. All consecutive transforms are done in relation to the current local system.

If you first rotate and then translate you will perform the translation in the local system's x-axis, which has been rotated. If you first scale with 0.5 and the translate 2 units, the real world translation will be 1 unit.

The advantage of this method is that both conceptual order of transforms and the order in your code is the same.

5.3 Model View and Projection matrices

OpenGL is a state machine, besides calling the proper transforms we must make sure the transforms are carried out in the right state. OpenGL has three different transform matrices commonly used, the Projection, Model View, and Texture matrices (there are more which we will ignore for now). They are specified using `GL_PROJECTION`, `GL_MODELVIEW` and `GL_TEXTURE`. The projection matrix is used to define the camera. The model view matrix defines where the objects have their origin. The texture matrix is used for transforming texture coordinates. You select the corresponding matrix for manipulation by calling `glMatrixMode(mode)`, as in:

```
glMatrixMode(GL_MODELVIEW);
```

The program `ndcube.c` is an extended version of `wcube.c` above. We have added a light source and defined materials for a better view of what is going on. Compile the program by typing `make ndcube` and run it. Something is not quite right, can you tell what? While you ponder, draw a smaller cube in front of or behind and run again. Use the functions `glTranslate*()`, `glRotate*()` and `glScale*()` in different combinations. We will explain lighting and shading later on.

Add another object at some location and then another one behind the the first. Notice that the latter will be drawn on top of the other even though it is more far away in the scene. The reason for this is that OpenGL doesn't know at which depth a previously drawn pixel is and newer pixels will always be drawn over existing ones. We must request OpenGL to use a Depth buffer to properly depth sort fragments and only draw a pixel if it is in front of a previously drawn pixel.

6 Depth buffer

The depth buffer must explicitly be requested. Again, this is part of the windowing system and we use GLUT to set it up for us. Add `GLUT_DEPTH` in `glutInitDisplayMode()`.

```
glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
```

In addition to a colour buffer, we now also a have a depth buffer and OpenGL can discard pixels behind already drawn pixels in the frame buffer. However, we must instruct OpenGL to activate this test:

```
glEnable(GL_DEPTH_TEST);
```

In addition, and in line with clearing the colour buffer, we must reset the depth buffer for each frame we wish to draw. We therefore add `| GL_DEPTH_BUFFER_BIT` in the call to `glClear`. Add the call to enable depth testing in your `init` function.

Modify the program `ndcube` such that it uses a depth buffer properly. Try the program again and let the objects rotate around each other to better see the effect.

7 Matrix stack

Each matrix unit has its own stack in which a number of matrices can be stored. The model view matrix has a stack depth of 32 and projection matrix only have a few, maybe just 2, since they are seldom altered.

To save the current active matrix you call `glPushMatrix()`, the current matrix is not changed. You can now modify the matrix and draw some new objects. To restore the previous transform, the one that was active when you pushed it, you call `glPopMatrix()`.

The stack is useful when you want to draw a group of objects, which uses transforms, and then continue to draw more objects at the previous origin, like for a robotic arm or fingers on a hand. If we want to draw a set of cubes attached to a larger center cube that will rotate we can write the following:

`glLoadIdentity();`	Load the unit matrix onto the transform matrix
`glRotatef(30, 0, 1, 0);`	Multiply active matrix with a rotation 30 degrees around the Y axis.
`glutSolidCube(3);`	Draw a solid cube with side 3.
`glPushMatrix();`	Save the current matrix on the stack.
`glTranslatef(1.5, 0, 1.5);`	Multiply a translation matrix with the current matrix.
`glutSolidCube(1);`	Draw a cube with the side 1.
`glPopMatrix();`	Pop the matrix, we will get the transform back before the translation.
`glPushMatrix();`	Make another copy of the matrix
`glTranslatef(1.5, 0, -1.5);`	Make another translation.
`glutSolidCube(1);`	Draw another solid cube.
`glPopMatrix();`	Pop the stack to get back to the previous transformation.

Have a go at the program `glutshow.c`, compile and run it. You can rotate all the objects by pressing left mouse button and drag the mouse around.

You should now have adequate skills to create more advanced constructs. Begin by the example above, if you wish, and develop a more creative graphics application. To allow the colours (material colours) to change when you call `glColor*` add the following in your initialization function:

```
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
```

Create a cube in the center and six spheres, one on each side of the cube. The distance between the spheres and the cube shall be equal to the diameter of the spheres. Each sphere shall have its own colour. For each new frame the entire construction shall be rotated slightly around the vector $(0.5, 0.5, 0.2)$. The size of the cube shall vary with the sine of twice the angular rotation. The size of the cube shall be defined as $s = 0.5 + 0.3sin(2\alpha)$ where α is the angular rotation.

8 Colour

In this part we will learn more about colours in OpenGL. Colour in OpenGL is tightly connected with the colours you find on a typical monitor or projector. A more detailed description of colours in OpenGL can be found under chapter 4, Colour, in the *OpenGL Programming Guide*.

8.1 Colours in OpenGL

The image on the screen consists of pixels (picture elements). The number of pixels are determined by the resolution being used. The colour of each pixel is defined by the colour channels Red (R), Green (G) and Blue (B). During the generation of an image in OpenGL a fourth component is often used, namely the Alpha (A) channel. The three or four valued vector, RGB or RGBA, thus represents the colour value of one pixel. When the pixels are sent to the monitor, the Alpha component is usually not transmitted since displays can not be made transparent, but the Alpha value of a pixel is important for blending.

You can setup the OpenGL rendering context using either RGB, RGBA or Indexed Colours. Indexed Colours use a single pixel value that is translated by a Colour Table that holds RGB-values. This is done automatically by the hardware and a monitor/display can not tell the difference. Colour Tables were used in earlier days when memory wasn't as large as it usually is today.

The colour components R, G, B, A are defined in the floating point interval [0, 1]. Zero means no contribution and One means full contribution. However, frame buffers usually have 8 bits of precision, meaning that intensity varies from 0 to 255 for each of the components. It is unfortunately still today unusual with higher precision on consumer hardware. Higher precision, such as 16 bit floating points, can be used in off-screen frame buffers with some limitations.

8.2 RGBA mode

When you select RGB or RGBA colour modes the hardware will setup a number of bit planes for colours, usually 8 planes per component. Thus, you might get 24 or 32 bit planes. The number of colours that can be represented with 24 bit planes is 16 million, in many applications good enough, but too little in demanding applications such as video editing, image processing, and compositing of multiple images.

By calling one of the many variants of `glColor` you assign a colour to be used for all subsequent primitives until you call `glColor` with a different colour.

```
void glColor3{b s i f d ub us ui} (TYPE r, TYPE g, TYPE b);
void glColor4{b s i f d ub us ui} (TYPE r, TYPE g, TYPE b, TYPE a);
```

If you use the versions with only R, G, and B (3 elements), OpenGL will automatically set Alpha to 1.0.

You can also specify the colour to be used when you clear the frame buffer by calling `glClear-Color()`.

```
void glClearColor(GLclampf r, GLclampf g, GLclampf b, GLclampf a);
```

When you need to clear the frame buffer, you need to call `void glClear(GLbitfield mask);` and specify what buffers to clear in `mask`. To clear the colour buffer you need to have `GL_COLOR_BUFFER_BIT` set in `mask`. In the following example we set the background colour to black, all components RGBA are zero and clear the colour buffer.

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

8.3 Colour Index mode

If Colour Index mode is selected OpenGL will use a Colour Table for translation of colour index to RGB. Indexed Colours are common when you only have a smaller frame-buffer, such as 8-16 bits in total per pixel. Your cell phone might have 12 bits/pixel giving a total of 4096 colours.

By changing the Colour Table without changing the content of the framebuffer various effects can be achieved, quite common in older games.

Since the Colour Table is a part of the windowing system, you can not change the colour in OpenGL. However, you can change it using GLUT. You can use the following function to set a specific colour for a given index.

```
void glutSetColor(GLint index, GLfloat r, GLfloat g, GLfloat b);
```

8.4 Shading types

Shading covers the art of modifying the primary colour of an object depending on a range of parameters such as lighting direction, surface normals, view direction, etc. The basic shading models supported by OpenGL are Flat shading and Smooth shading. With Flat shading a primitive will have a constant colour, with Smooth shading each colour at each vertex of a primitive will be smoothly interpolated over the primitive.

OpenGL allows you to select a shader model through a call to `void glShadeModel(GLenum mode);` You may specify either `GL_FLAT` or `GL_SMOOTH`.

Note that specifying Flat or Smooth shading isn't enough, we will need more operations to activate lighting. However, Flat or Smooth shading affects how vertex data will be interpolated over a primitive.

8.5 Lighting, Shading

To achieve real Shading we need to activate Materials in OpenGL, instead of primary colours provided by `glColor`. You can find more information about Lighting in chapter 5 of the *OpenGL Programming Guide*.

9 Lighting in OpenGL

Real world lighting is a very complex interaction between materials and light (photons) at different wavelengths. In computer graphics we have initially restricted the colour space to RGB, usually. In order to determine the pixel colours in reasonable time we have to make a number of crude approximations and simplifications.

The resulting colour of a pixel is determined by separating the lighting formula into four separate components: Ambient, Diffuse, Specular and Emissive contribution.

Ambient represents the light that comes from everywhere, it has bounced all around the space such that we can not tell where it comes from.

Diffuse lighting is computed from light from a specific direction but is spread evenly in all directions. Diffuse light thus looks the same from all viewing directions.

Specular light is the light that hits a surface from a specific direction and bounces off the surface in a specific direction. Present in glossy plastic materials and polished metals.

Emissive light represents the light that is emitted from an object regardless of the received light. Note that a primitive drawn with emissive light in itself is not a light source and will thus not light other objects. This can be simulated by placing a lightsource (point light) near the emissive surface.

These four components are computed separately and summed together. A light can affect three of the components and they have three parameters that is multiplied to each contribution. The colour of the light also affects the final colour of a pixel (or a fragment, as it is called before it ends up in the frame buffer).

9.1 Lighting a Scene

An outline for the steps involved to light a scene.

1. Create objects and define their surface normals for each vertex.
2. Create and place a number of light sources.
3. Choose a colour model.
4. Define material properties for the objects.

Lighting is by default turned off in OpenGL. To turn it on, you must call `glEnable(GL_LIGHTING)`. Note that when lighting is turned on, the colour of an objects comes from its material colour, not the primary colour you defined using `glColor`.

9.2 Normals

Most shading operations involve the surface normals. You must specify these to OpenGL since OpenGL does not compute any normals by itself. A normal is specified for a vertex by first calling one of:

```
void glNormals{bsidf}(TYPE nx, TYPE ny, TYPE nz);
void glNormals{bsidf}v(const TYPE *v);
```

The following example illustrates how to specify normals.

```
/* Assign three normals (all being the same) */
GLfloat normal0[] = {0.0, 0.0, 1.0};
GLfloat normal1[] = {0.0, 0.0, 1.0};
GLfloat normal2[] = {0.0, 0.0, 1.0};

/* Defined three vertex coordinates */
GLfloat vertex0[] = {-1.0,  1.0, 0.0};
GLfloat vertex1[] = { 0.0, -1.0, 0.0};
GLfloat vertex2[] = { 1.0,  1.0, 0.0};

/* Draw a triangle */
glBegin(GL_TRIANGELS);
 glNormal3fv(normal0);
 glVertex3fv(vertex0);
 glNormal3fv(normal1);
 glVertex3fv(vertex1);
 glNormal3fv(normal2);
 glVertex3fv(vertex2);
glEnd();
```

[Note: Most OpenGL functions generate an invalid exception if called between a `glBeginglEnd` pair. This is always specified in the corresponding function reference.]

Since OpenGL is a state machine you could also have given the first normal and then specified the three vertices.

Table 2: Parameters for a light source.

Attribute	Default	Meaning
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	RGBA intensity for ambient component
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	RGBA intensity for diffuse component
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	RGBA intensity for specular component
GL_POSITION	(0.0, 0.0, 1.0, 1.0)	Light source position (x,y,z) and type (w, directional or point/positional).
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0, 0.0)	(x,y,z,w) spotlight direction
GL_SPOT_EXPONENT	0.0	Spotlight exponent
GL_SPOT_CUTOFF	180.0	Spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	Constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	Linear attenuation factor
GL_QUADRIC_ATTENUATION	0.0	Quadratic attenuation factor

```
/* normal0 will be applied to all vertices */
glBegin(GL_TRIANGELS);
 glNormal3fv(normal0);
 glVertex3fv(vertex0);
 glVertex3fv(vertex1);
 glVertex3fv(vertex2);
glEnd();
```

In order for the lighting calculations to be correct, you must provide normalized normals (kind of obvious). The normals will remain normalized through all transforms as long as you stick to uniform scaling. Otherwise you will need to tell OpenGL to re-normalize the normals by using `glEnable(GL_NORMALIZE)`. However, this will most likely slow down your rendering.

9.3 Light sources

A light source has a number of attributes such as colour, position, and direction. To specify these attributes you use the following functions:

```
void glLight{if}(GLenum light, GLenum pname, TYPE param);
void glLight{if}v(GLenum light, GLenum pname, TYPE *param);
```

`glLight*()` assigns parameters for the selected light source, `light`, being one of `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7`. The functions specify the parameter indicated by `pname`. The parameter is assigned the value given in `param`.

Table 2 shows available parameters.

For a light source to effect the rendering of scene it must be turned on. This is done by calling `glEnable(GLenum light)`. To turn it off, call `glDisable(GLenum light)`.

```
/* Some light parameters */
GLfloat light0_amb[]= {0.4, 0.2, 0.1, 1.0};
GLfloat light0_diff[] = {0.4, 0.2, 0.1, 1.0};
GLfloat light0_spec[] = {0.6, 0.8, 0.2, 1.0};
GLfloat light0_pos[] = {10.0, 10.0, 10.0, 1.0};
GLfloat light0_const_att = 0.5;
```

```
/* Assign some parameters for light 0 */
glLightfv(GL_LIGHT0, GL_AMBIENT, light0_amb);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diff);
glLightfv(GL_LIGHT0, GL_SPECULAR, light0_spec);
glLightfv(GL_LIGHT0, GL_POSITION, light0_pos);
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, light0_const_att);
```

The colour components of the light source is multiplied with the colour components from the primitive. For example, to add 20% ambient lighting contribution set the ambient colour of the light to (0.2, 0.2, 0.2, 1.0).

9.4 Lighting model

The Lighting model has three components.

1. Global ambient light,
2. Placement of the camera, at infinity or within the scene, and
3. Whether lighting computations are done at front, back or both faces of the primitives.

Use the following function to set the these components.

```
void glLightModel{if}(GLenum pname, TYPE param);
void glLightModel{if}v(GLenum pname. TYPE *param);
```

Table 3 displays the values available for `glLightModel(*)`.

9.5 Material properties

The lighting model in OpenGL computes the fragment colour of a primitive by multiplying the material properties with the incoming light. This is done per lighting component separately and the Material properties have unique colours defined for Ambient (A_M), Diffuse (D_M), Specular (S_M) and Emission (E_M). With the properties for a light as in table 2 we can write a somewhat complete lighting formula for one light source.

$$\begin{aligned} C_F &= E_M + A_M A_{LM} + \\ &\quad \tfrac{\text{spoteffect}}{k_c + k_l d + k_q d^2} \left(A_M A_L + \max(L \cdot n, 0) D_M D_L + \text{shininess} \cdot \max(s \cdot n, 0) S_M S_L\right) \end{aligned}$$

The terms spoteffect and shininess represent spotlight effects and specular shininess parameters. The vectors n, L and s represent the surface normal, the incoming light vector and the light vector reflection, which also depends on the view point. The distance d is the distance between the light source and the fragment position.

To set the material properties you must call `glMaterial*` and set the corresponding material property.

Table 3: Attributes for the lighting model

Attribute	Default	Meaning
GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	RGBA values for global ambient light.
GL_LIGHT_MODEL_LOCAL_VIEWER	0.0, GL_FALSE	How to compute specular light.
GL_LIGHT_MODEL_TWO_SIDE	0.0, GL_FALSE	Single sided or double sided lighting.

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE param);
void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);
```

`face` is one of GL_FRONT, GL_BACK eller GL_FRONT_AND_BACK. `pname` specifies which property to set and `param` carries the property values. Table 4 displays the available properties.

Each call to `glMaterial*` will set any of the property above, to set multiple values you have to call the function several times.

```
GLfloat mat_amb[] = {0.4, 0.2, 0.1, 1.0};
GLfloat mat_diff[] = {0.4, 0.2, 0.1, 1.0};
GLfloat mat_spec[] = {0.6, 0.8, 0.2, 1.0};

glMaterialfv(GL_FRONT, GL_AMBIENT, mat_amb);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diff);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_spec);
```

For each of the lighting components you can view the values as a contribution percentage of the incoming light per colour channel. For instance, the diffuse colour will reflect 40% of the incoming red channel, 20% of the green channel and 10% of the blue channel, for a white lightsource it might look a bit orange.

9.5.1 Flat and Smooth Shading

The Shading Model described above also applies to lighted materials. In fact, the fragment colour is initially computed at the vertices and then interpolated over a primitive (Smooth Shading). In other words, the lighting formula above is only evaluated at the vertices and the resulting colours C_F are interpolated. Since the normals are not interpolated and the lighting formula is not evaluated per fragment, objects with few polygons will produce poor specular lighting effects. Smooth shading thus implements Gouraud shading in OpenGL.

9.5.2 Shader programming

For more accurate, and per fragment/pixel, lighting it is now common to implement this in Fragment Shaders, using OpenGL ARB Fragment Program, OpenGL ARB Fragment Shader (OpenGL Shading Language / OpenGL 2.0), or NVIDIAs Cg.

Table 4: Material Lighting Parameters

Attribute	Symbol	Default	Meaning
GL_AMBIENT	A_M	(0.2, 0.2, 0.2, 1.0)	Ambient colour
GL_DIFFUSE	D_M	(0.8, 0.8, 0.8, 1.0)	Diffuse colour
GL_AMBIENT_AND_DIFFUSE	A_M, D_M		Ambient and Diffuse Colour
GL_SPECULAR	S_M	(0.0, 0.0, 0.0, 1.0)	Specular colour for the material
GL_SHININESS	shininess	0.0	Specular exponential
GL_EMISSION	E_M	(0.0, 0.0, 0.0, 1.0)	Emissive Colour

10 Textures in OpenGL

As you are encouraged to use textures in this lab use the list of commands below to get you started. Note that this is far from all the functions available or even neccessary but some of them are essential. [Note: Read up on the functions in any reference and also in what order they should be called (as they are now in alphabetical order).]

```
glActiveTexture() glBindTexture() glDisable() glEnable()
glGenTextures() glTexEnvi() glTexImage2D() glTexParameteri()
```

For `glTexEnvi` the following parameters might be neccessary

```
GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE
```

The Internet is flooded with good texture tutorials, feel free to use these (in a respectful way). For now it might be a good idea to copy some simple file reading function (or maybe library) to get your images as a character string and then import them to OpenGL yourself. It is not very hard to write your own reader for a simple format, e.g. targa, but it is out of the scope for this lab.

11 A Planetarium

We hope you have gained some insights on how the OpenGL pipeline works. However, it is your responsibily to learn and if you need additional information on OpenGL there are plentiful of references online. Early versions of the OpenGL Programming Guide can be found online unless you buy this useful book.

Now continue to work on the exercises below.

The planet sizes shall be configured by parameters, i.e. variables. The same shall be applied for angular velocities, etc. Please use structures to store data about planet objects and implement the drawing of planets and bodies in separate functions. Your planetary simulation must not be 100% correct. You should use the available gl-functions to position the celestial bodies, not trigonometric cosine functions.

Task 1:
Start by creating a sun in the center of your universe. The sun is of course not affected by lights and will, instead, shine on your planets. In other words, place a point light source at the center of your sun.

Task 2:
Now add a planet at some suitable distance, say $R/2$ from the sun. The planet should rotate around the sun in 4500 frames. The planet axis is 10 degrees off the sun rotation axis and your earth should do one rotation every 150 frames. Your planet should be illuminated by your sun and the ambient lighting must be very faint or off. You are encouraged to use textures for colouring the earth, there are many textures available on the Internet.

Task 3:
Also add the Earth's trajectory around the sun using at least a Line Loop. The trajectory shall not be lit and your program must support to turn it off and on, using the key `o`, as in orbit. Here cosines might be useful for placing your vertices. You may create a more fancy trajectory if you wish. To show the tilt angle of the Earth as it travels around the Sun, thus giving us seasons, add an axis in the form of a non-lit line through the Earth. The line should extend some distance from the poles.

Task 4:
In the next step you shall add a small moon to orbit your Earth. The Moon is light grey and shall be lit by the sun. The moon shall complete an orbit in 750 frames. You'll easily find moon textures on the Internet as well. The Earth does not need to cast a shadow on the Moon (and vice versa).

Task 5: (optional)
In addition, also create a spherical shell of stars. They shall be of varying sizes, at least 1 and 2 pixels wide, and varying intensity. The stars shall be randomly placed within a spherical shell between R and $2R$, however, they must not be randomly placed per frame. If you can create uniform distribution of the stars in the shell it is preferred. A couple of thousand stars (up to 10000) is easily doable.

Task 6: (optional)
If you are not using textures for your planetary bodies you will need to create your planets yourself by randomly varying the colours smoothly between green (land) and blue (sea) for the Earth and two grey colours, light grey and dark grey, for the Moon.

Tip: Minimize the computation at run-time by storing precomputed coordinates and vertex data in arrays and reuse these when you draw your objects.

A sphere can be constructed using either `glutSolidSphere` or if you need per vertex control by constructing two triangle fans (`GL_TRIANGE_FAN`) and a number of quad strips (`GL_QUAD_STRIP`). Create all vertex data initially and put them in one or more arrays. Then you draw your objects by looping over the arrays or use OpenGLs Vertex Arrays functions.