Computer graphics Labs: OpenGL (1/2)
Geometric transformations and projections

Exercise 1: Geometric transformations

(Folder transf contained in the archive OpenGL1.zip and available on the web page of this course on http://www.cgeo.ulg.ac.be/infographie/)

Project creation: Makefiles

The project/Makefile can be generated using CMake. To do so:

1. Open a shell from the directory in which you want to create the project or Makefile (for instance, in the directory ‘build’ from the sources directory).
   - For a CodeBlocks project, write the command:
     cmake –G "CodeBlocks - Unix Makefiles" ..
   - To obtain a Makefile, write the command:
     cmake ..
   where … designates the path to the sources directory.

Designing the base object

1. In this first exercise, the base object is a square. It will be designed as a polygon.

   Method outline:
   - To draw a polygon on the screen, it is needed to insert its vertices as follows:
     ```c
     glBegin(GL_POLYGON);
     glVertex3f(x1, y1, z1);
     glVertex3f(x2, y2, z2);
     ...
     glEnd();
     ```
   The arguments x1, y1, … are floats specifying the position of the added vertex. The 3f suffix of the glVertex3f command means this command can take three float-type arguments. Numerous OpenGL commands have several variations depending on the type and number of arguments. For example, glVertex2i (2 integers) and glVertex4d (4 doubles) are also available.

   Application:
   - In the (x,y) plan, draw a unit square centred at the origin by adding the commands to the display() function.
   - Compile your program and execute it.

2. Adding colours

   Method outline:
   - Defining an object colour is done through using a command, such as:
     ```c
     glColor3f(r, g, b);
     ```
   Because OpenGL functions according to the principle of a state machine, it is first needed to define the colour that is to be assigned to the object before defining the object itself. The glColor* commands allow defining the current colour that is to be applied to all objects created afterwards. A 4-argument command is also available in order to set the alpha channel value.
In the current state of the code, it is possible to assign only one uniform colour to the polygon. The colour assigned to the polygon will be the one active when the glEnd() command ends the polygon creation.

**Application:**
- Add for instance the command glColor3f(1.,0.,0.) before defining the square in order for it to be drawn in red.
- Assign a different colour to each vertex. To do so, it is needed to change the ‘shade model’ by replacing the argument of the command glShadeModel by GL_SMOOTH in the function main().
- As for the polygons, the colour of a vertex is defined at its creation by the current colour. By changing the current colour using glColor3f before each vertex creation, you may assign a different colour for each vertex of the polygon.

3. Drawing in wireframe mode.
- OpenGL allows representing easily a scene in wireframe mode. To do so, it is enough to make use of the command:
  ```cpp
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
  ```
  The GL_FRONT_AND_BACK argument means that both faces of the polygons are impacted by the command. The second argument corresponds to the drawing mode GL_LINE, which gives a wireframe render whereas GL_FILL gives filled polygons.
- Modify the keyboard function to have the ‘z’ key switch from one mode to the other. (Tip: the glutPostRedisplay() command allows refreshing the image wherever you are in the code).

**Applying geometric transformations**

**Introduction to the method:**
The final aim of this exercise is to create an animation in which the previous drawn square will ‘orbit’ the image centre while turning on itself (like the Earth around the sun). First of all, we will set the necessary different geometric transformations before animating anything. As described in the theory, OpenGL geometric transformations are carried out in a matrix fashion. Given that a vertex is represented by a 4-components p column vector, its transformation using the M (4x4) matrix is obtained by the matrix multiplication:

\[
p' = M.p
\]

Successive transformations (rotation, scaling, translations…) can be combined in a single transformation matrix. For instance, if we want to successively apply the transformations \( M_1 \), \( M_2 \) and \( M_3 \), it is needed to calculate the product of three matrices and to multiply the vertex by the resulting matrix:

\[
p' = M_3 . M_2 . M_1 . p
\]

When using OpenGL, transformations are also defined and carried out according to the principle of a state machine. Before creating an object, we define the transformation matrix that will be applied to it. To do so, we will incrementally introduce base transformation matrices; OpenGL will then carry out the necessary multiplications in order to get the complete transformation matrix.
There is, though, an idiosyncrasy to OpenGL: transformation matrices are to be introduced the other way round. For instance, if we wanted to carry out the transformation described by the previous equation, it would be needed to begin from inserting $M_3$ and then multiply the current matrix by $M_2$; finally, multiply the current matrix by $M_1$.

The useful commands are the following:

- `glLoadIdentity()`: allows reinitialising the transformation matrix.
- `glLoadMatrixf(float *)`: allows defining the current transformation matrix.
- The argument is a 16-float table containing the **columns** of the matrix.
- `glMultMatrixf(float *)`: allows multiplying (on the right) the current transformation matrix. The argument is a 16-float table containing the **columns** of the matrix.

The base transformation matrices are given as reminder hereafter:

- **Scaling**
  
  $$
  \begin{bmatrix}
  S_x & 0 & 0 & 0 \\
  0 & S_y & 0 & 0 \\
  0 & 0 & S_z & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  $$

- **Rotations**

  $$
  R_x = \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & \cos \theta & -\sin \theta & 0 \\
  0 & \sin \theta & \cos \theta & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  \quad R_y = \begin{bmatrix}
  \cos \theta & 0 & \sin \theta & 0 \\
  0 & 1 & 0 & 0 \\
  -\sin \theta & 0 & \cos \theta & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  $$

  $$
  R_z = \begin{bmatrix}
  \cos \theta & -\sin \theta & 0 & 0 \\
  \sin \theta & \cos \theta & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  $$

- **Translation**

  $$
  \begin{bmatrix}
  1 & 0 & 0 & T_x \\
  0 & 1 & 0 & T_y \\
  0 & 0 & 1 & T_z \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  $$

**Application:**

Use the geometric transformations to modify the previously written program in order to get an image that fulfills the following sketched requirements:
Use the above presented transformation matrices. Don’t forget that the next step consists in animating the image based upon these transformations. It is thus a good idea to parameter transformations using variables.

**Animation**

**Principle:**
We will now animate the image we have just obtained. The animation process consists in creating a loop in which we draw again completely the image while slightly changing its transformation parameters.

This loop could be directly carried out in the display function, but this would cause it to block the program on the whole duration of the animation (it is not possible to resume it using only the keyboard for example).

It is possible, though, to define a function that will be executed as long as the program has nothing else to do.

**Application:**
1. Define a `void idle()` and add the main `glutIdleFunc(idle)` command to the function. This indicated to glut that the function to use when the program is not busy.
2. The role of the idle function is to slightly modify the transformation parameters (defined by global variables) and to refresh the display by invoking `glutPostRedisplay()`.
3. Tip: add the following lines to the idle function.
   ```c
   int t = glutGet(GLUT_ELAPSED_TIME);
   int passed = t - t_old;
   t_old = t;
   revolution_angle += passed/1000.0 * v_revolution;
   rotation_angle -= passed/1000.0 * v_rotation;
   glutPostRedisplay();
   ```
   
   Where `v_revolution`, `v_rotation` and `t_old` are global variables setting the revolution and rotation speeds. `revolution_angle` and `rotation_angle` are global variables setting the corresponding angles. For example, in the above figure, these angles are, respectively, 30° and 20°.
4. Modify the above code in order to avoid the overflow of the variables `revolution_angle` and `rotation_angle` (see e.g., the `fmod` function on cppreference).
5. At the moment, the proposed animation has probably a huge flaw. It happens indeed that the used render mode produces black strips on the screen and a chopped image. This is caused by the program not being synchronised with the screen display. If the program is writing the buffer (erasing it, for instance) while the screen is displaying the previous image, the image will appear like it is chopped. To avoid this, the display buffer can be split into two: one of the buffers is used by the program for writing and the other is used by the display itself. The command glutSwapBuffers() allows, in this case, exchanging buffers while waiting for the image display to be over.
   - For activating this functionality, it is first needed, when initialising, to ask glut to initialise the second buffer using the command:
     
     ```
     glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
     ```
   - Instead of:
     ```
     glutInitDisplayMode(GLUT_RGB);
     ```
   - Then, you may add the glutSwapBuffers() command at the end of the display() function.

6. For the program to be more interactive, you may add, for instance, new keyboard shortcuts that allow controlling both rotation speeds.

7. The geometric transformations being very frequently used, it is obviously uneasy to manually define each time the transformation matrices. For this reason, the OpenGL library offers several functions that allow carrying out directly base transformations, as follows:
   - `glTranslatef(Tx,Ty,Tz)`: operates a translation on three axes according to three float parameters given above.
   - `glRotatef(angle,ex,ey,ez)`: carries out an axis rotation `[ex,ey,ez]`, the angle being expressed in degrees.
   - `glScalef(sx,sy,sz)`: scaling with a (different) coefficient for each axis.

In order to check the implementation of your transformation matrices, substitute the upper commands to these (comment your previous code). Be sure you are correctly initialising the transformation matrix (`glLoadIdentity()`).

**Viewport management**

**Presentation**

The viewport is of the part of the window that is dedicated to the display of the image. Its dimensions condition the ratio aspect (that is, the width/height relation) of the obtained image. This command allows defining how the projected scene will be transformed into a pixel table. More precisely, `glViewport` defines the transformation that allows switching from the normalised space to the windows coordinates. This command accepts 4 arguments:

- `int x, int y`: specifies the position of the lower left corner of the normalised space in the window (in pixels).
- `int width` and `int height`: specifies the size of the normalised space in the window (also in pixels).

By doing so, it is possible to define more than one ‘viewport’ for each window. Currently, the resizing of the window is managed by the function `reshape(int x, int y)` that uses the command `glViewport`. The disadvantage of implementing it is that, when resizing the window, the height/width viewport ratio is not fixed. The consequence is the image being deformed (the square ceases to be a square and the circular orbit is not circular anymore).
**Application:**
The following modification allows solving this problem by keeping viewport height and width the same size:

```
if (x<y)
    glViewport(0,(y-x)/2,x,x);
else
    glViewport((x-y)/2,0,y,y);
```
Exercise 2: drawing a cube

(Folder cube contained in the archive tplogl.zip and available on the class web page at http://www.cgeo.ulg.ac.be/infographie/)

Drawing the base object

1. The cube is drawn through successively representing each of its 6 faces using a square according to the method described during the previous tutorial.

   **Method outline:**
   - Reminder: to draw a polygon on the screen, insert vertices as follows:
     ```
     glBegin(GL_POLYGON);
     glVertex3f(x1, y1, z1);
     glVertex3f(x2, y2, z2);
     ...
     ...
     glEnd();
     ```
     The x1, y1 ... arguments are floats that specify the added vertex position.
   - To clarify the algorithm, use a first 2-dimension table that allows storing the positions of the 8 vertices of the cube. A second 2-dimension table storing (in a position table) index of the vertices of each face will then enable you to reach the positions of the summit of a face.

   **Application:**
   - Draw a 1-unit cube centred at the origin by adding the commands to the function `display()`.
   - Compile your program and execute it.

2. Adding colour

   **Method outline:**
   - Reminder: defining an object colour is done by using the command:
     ```
     glColor3fv(float [3]);
     ```
     this command is to be used according to the principle of a state machine. It defines the current colour that is the one that will be used for all objects created before any new use of this command.
   - In the same way as explained before, use a 2-dimension table for defining the faces colours.

   **Application:**
   - Define a different colour for each face.

3. Activating the depth test.

   - If you run your program at this stage and if you manipulate the cube using the mouse, you will notice quite quickly that there is a problem. It comes from OpenGL which draws polygons according to the commands order without taking into account interactions (hidden parts) between those.
   - Thankfully, it is possible to add an automatic depth test using the command:
     ```
     glEnable(GL_DEPTH_TEST);
     ```
     that is to be set in the initialisation section.
   - This sole command is not enough because the depth test depends on an extra buffer called ‘depth buffer’ or ‘z-buffer’. It is thus necessary to initialise the depth buffer at the beginning of the program and to erase it each time the image needs to be refreshed. Modify consequently the glutInitDisplayMode command in the initialisation section as follows:
glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);

and in the display function, the glClear command,

```c
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

- Execute the program to check the result.

## Adding a simple lighting

So far, the lighting was not considered. We will now add a light source to the scene for it to be more realistic.

1. Defining a light source.

   **Method outline:**
   - The lighting calculation is enabled by adding in the initialisation section the command:
     ```c
     glEnable(GL_LIGHTING);
     ```
     now that the lighting calculation is enabled, it is needed to define a light source for the scene in order for the cube not to be black during the rendering.
   - OpenGL allows using simultaneously 8 light sources. Each one has an identifier such as GL_LIGHTi where i is an integer between 0 and 7. The use of a lamp requires its activation by specifying its identifier through the command:
     ```c
     glEnable(GL_LIGHT0);
     ```
   - The command `glLightfv(GL_LIGHT0, param, float [4])` allows then defining the different parameters associated with the light source. The possible values of `param` are, for instance:
     - `GL_POSITION` defines the lamp’s position.
     - `GL_DIFFUSE` defines the diffuse light’s colour.
     - `GL_SPECULAR` defines the specular light’s colour.
     - `GL_AMBIENT` defines the colour of the contribution of the light source to the ambient light.

   The given table, for all these parameters, should be of size 4. The four values correspond to the coordinates x, y, z and w for one position and for the RGB triplet, plus the alpha channel for one colour.

   **Application:**
   - Add a white light source (diffuse light) at position (0, 0, -100, 1).
   - If you are executing the program at this stage, the displayed cube will poorly illuminated and greyish.
For the lighting to be properly calculated, OpenGL needs to know normals at each vertex from each surface. For now, the normal vectors are not defined because they are set at their default value according to the z axis. The next stage will thus be to define the normals.

Finally, the lighting calculation does not take into account the colours defined by glColor, hence the greyish colour of the cube. For the cube to be coloured with the lighting calculation, the last step is thus to define a material for each face of the cube.

2. Defining normals.

**Method outline:**
- Defining a normal at a vertex is done through the principle of a state machine. Begin from defining the current normal using the command `glNormal3fv(float [3]);`
- this normal will then be applied to all vertices created afterwards.

**Application:**
- When creating faces, define the normal vector to the face for each vertices of it. You shall use a table defining the normal for each faces.

3. Material definition.

**Method outline:**
- Defining a material is also carried out according to the principle of a state machine. To modify the current material, it is enough to use the function `glMaterialfv(face, param, float [4]).`
  - The argument `face` determines which faces will be influenced by the property modification. Usually, we will use `GL_FRONT`.
  - The modified parameter is designed by `param`. For example, `GL_DIFFUSE` modifies the diffuse light’s colour reflected by the surface.
  - A 4-floats table, corresponding to the RGB triplet plus the alpha channel.

**Application:**
- Modify the current material when creating each of the faces in order to retrieve the colour previously used.
- Can you identify an effect of the lighting calculation on the result?
Applying a texture

4. Defining texture and loading of an image

Method outline:

- Before using textures (2D), it is necessary to activate them in the initialisation section with the instruction:
  \[ \text{glEnable(GL\_TEXTURE\_2D);} \]
- Then, the next step consists in initialising a certain number of textures using the following command:
  \[ \text{glGenTextures(num, textureIdx);} \]
  where \( \text{num} \) indicates the number of textures you would like to initialise. \( \text{textureIdx} \) is a GLuint table of length \( \text{num} \) in which OpenGL will set indices of the created textures.
- The texture index then allows to make the texture active:
  \[ \text{glBindTexture(GL\_TEXTURE\_2D, textureIdx[0]}); \]
- When activated, we can add an image for the texture. The software has an in-built function (loadTiffTexture) that allows loading a tiff-format image into the global variable \( \text{image} \). It is then needed to give it, as an argument, the access path to the file that is to be charged.
- Then, you insert the image into the active texture using the command:
  \[ \text{glTexImage2D(GL\_TEXTURE\_2D,0,GL\_RGB,256,256,0, GL\_RGB,GL\_UNSIGNED\_BYTE, image);} \]
  where arguments designate in the order:
  - \text{GL\_TEXTURE\_2D}: type of defined texture.
  - \text{0}: number of texture level (useful only when using multi-resolution textures).
  - \text{GL\_RGB}: internal storing format of the texture.
  - \text{256 and 256}: size (width and length) of the image.
  - \text{0}: the texture has no edges (1 elsewhere).
  - \text{GL\_RGB}: the given image format.
  - \text{GL\_UNSIGNED\_BYTE}: the storing type of the given image.
  - \text{image}: a pointer to the data.
- Finally, it is still necessary to define how OpenGL will filter the texture.
  - The filtering stage is necessary due to the form and dimensions changes, one pixel rarely match the same pixel on the texture of the base image. Defining the filter allows choosing how the colouring of pixels will be attributed.
  - The filtering is defined using two instructions:
    \[ \text{glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER,method);} \]
    \[ \text{glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER,method);} \]
  the value \( \text{method} \) may be \text{GL\_NEAREST}; if so, the pixel colour on the screen is the same as the colour of the nearest pixel in the (most quickly obtained) deformed texture. You may also use \text{GL\_LINEAR}, which is a linear interpolation between four neighbouring pixels in the deformed texture (better result).
Application:
   o Insert the necessary commands for defining a texture on the basis of the image ‘texture.tif’ in the initialisation section.
5. Defining the ‘texture’ coordinates on the object.

Method outline:
   o For applying the texture on the object faces, it is needed to define the orientation and position of the texture on each face. This is carried out by defining texture coordinates in each of the vertices of the face.
   o In the present case, the faces being square, this is quite simple. By using the proposed coordinates as indicated in the given figure, the results are usually quite good.

![Figure 1: texture coordinates](image)

The texture coordinates of a vertex are fixed by introducing, before creating the vertex, the command:
```c
glTexCoord2fv(float [2]);
```

Application:
   o Disable the command that allows assigning the material to the vertices (glMaterialfv…).
   o Define the texture coordinates according to the figure for each of the faces and check the result.
   o Modify the coordinates and watch the result.
**Exercise 3: Introduction to shaders**


**Introduction to the dynamic pipeline**

Initially, the graphic pipeline used for the rendering on graphic cards was fixed. If the graphic card manufacturers could optimize the hardware architecture, users would be unable to modify any algorithm. Only calculation on the CPU (Central Processing Unit) could overcome this problem.

The previous exercises fit into this framework. We were then using a set of functionalities provided by OpenGL to interface the graphic card.

Under the pressure of the market for films and video games, the development of configurable graphic cards changed the graphic pipeline into a dynamic pipeline using programmable “shaders”.

A shader can be defined as a small software piece executed by the GPU (Graphical Processing Unit), written in a language close to C, executing a part of the computations needed for the rendering. There are several types of shaders. Here, we will take a closer look to mainly two types: the **Vertex Shaders**, executed on each vertex of a mesh to display, and the **Fragment Shaders** (also called **Pixel Shaders** for Microsoft DirectX), executed for each displayed pixel.

The algorithm here below shows were the shaders operate inside the pipeline of a standard rendering.

Rendering pipeline for the display of a triangle mesh.

```
Given M a triangle mesh.
For each triangle T of mesh M
    | For each vertex S of triangle T
    |    | Transform S into the camera frame
    |    | Project S on the camera projection plane
    |    | Compute the illumination of S
    | End For
    | For each pixel P of triangle T
    |    | Compute the colour of P
    |    | Compute the depth of P
End For
```

The use of shaders makes obsolete some functionalities used during the previous exercises, such as geometric transformations, the illumination functionalities and depth test. Despite the seeming complexity caused by these changes, GPU programming opens a new field for customizing visual effects. We will henceforth work in the framework of the “all shader” initiated by version 3.0 of OpenGL.

Before using shaders programming, we will introduce the data management needed for this new graphic pipeline.
**Data management: VBOs and VAOs**

The first step before entering the graphic pipeline is to provide OpenGL the geometry to be stored on the GPU. During the previous exercises, we used a specific primitive for displaying a square (GL_POLYGON) together with the command `glBegin [...] glEnd`. However, graphic cards only manage points, segments and triangles. These functionalities have therefore been removed from OpenGL. The scene needs to be triangulated before to be displayed on the screen. Thus, a square is the union of two triangles, themselves made of three vertices (vertex in the singular).

In OpenGL, a vertex is made of a set of attributes, as its position, colour, normal, texture coordinates, and so on. It is then possible to associate any data type (of geometric character or not) to a vertex. The only limitation is that these data must have a numeric representation (e.g., a temperature, a force vector).

Once the set of attributes defined, the vertex list has to be stored inside a storage space of the graphic card called **Vertex Buffer Object (VBO)**. The data storage management in VBOs provided by OpenGL is very flexible. For instance, consider a triangle where each vertex contains data on its position and colour (figure 1). Several storage options are available: 1a) non-interleaved with two VBOs, 1b) non-interleaved with one VBO, or 1c) interleaved with one VBO.

![Figure 2: Data management of VBOs.](image)

VBOs allow an efficient storage of the data, but they do not suffice in themselves. OpenGL does neither know what is the type of the stored data inside the VBO, nor how to regroup them in order to interpret them. The solution is to use a **Vertex Array Object (VAO)** in order to give OpenGL enough information for interpreting the scene. A VAO stores the active attributes, the storage format (interleaved or no inside the active VBO), as well as the format (4 floating points for the position in homogeneous coordinates).

The dynamic pipeline requires data to be transmitted to the graphic card, but also needs to be told how these data have to be processed. This role is devoted to the shaders. These small software pieces are typically used for computing images (mainly 3D transformations and illumination). However, they can also be used for other computations (physical simulations, digital arts...).

The following figure illustrates the main interaction with the graphic pipeline.

![Figure 3: Inputs/outputs of the dynamic pipeline](image)
Through the following paragraphs you will learn how to use VBOs and VAOs and will load and then program your first shaders.

**Creating a square**

For this exercise, we will geometrically represent a square.

1. Creating the geometry.
   - Start by describing the mesh vertices inside an array.
     - For defining the vertex list, 4 homogeneous coordinates for each vertex are needed. In the file `square.cpp`, inside the function `UpdateVertexBuffer`, begin to allocate the variable `vertexData`. This variable will store the data list associated to the mesh nodes (for the moment only coordinates) with floating data points (example: 0.75f). Firstly, we will duplicate the vertices shared by the two triangles. Also, we will sort the data by triangle. Compute the array length to declare. We will store this array length into a variable called `size` which will be reused into that function.
     - Then, at the end of the function, add the following command in order to free the storage associated to the array.
       ```
       delete[] vertexData;
       ```
     - Finally, inside the function `BuildVertexData`, initialize the array data with the corresponding coordinates of the vertices following the figure below:

   ![Figure 4: Mesh of the square](image)

2. Sending data to the GPU thanks to the VBO.
   - Although the data have been generated, they cannot be directly used by OpenGL. By default, OpenGL does not have access to the data stored. Therefore, the first task at hand consists in assigning a storage space visible to OpenGL and then fill-in this space with our data. This operation is performed thanks to the buffer introduced earlier: the VBO. Note that the notion of VBO can be extended to data different from vertices. We then simply refer to **Buffer Object**. This type of object is instanced and managed by OpenGL. The user can control this storage only indirectly, but benefit from the fast GPU storage access.

   **Introduction to the method:**
   - In order to be handled, almost all the different OpenGL objects (buffers and others) are identified by an unsigned integer (of type `GLuint`).
   - The first step consists in generating the identifiers of the objects to be created. To this aim, a command in the following fashion is used:
     ```
     glGen*(nb_objects, GLuint_ptr)
     ```
     Where the symbol `*` should be replaced by the object type, `nb_objects` corresponds to the number of objects to create, and `GLuint_ptr` corresponds to the address of the identifier. The identifier targeted by the pointer is then generated, but without allocating storage for the object.
The object is then linked to a context thanks to the function:

```c
glBind*(target, GLuint)
```

The symbol * should be replaced by the object type. The parameter target, chosen among a list of admissible targets (depending on the context), allows to change the function’s behaviour.

- We can then allocate storage to the object depending on the data to store.
- Finally, we break the previously created link between the object and the context by replacing the object address by 0 inside the command:

```c
glBind*(target, GLuint).
```

- The last stage consists in freeing the storage. This is done with a command in the following fashion:

```c
glDelete*(nb_objects, GLuint_ptr)
```

Its use is similar to its dual `glGen*`.

### Application:

- The identifier (of type GLuint) for the buffer object used for storing the vertices is `vertexBufferObject`.
- The initialization of the buffer corresponding to the identifier `vertexBufferObject` is performed in the function `InitializeVertexBuffer`.
- Create one object associated to `vertexBufferObject` with the function `glGenBuffers(GLuint, GLuint*)`.
- Then, go to the end of the function `UpdateVertexBuffer`, just before `delete[] vertexData`.
- Link the object `vertexBufferObject` to the target `GL_ARRAY_BUFFER` thanks to the command `glBindBuffer`.
- You can then allocate storage for the object in order to store the array `vertexData` by adding the command:

```c
glBufferData(GL_ARRAY_BUFFER, size*sizeof(float), vertexData, GL_STATIC_DRAW);
```

This command allows to dimension the GPU storage to allocate the size `size*sizeof(float)`, where `size` is the number of elements inside the array `vertexData`, and then to copy the data contained in `vertexData`.

- Unlink the object `vertexBufferObject` from the target `GL_ARRAY_BUFFER` by calling again the function `glBindBuffer` and by replacing the object address by 0.
- Finally, in the function `DeleteVertexBuffer`, add the command which frees the allocated resources to `vertexBufferObject`.

3. **Identifying the data of the VBO via the VAO.**

We have just sent the vertex data to the GPU storage. However, the **Buffer Objects** are not formatted. For OpenGL, what we have done is only creating a **Buffer Object** and filling it with binary data. We have now to tell OpenGL that the data contained inside the buffer object correspond to vertices coordinates and what is the data format.

- In the function `Display`, add the command `glBindBuffer` in order to link the `vertexBufferObject` to the target `GL_ARRAY_BUFFER`.
- It is mandatory to enable the array in order to be able to use it. For this, add the following command. The argument is the index of the considered array:

```c
glEnableVertexAttribArray(0);
```
Finally, add the following line:
   glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
This call to the function glVertexAttribPointer indicates OpenGL that the data format used for the vertices has 4 floats per vertex.
The parameters are the following:
   - the index of the array of vertices,
   - the number of values per vertex,
   - the data format of one value,
   - a boolean indicating if the data has to be normalized,
   - the last two arguments are set to 0 and will be introduced later on.

4. Image rendering.
Now that OpenGL knows what the vertex coordinates are, we can use these coordinates for rendering a triangle.
   - Use the following command for drawing the triangles:
     glDrawArrays (GL_TRIANGLES, 0, 2*3)
The first parameter tells OpenGL that we want to draw from a list of vertices of triangles. The second parameter is the first vertex number and the last parameter is the total number of vertices.
   - Disable the array, then unlink the object vertexBufferObject from the target GL_ARRAY_BUFFER with the following commands:
     glEnableVertexAttribArray(0);
     glBindBuffer(GL_ARRAY_BUFFER, 0);
   - Finally, run the code in order to visualize a white square (default colour).

Although the obtained result is the expected one, some vertices are duplicated by the code. For our elementary case, the impact of this duplication is negligible. However, for more complex cases, this duplication may significantly lower the performances because the array to send to the GPU may become much bigger than necessary.

5. Creating an indexed array.
In order to avoid this unnecessary overhead, we will use two arrays in parallel.
   **Introduction to the method:**
   - The first array will contain the vertex list without duplication.
   - The second array will contain the list of three successive indices making a triangle.
   - In the case of a shared vertex between several triangles, only its index will be duplicated.

   **Application:**
   - Begin to delete duplicated vertices from the array vertexData. Do not forget to change the declaration and initialization of this array.
   - Then, in the function UpdateElementBuffer, allocate an array of GLuint called elementArray with the right size.
   - Delete this array at the end of the function.
   - Initialize elementArray in the function BuildElementArray with the vertex indices of each triangle.
6. Sending data to the GPU and identification.
Once this new data structure set, we will send it to OpenGL.
   o Similarly to the vertexBufferObject, begin by initializing a buffer object called elementBufferObject inside the function InitializeElementBuffer.
   o Afterwards, in the function UpdateElementBuffer and in the same fashion as for vertexBufferObject, allocate storage for the object, using the target GL_ELEMENT_ARRAY_BUFFER.
   o Finish by freeing the allocated storage for the elementBufferObject in the function DeleteElementBuffer.

7. Image rendering.
Now, it only remains to display the square.
   o In the function Display, replace the command `glDrawArrays(GL_TRIANGLES, 0, 2*3)` by the two following:
      `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBufferObject);
      glDrawElements(GL_TRIANGLES, 2*3, GL_UNSIGNED_INT, 0);
   o After the command `glBindBuffer(GL_ARRAY_BUFFER,0)`, unlink the object elementBufferObject from the target GL_ELEMENT_ARRAY_BUFFER in a similar fashion that this command.
   o Eventually, run the code in order to display the same white square as obtained previously.

During this first stage, we have sent a list of vertices to OpenGL. We will now process these vertices inside the graphic pipeline thanks to the use of shaders. Without shaders no transformations of the vertices’ coordinates can be computed (their positions are used as is) and the pixels’ colours of a given object is set to white. In order to address this new stage, we will introduce here below some functionalities for interfacing the shaders with OpenGL.

**Introducing the GLSL language**

In order to directly program on a graphic card, the language GLSL (OpenGL Shading Language) has been specifically developed for OpenGL. The shaders are programs written in this language and executed in the OpenGL rendering process. However, these shaders need to be compiled before to be executed. Henceforth, the OpenGL code includes two compilations: one for the Vertex Shader and the other for the Fragment Shader. Moreover, these two compilations have to be followed by a link edition between these two shaders and the OpenGL program. Similarly to other integrated objects in OpenGL, objects have to be created for containing the shaders. Functionalities are dedicated for loading shaders from external files.

In our code, the function InitializeProgram first loads the shaders with the command LoadShader. This command takes as arguments the shader type (GL_VERTEX_SHADER or GL_FRAGMENT_SHADER) and the file name. LoadShader, in turn, reads the corresponding shader file and calls CreateShader which compiles the shaders and checks if there is no error.
Secondly, a program is created by CreateProgram. A new OpenGL object is then created with an identifier and shaders are then linked to this program.

Load the two available shaders in the archive in order to enable them:
- In the function init, add the following command before the function call
  ```
  InitializeVertexBuffer:
  InitializeProgram();
  ```
- At the end of the del function, add the following command:
  ```
  DeleteProgram();
  ```
- You should see a gray square when running the code.

9. GLSL language syntax.

We will now take a closer look to the syntax of the two shaders provided with the archive.
- Open the files vertex.glsl and frag.glsl located in the folder data.
  These files contain:
  - the GLSL version number,
  - followed by one or more attributes,
  - and by a main function.
  In a shader, several attributes can be defined by the user. These attributes corresponds to inputs and/or outputs.
  - The variable type may be a scalar (int, float...), a vector (vec2, vec3...) and so on.
  - The inputs can be preceded by the keyword in or uniform. The first keyword is used for a data array, each of its entry being processed in parallel by the shader. On the contrary, the second keyword refers to a constant entry for the whole execution of the shader.
  - Output variables defined by the user are identified by the keyword out. Moreover, the GLSL language provides a certain number of “built-in” output variables, all prefixed by gl_. The most used one is gl_Position which allows sending the vertex positions to the next shader.

10. Introduction to the Vertex Shader.

As you may guess, the Vertex Shader takes on input (as attributes) data associated to a vertex.

This shader should output at least one value: the “built-in” variable gl_Position, initialized with the vertex position in the camera space. This output value will then be used by the “rasterizer” for filling the triangles with fragments.

This shader may also compute output variables which will be interpolated between vertices and sent to the Fragment Shader for each computed fragment by the rasterizer (in the previous exercises, the colour gradient in the square has been obtained by interpolation). Furthermore, the Vertex Shader has access to the “uniform” variables which typically contain transformations to apply to the vertices of a given object.
The following scheme gives a sketch about how the Vertex Shader operates:

![Diagram of Vertex Shader](image)

**Figure 5 : Inputs/outputs of the Vertex Shader**

In addition to the prefix “in”, input attributes are prefixed by a code similar to `layout(location = index)`. This code specifies the index associated to the attribute which was defined at the VAO initialization. A common alternative is to resort to a function called `glBindAttribLocation`.

11. Introduction to the Fragment Shader.

After the processing of each vertex, the **Fragment Shader** is called for each fragment processed by the rasterizer. Fragments can be thought as pixels covering the apparent surfaces of the triangles in the scene. The rasterization stage allows, among other things, to interpolate the output variables of the Vertex Shader in order to use them as input variables in the Fragment Shader. The Fragment Shader then has to compute and output colour that will be displayed in the final image. This output colour can be set by declaring and setting an output variable of type `vec4`.

The Fragment Shader has also access to the “uniform” attributes, which are mainly used for textures. We will use them in the next practical course.

The following scheme shows how the Fragment Shader operates:

![Diagram of Fragment Shader](image)

**Figure 6 : Inputs/outputs of the Fragment Shader**

The Fragment Shader given in the archive is very simple: it sets all the fragments it received to a gray colour.

- Change the associated value to the colour set at the Fragment Shader output and check the change by running the code.
The following scheme provides a global view of the graphic pipeline and its processes:

![Diagram of the graphic pipeline](image-url)

Note: in addition to the two kinds of shaders introduced here, there exist two more recent types: geometric shaders (for modifying the mesh) and tessellation shaders (decomposing the mesh in subelements in order to add details to object at a low computational cost, using, for instance, the “displacement mapping” technique – see the Blender Practical course 4 for more details).

The program is now ready for the use of customized shaders. We will now program our shaders in order to add colours to our square.

**Vertex processing**

Firstly, we will use a **Vertex Shader** mimicking the behaviour of a fixed pipeline for defining the vertex positions and colours.

12. Adding the colour attribute

   We represent a colour with 4 floating point values between 0 and 1. The first 3 correspond to the 3 RGB channels while the last one corresponds to the alpha channel. As the storage of contiguous data is more efficient, and in order to avoid to store the vertex attributes in two different storage zones, you will have to add the colour after the position of each vertex in the array `vertexData`.

   o Begin by adding the colour data associated to each vertex.

Then, we have to send these informations to the graphic card via the VBO and state the data format via the VAO. As explained earlier, several storage methods are possible. Here, we choose to use one VBO with interleaved data for the vertex. It is not necessary to make changes to the VBO. However, two VAO are necessary for pointing on the interleaved data: a first VAO of index 0 will point on the position while a second VAO of index 1 will point on the colour.
Take a closer look to the prototype of the initialization function of a VAO:

```c
void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid* pointer);
```

where `stride` corresponds to the size (in bytes) of a whole vertex and `pointer` to the offset in bytes of the first attribute. These two arguments have to be modified.

We use 8 floating points per vertex (4 for the position and 4 for the colour). The value of the parameter `stride` is then `8*sizeof(GLfloat)`. The shift in bytes for getting the first address of the colour attribute in the VBO is `4*sizeof(GLfloat)`. Though, we cannot pass directly this value as argument because the associated type to `pointer` is `const GLvoid*`. A cast of this value is needed. A straightforward solution is to use the command:

```
(const GLvoid*) (4*sizeof(GLfloat))
```

However, we will prefer the use of a function expliciting this cast operation:

```
BUFFER_OFFSET(4*sizeof(GLfloat))
```

Write the needed changes in the function `Display`. Begin by changing the stride parameter of the position VAO, and then define a new VAO of index 1 for the colour.

As the same VBO is used by the two VAOs, one call to the function `glBindBuffer` is required inside the function `Display`.

- Finally, disable the new VAO associated to the colour by calling the function `glDisableVertexAttribArray` before disabling the VAO associated to the position. In this way, the deactivation is always done in the inverse order of the activation.

13. Programming the Vertex Shader

In order to add colour information to a vertex, it is mandatory to change the inputs/outputs of the shader.

- Open the file `vertex.glsl`.
- Declare a new input attribute `color` for the model colour by choosing the right index value.
- Then add an output attribute associated to the colour called `theColor`.
- In the main function, initialize the new output to the value `color`.

14. Programming the Fragment Shader

- Open the file `frag.glsl`.
- Declare a new input attribute for the colour. Be careful to give it the same name as the corresponding output variable of the Vertex Shader.
- Change the initialization of the output attribute `outputColor` by setting it to the value of the input colour.

We can note that for the moment no computations are done in this shader. It only sends back the interpolated colour by the rasterizer.
Modify the Fragment Shader for changing the colour by using mathematical functions available in the GLSL language (cos, sin, exp, abs, ... cf. https://www.opengl.org/sdk/docs/man4/index.php).

For instance, add a halo effect for which the fragment colours tone down as the distance to the origin increases. The computations are done in the Fragment Shader with the formula given here below. However, this calculation needs the position of the fragment obtained by interpolation. This information will be obtained by adding a new output variable in the Vertex Shader (the “built-in” variable gl_Position is not available in the Fragment Shader).

\[
\text{attenuation} = \exp \left( -9 \times \text{distance}^2 \right)
\]

Note: use the function distance(vec4, vec4) for computing the distance from the origin.