TU/E FACULTY OF ELECTRICAL ENGINEERING

COMPUTATION II - 5EIB0

PROGRAMMING:
FINAL TASK

PART 1 OF 2:
BASIC OPENGL GRAPHICS

DO NOT SUBMIT PART 1 TO PEACH

MARCH, 2015

FINAL TASK, PART 1.1: CREATING AN EMPTY OPENGL PROJECT

In this task, you will get the opportunity to work with OpenGL to create your first graphical program!

Make sure you have Visual Studio 2013 Express for Windows Desktop installed. Download the **OpenGLSkeleton.zip** file from OASE. The skeleton is completely setup and should compile and run without errors. If you get errors during compiling or running the program, contact one of the instructors.

Linux users will have to take the source files from the skeleton and add them to their own project. You are required to install GLUT (`apt-get install freeglut3 freeglut3-dev`). Configure your project to link against OpenGL and GLUT, by passing the flags `-lGL` and `-lglut` to your compiler.

FINAL TASK, PART 1.2: EXTENDING THE SKELETON PROGRAM

If you are having trouble understanding what exactly is happening when a certain function is called, you can refer to the following sites:

- Formal specifications of the OpenGL API: http://www.opengl.org/sdk/docs/man/
- Formal specifications of the Glut API: http://www.opengl.org/resources/libraries/glut/spec3/spec3.html
- Lecture notes, containing detailed information on glut and the "why" behind the functions: http://homepages.ius.edu/RWISMAN/b481/html/Syllabus.htm

From this moment on we assume you have loaded the skeleton project in Visual C++. The main function might look a bit different than what you are used to:

```
int main(int argc, char* argv[])
```

This is actually the standard format for a main function, but Visual C++ allows you to deviate from this, using `void main()`. The parameters `argc` and `char* argv[]` represent the input arguments or command line parameters that were passed to your program. `argc` is the number of input arguments, whereas `argv` is a vector (array) of pointers to strings. They have to be given to the `glutInit` function. The return value of main is used to indicate whether your program successfully executed.

The initialization of your application is done in the `init()` function. **Check it out and read the comments.** You will see that OpenGL acts as an empty sheet of paper: different objects can be drawn on this sheet. First the height, width and color of the drawing plane are defined. Then, the color of the sheet and the "pen" color are defined. Note that, while you have a graphical window, it is still possible to print to the console window.

CALLBACK FUNCTIONS

One line in the `init()` function might need some extra explanation: `glutDisplayFunc(display);`

GLUT works with so called callback functions. These functions can be called at arbitrary moments by GLUT. In this case, the "display" function is called each time a new frame has to be drawn to the screen.

The line `glutDisplayFunc(display);` actually gives a pointer to your display function (a function pointer) to GLUT. GLUT now knows which function to call in case a redraw event happens and will do so when needed. There are different callbacks that are each triggered by different events. Below is a description of the ones you will most likely want to use, take a look at the GLUT API for more details and a complete list!

- *Callback register function:* `glutIdleFunc`.
  Registered callback is called when: nothing else has to be done.
  Remarks: It might be tempting to do all your calculations in this callback, but it is better to use a timer callback that gives you some control over the intervals at which the function is called.

- *Callback register function:* `glutMotionFunc`.
  Registered callback is called when: the mouse moves over window while a mouse button is pressed.
  Remarks: Your callback function should have the following form:
  ```
  void your_motion_callback(int x, int y);
  ```

Each time your callback "`your_motion_callback`" is called by Glut, the x and y coordinates of the mouse are passed to this function. All coordinates are relative to the upper left corner of the screen (unlike the pixel drawing coordinates!), with positive x pointing from left to right, and positive y pointing from top to bottom.

- *Callback register function:* `glutPassiveMotionFunc`.
  Registered callback is called when: the mouse moves over window while NO mouse button is pressed.
  Remarks: Can be used to follow the motions of the mouse.

- *Callback register function:* `glutKeyboardFunc`.
  Registered callback is called when: A key is pressed when the GLUT window is active.
  Remarks: The callback should have to following form:
  ```
  void your_keyboard_callback(unsigned char key, int x, int y);
  ```

The x and y coordinate give the mouse location at the time the key was pressed. When the mouse is positioned outside the GLUT window, the position is still reported, relative to the upper left corner of the

window. If you want to know which keys are pressed by the user, you can simply print them to the console.

- *Callback register function:* `glutTimerFunc`.
  Registered callback is called when: a timer expires.
  Remarks: This function is a bit special. See below.

`glutTimerFunc` takes 3 parameters:

```
void glutTimerFunc(unsigned int msecs, void (*func)(int value), value);
```

The first parameter is the amount of milliseconds after which your callback will be called. The second parameter is the function pointer to your callback, that accepts a single integer parameter and returns void (nothing). The third parameter is a value that will be passed to your callback when it is called. The following section of code should exemplify the use of timers.

```
void alarm(int alarmNumber)
{
    if(alarmNumber == 112) {
        cout << "Ring Ring!!! This was alarm 112!" << endl;
        cout << "Next alarm will ring in 100 ms." << endl;

        // Register another timer with a different number that triggers
        // after 100 ms
        glutTimerFunc(100, alarm, 1337);
    } else {
        cout << "Ring Ring!!! Alarm with alarmnumber " << alarmNumber
             << " called!" << endl;
    }
}
```

You can register a set of timers that call the `alarm` function as follows. You can find this in the `init` functions of the skeleton.

```
glutTimerFunc(1000, alarm, 112);
glutTimerFunc(1500, alarm, 1);
glutTimerFunc(2000, alarm, 2);
```

The output will be as follows:

```
Ring Ring!!! This was alarm 112!
Next alarm will ring in 100 ms.
Ring Ring!!! Alarm with alarmnumber 1337 called!
Ring Ring!!! Alarm with alarmnumber 1 called!
Ring Ring!!! Alarm with alarmnumber 2 called!
```

As you can see, multiple timers can registered, all using the same callback function (but you could use separate functions as well). The value parameter can be used to distinguish between the different instances of the timer.

DRAWING PIXELS

Now it's time to look at the display function in the skeleton. All it does at the moment is drawing a pixel. It is pretty bare, but it gives you all the tools needed to draw whatever you want.

```
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);   // clear the backbuffer
    glBegin(GL_POINTS);             // Start a new drawing block

        // Draw points here
        // The point (0,0) corresponds to the lower left corner.

        // The following lines will draw the point (100, 200) in red
        glColor3f(1.0, 0.0, 0.0);   // Set color
        glVertex2f(100, 200);       // Set position and draw

        // The following lines will draw the point (200, 400) in blue
        // Alternative approach to the above:
        Color color = { 0.0f, 0.0f, 1.0f }; // A color (see drawtools.h)
        glColor3fv(color.data());           // Set color from array
        PointF point = { 200, 400 };        // A two dimensional point
        glVertex2fv(point.data());          // Set position from array and draw

    glEnd(); // End of the drawing block

    // Visualize the drawing commands
    glFlush();            // Execute all commands waiting to be executed
    glutSwapBuffers();    // Swap the backbuffer and frontbuffer
}
```

First, we call `glClear` to clear the backbuffer of any pixels previously drawn. This gives us a clean sheet to draw on. We then use `glBegin` to tell OpenGL that we want to start drawing objects. The object type you are about to draw is set as an argument of this function. In this case `GL_POINTS` indicates that we

will draw individual points, or pixels. There are other objects you can draw which might be very helpful for this exercise, so check out the manual topic on this function.

Next we set the pen color and drawing location using the functions `glColor3f` and `glVertex2f` respectively. `glColor3f` takes three values: red, green and blue. `glVertex2f` takes two parameters: an x value and a y value. This is all the information OpenGL needs to draw a pixel: a color and a location.

Alternatively, you can use `glColor3fv` and/or `glVertex2fv`, which take a pointer to an array of respectively three and two floating point values. The code example shows how to use these functions with the `Color` and `PointF` classes, which are provided by `drawtools.h`.

Note that `glVertex2f` and related functions should be called last, as these issue the actual drawing command. You can draw any amount of pixels you want using the above, as long as you stay inside the `glBegin` – `glEnd` block. You may only draw inside such a block. (The only exception to this rule is drawing text.) However, there is nothing that can stop you from opening a new draw block, possibly even with a different argument, to add some more objects to your drawing.

`glFlush()` will execute any pending commands previously issued. A drawing action might be internally buffered by OpenGL, `glFlush()` will make sure all buffered commands are executed.

`glutSwapBuffers()` is an interesting command, because it allows us to create fluently moving objects. Remember the call to `glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB)` from the `init` function? The `GLUT_DOUBLE` flag says we want two drawing buffers (or two sheets of paper): one on which we can draw (the back buffer), and one to show to the user (the front buffer). Whenever we are done drawing, these two buffers are quickly switched. If we were to use just a single buffer, the drawing process would be visible to the user, resulting in an unpleasantly flickering image. Now it is done behind the scenes and only the end result is shown.

Press the "run" button of the skeleton project if you haven't done so already. Now push your nose against the screen and look for a blue and a red pixel. They should be there! Now, there is one last trick to learn: drawing text to the graphics window.

DRAWING TEXT

You will now have to extend the skeleton program. What we are going to build is a debug function; whenever a user presses a key, this key and the mouse location at the time the key was pressed are printed to the screen. For now, we will use a fixed position for the text, say position (300, 300).

1. First, move to the function declarations. We will need 2 more functions: a callback which is called when a key is pressed and a function that is able to print text. You will get the callback declaration for free:
   ```
   void keyfunc(unsigned char key, int x, int y);
   ```

Besides the callback, you also have to declare the function that reads two coordinates and a string. This will be the general purpose text drawing function. Also add a global variable:
`string keytext;`

2. Now, move to the `init` function, where all the callbacks are registered. Register your newly made function by adding following line:
   `glutKeyboardFunc(keyfunc);`

3. Implement the `keyfunc` callback:

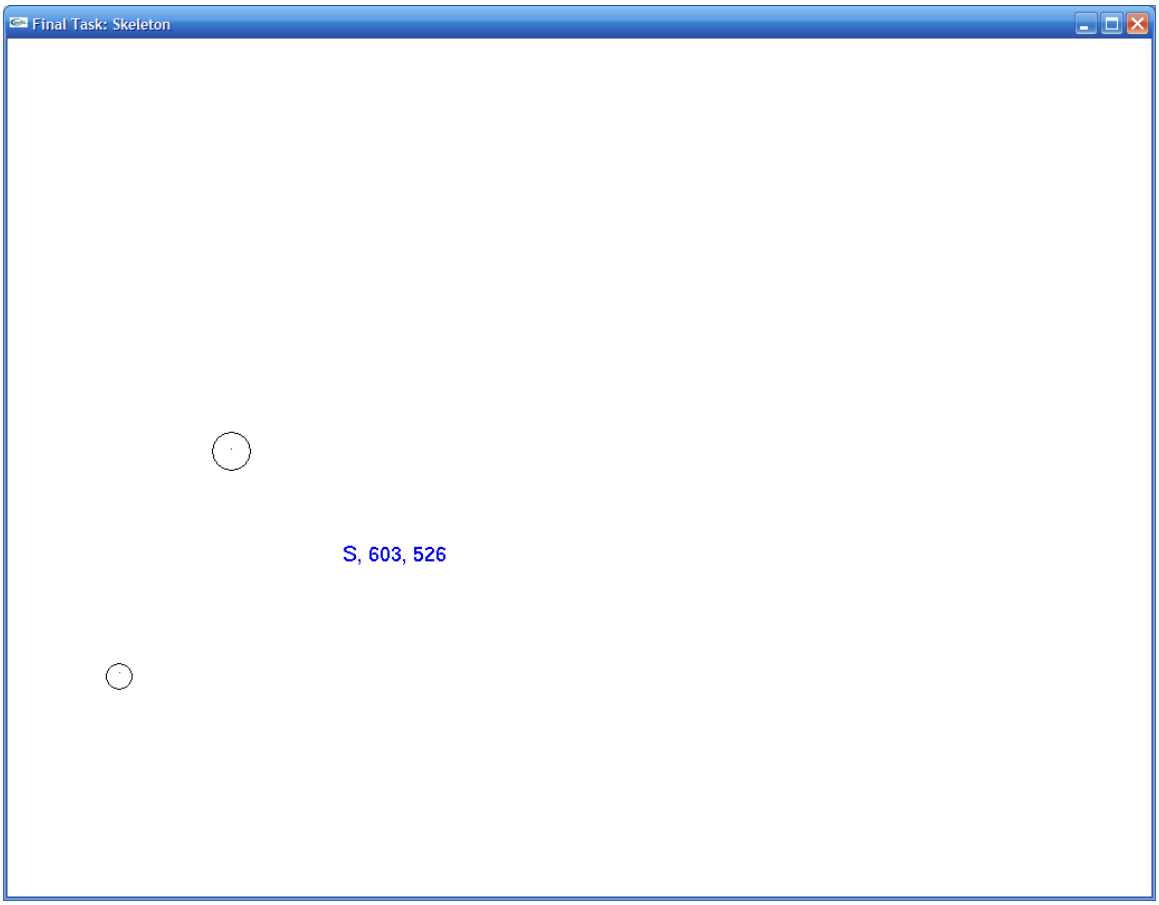
```
void keyfunc(unsigned char key, int x, int y)
{
    char c = key;
    keytext = string{ c } + ", " + to_string(x) + ", " + to_string(y);
    glutPostRedisplay();
}
```

Each time a key is pressed, the variable "`keytext`" is updated with the latest information. `glutPostRedisplay()` forces a redraw of the screen, effectively triggering the `display` callback.

4. Move on to the `display` function. Besides the two pixels, we also want to draw our small sentence. Call your own text drawing function, right below the `glEnd()` command. As arguments, give the text position and the `keytext` variable.

5. Implement your text drawing function. The starting position of the text you are about to draw can be set using `glRasterPos2f(x, y);` Note that you will have to store the position of the cursor somewhere. Individual `char`s can be drawn using the following function call, where `my_char` is the character you want to draw:
   `glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, my_char);`
   So, what you will have to do is:
   a. Call `glRasterPos` and move to the coordinates passed to your function.
   b. Loop through the chars in the string that was passed to the function, calling `glutBitmapCharacter` for each char.

Now let's recap what happens. A user presses a key, which is picked up by GLUT. GLUT notifies the `keyfunc` callback. `keyfunc` prepares a string and notifies GLUT that a redraw has to be done. This will trigger the `display` callback. The `display` callback draws two pixels and then executes the text drawing function. This function draws the characters to the screen. Once the buffers are swapped, the result will be shown on the screen:



In this example, a capital ‘S’ was entered by the user. The mouse cursor was in the position (603, 526) when this happened. Two circles were drawn around the 2 pixels to make them easier to spot.

You might wonder why it is necessary to use a global to pass the text around. The reason is that `keyfunc` cannot draw to the screen directly. If you would call your text drawing function from within this callback, the text would be drawn to the current backbuffer. The problem is that the **backbuffer is cleared** by the `display` function right before it starts drawing!

So you might say, why not move the `glClear` command to right after the buffers are swapped? This indeed would solve the problem in this case, but we need to look a bit further. The `keyfunc` function is only called once, when the key is pressed, but you also want to draw the same text in the upcoming frames, until a new key is pressed. Other callbacks might also call for redraws, but they will not draw the text, because it is not their responsibility. This means you have to store the text somewhere, and redraw it every time using the structure you have just implemented for example.

There has to be a place where you merge all changes into *one* new picture. The most obvious place to do this is the `display` function. You will also have to store all the objects that you want to draw. For this we will use a list.

THE DRAWLIST

You might have noticed that the skeleton consists of some more files than we have covered until now. Check out the "drawtools" and "drawlist" files. The drawlist.cpp and drawlist.h give you an implementation of a `Drawable` class, which can be uses as the basis for all objects that can be drawn to the screen.

In addition, drawlist.h also conveniently provides an alias "`DrawList`" to a `std::list<Drawable*>`. A `std::list<YourType>`, where `YourType` can be any type you want, functions similarly to the doubly-linked circular list that you made for Task 3. The main difference is that it is not circular, heavily optimized and provides much more functionality. Its use is preferred over your own list.

Students that completed Task 3c should recognize that `std::list` is actually a template. You are not required to fiddle with templates for this assignment, and you can ignore it if you want. You can limit yourself to the use of "`DrawList`".

ADDING DRAWABLES TO THE LIST

Let's add a new global variable to main.cpp:

```
DrawList drawList;
```

This list will be used to store all the `Drawables` we want to draw. Each `Drawable` in the list can represent a pixel, text string, line, circle or whatever object you want to draw. In the skeleton, you will find an implementation for a "`Pixel`" and a skeletons for "`Line`" and "`Text`" classes.

Open drawtools.h and look at the definition of the pixel class. As you can see, a pixel inherits from the `Drawable` class, which isn't all that surprising, as we want to insert it into the `DrawList`. Its private data members have all the info you expect for a pixel: a position and a color.

To create a Pixel and add it to a DrawList, add the following code below the glMatrixMode() command at the bottom of the init function:

```
PointF position = { 512, 384 };
Color color = { 0.2f, 1, 0.2f };
Pixel* pixel = new Pixel{ position, color };
drawList.push_back(pixel);
pixel->print();
```

ITERATING OVER THE LIST

Now, move to the `display` callback. Below the call to your text drawing function, add the following:

```
for (Drawable* drawable : drawList) {
    drawable->draw();
}
```

This is a special loop that is called a "range based for loop". The code should be read as "*for each pointer to Drawable in drawList, do the following*." This construction can be used to conveniently iterate over all elements in a container such as a `DrawList`.

Note that you should not modify the structure of the list during such a loop, i.e. you should not add, remove or reorder the elements of the list. If you wish to modify your list during the loop, you should do something like the following:

```
for (auto it = drawList.begin(); it != drawList.end(); /* nothing */) {
    if (/* condition */) {
        it = drawList.erase(it); // Erase returns the next iterator
    } else {
        ++it; // Increment the iterator
    }
}
```

PURE VIRTUAL FUNCTIONS

A `Drawable` has a `virtual` function called `draw`:

```
virtual void draw(void) const = 0;
```

Because this function is `virtual`, the overriding draw function of an inheriting class will be called. In this case, the `draw` function of the `Pixel` class is called, instead of that of the `Drawable` class. This is very useful, because you will want to draw different types of objects from within the same list. They will all have a draw member function, but the functionality of these functions can be totally different.

Note the strange "`= 0`" notation. This means that the virtual function is pure. A pure virtual function has no body or definition, and classes containing such a function cannot be instantiated (you cannot create a `Drawable` directly, but only through inheriting from it). Classes containing pure virtual functions are called **abstract classes**. Any class that inherits from an abstract class is also abstract, unless it overrides the pure virtual function and defines its body. This is exactly what `Pixel` does: it overrides the `draw` function and gives it a body. As such, `Pixel` is not abstract and can be instantiated.

EDIF++

Next, you will have to write a file parser that is able to read EDIF++ files. EDIF++ stands for "Eindhoven Drawing Input Format++" (we could have called it EDIF, but that already existed, hence the ++). It's a file format used to specify a drawing. An EDIF++ file consists of a number of lines, each describing an object that has to be drawn to the screen. In the first part of this Task, this will stay limited to a single command that draws a line. Note: all lines that do not start with a '.' (dot) can be regarded as comment.

CREATE A LINE DRAWING FUNCTION

Drawing a line is surprisingly similar to drawing a point, with just two small changes: instead of a single coordinate pair, you will need two. So for each line, you will have to specify two coordinates. Remember the two pixels we drew? They can easily be converted to a line.

Scroll to the display function and change `GL_POINTS` to `GL_LINES`. This means that in the next draw block, a call to `glVertex2fv` specifies begin or end points of a line. Press "run" and check out the result. Instead of two pixels, you have drawn a line! A nice gradient color effect is added as a bonus.

Lines have a width. The linewidth can be set using the `glLineWidth` function, which takes the width of the line as an argument. Try calling it with different settings, right before the `glBegin` command.

Now pack all the functionality described above into one neat function. This function takes two coordinates as an argument, a color and a width, as shown below. Note that the function takes its arguments by **const reference (&)** to avoid making copies. You do not have to do anything fancy to use references: they are almost like regular objects.

```
void drawLine(const PointF& begin, const PointF& end,
              const Color& color, float lineWidth);
```

Design this function so it can be called from within your drawing routine. **Try it out and integrate its functionality in the** `draw` **function of the** `Line` **class** (you can remove the `drawLine` function afterwards)**.** First, fill the body of the constructor of the `Line` class. This constructor should take the same parameters as the `drawLine` function. Then, add the `print` and `draw` functionality.

You should now be able to create `Line` objects just like you created `Pixel` objects. Experiment with it by creating several lines and adding them to your `drawList`. They should automagically be drawn by the `display` callback.

CREATE A PARSER FOR THE .LINE, .PIXEL, .DRAWING AND .TEXT COMMANDS

Now create an interface to read an EDIF++ file and convert it into an image.

The `.line` command has the following format:

```
.line x1 y1 x2 y2 r g b linewidth
```

This means a line has to be drawn from (x1, y1) to (x2, y2), having color (r, g, b), and a width of linewidth pixels. An EDIF file with the following content will draw a line starting at (0, 0) and ending at (100, 20). The line is red and has a width of 2 pixels:

```
.line 0 0 100 20 1.0 0.0 0.0 2
```

You can read separate lines from a file using `std::getline(filestream, outputstring)` In order to parse the commands. You can then create a `std::stringstream` (from the header `<sstream>`) and pass `outputstring` to its constructor. You can then use the stream to read the contents of the string like so:

```
std::stringstream ss{ outputstring };
ss >> command;
if (command == ".line") {
    ss >> begin[0] >> begin[1] >> end[0] >> end[1]
       >> color[0] >> color[1] >> color[2] >> lineWidth;
}
```

Where `command` is a `std::string`, `begin` and `end` are `PointF`, `color` is a `Color` and `linewidth` is a `float`. You can play around with the variable types if you like, because the coordinates do not necessarily have to be a `float`. However in the end, you will have to call the `glVertex2fv` function on them, which takes floats as input.

Create a nice wrapper function around your parser, something like:

```
void readEDIF(const std::string& filename);
```

It opens a file and reads its contents line by line. For each line read, figure out which command type it specifies. When it is a valid command type, parse the remaining parameters and create an object that corresponds to the command. The commands you have to support are:

Command format	**What it does**
.drawing "title"	Change the window title to "title" (use `glutSetWindowTitle("title")`)
.pixel x y r g b	Draw a pixel of color (r, g, b) at location (x,y)
.text x y r g b "text"	Draw the string "text"
.line x1 y1 x2 y2 r g b linewidth	Draw line from (x1, y1) to (x2, y2) with color (r, g, b) and width linewidth

For both the .drawing and the .text command you will need to remove the quotes from the string you read from the file.

An EDIF++ file called `test.edif` is provided with the skeleton. Add a small section of code in the main function, right before the call to `glutMainLoop` that asks a user for a file name. Call your `readEDIF` function on this file name. The result is shown on the next page.

```
Enter EDIF++ filename: test.edif
Starting GLUT main loop...
```