Graphical Processing Systems

Project – Documentation

1. Contents:

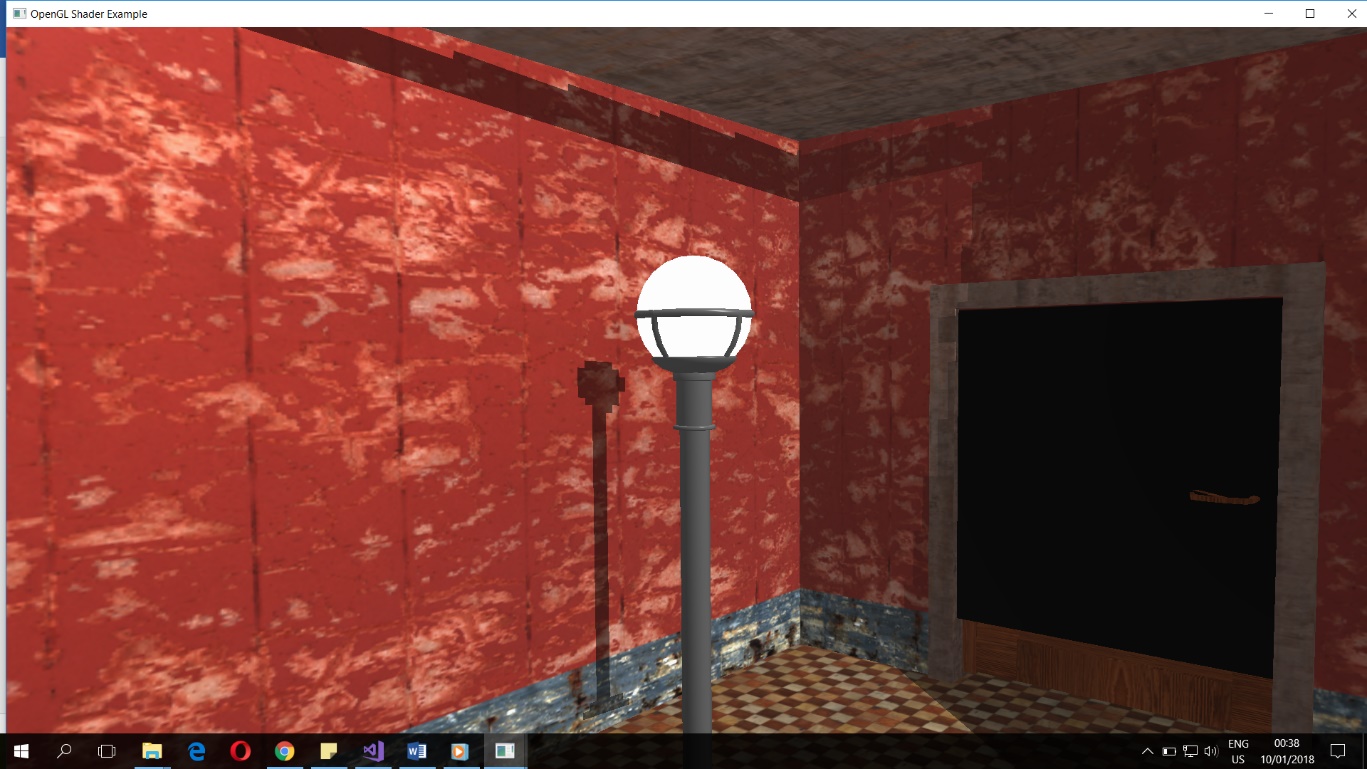
1. Contents  
2. Subject specification  
3. Scenario  
    3.1. scene and objects description  
    3.2. functionalities  
4. Implementation details  
    4.a. functions and special algorithms   
    4.b. graphics model  
    4.c. data structures  
    4.d. class hierarchy  
5. User manual  
6. Conclusions and further developments

1. Subject Specification:

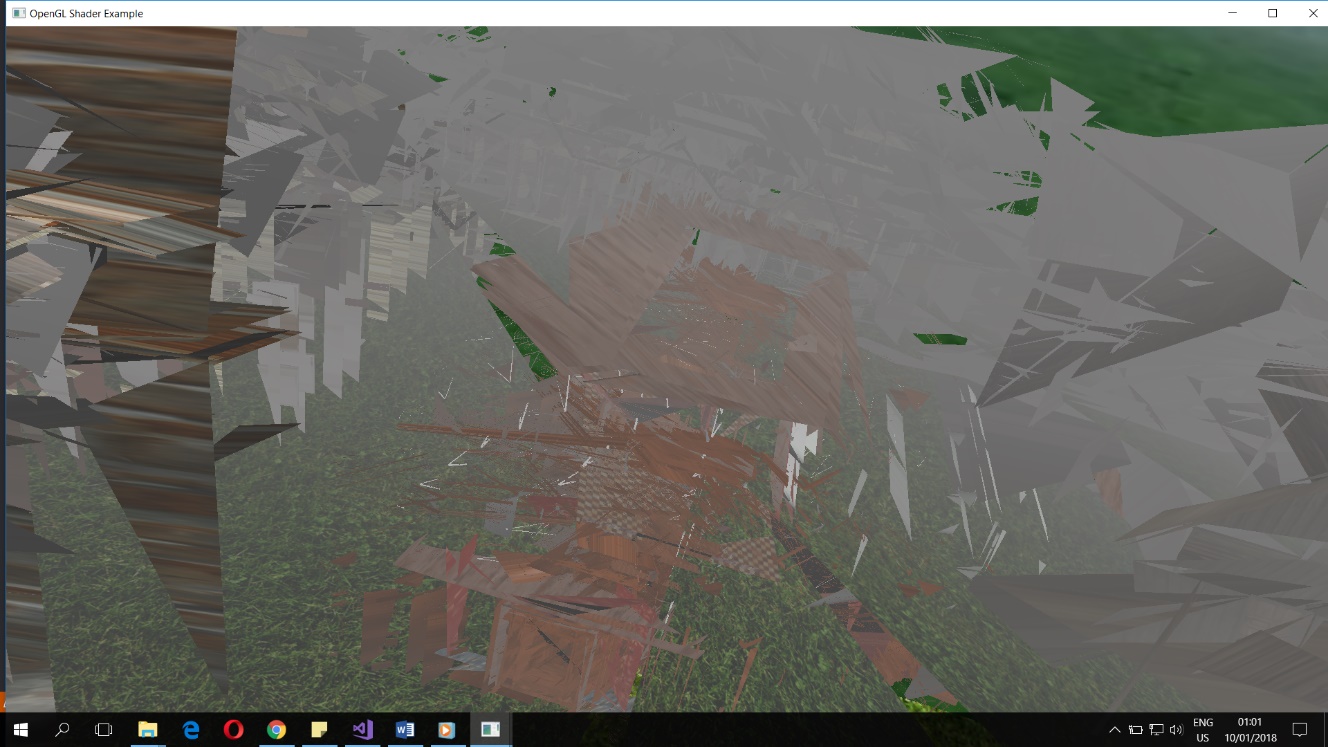
The subject of the project consists in the photorealistic presentation of 3D objects using OpenGL library.[[1]](#footnote-1)

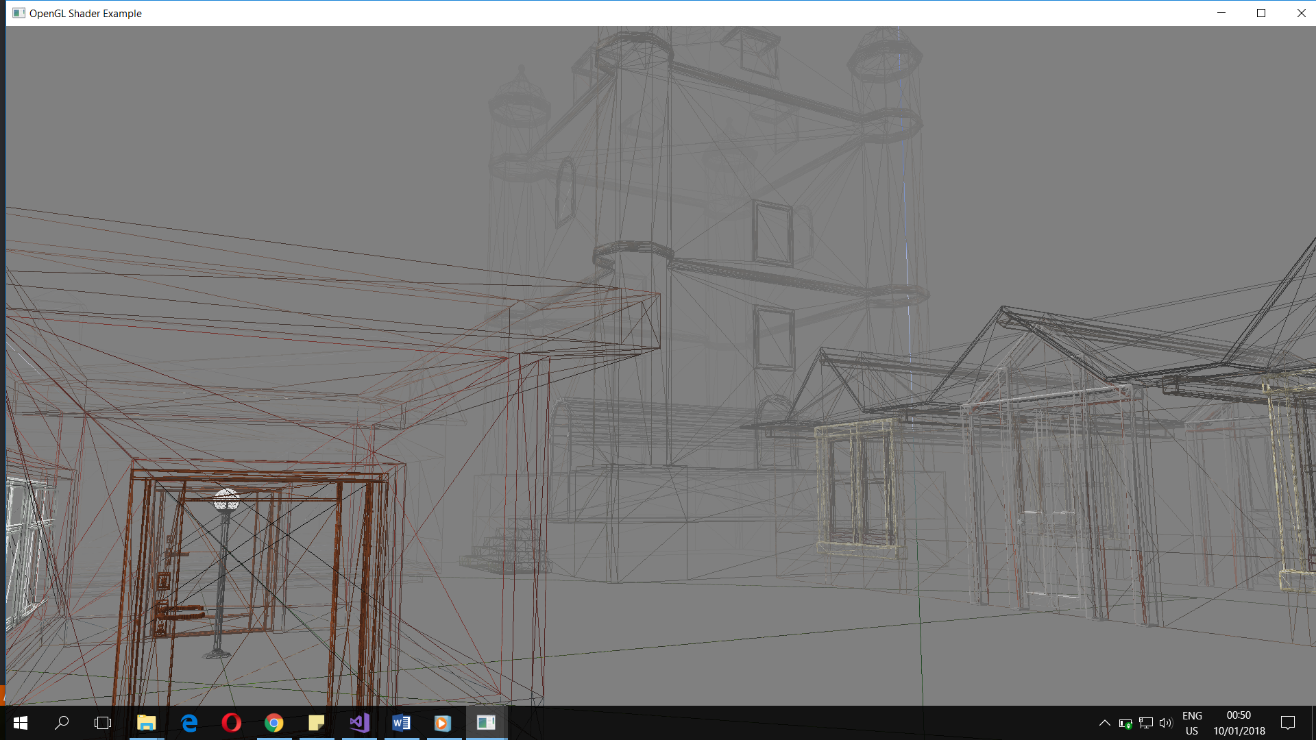
1. Scenario:

My project shows a rural scene in a sunny but foggy day.

The scene consists of various objects, mainly buildings, that are placed on a grassy ground, surrounded by a skybox. The 3D objects are texture mapped .obj files. Some of the objects can be seen in the image above. Here are some other images of objects used:

From the images above, it can be seen the fog present in the scene. The scene also has 2

different light sources: a directional one and a local one (the lamp). Shadows are also computed as it can be seen. Movement inside the scene is possible using mouse and keyboard. Automated movement (flying around the scene) is also possible. The objects can also be viewed as wireframe objects. Object component animation is also present: the doors of the house can be opened and closed. Another functionality of the project is the capability to show the image in inverted colours. The point light inside the house can be switched off to demonstrate the presence of the light. The whole scene can be made to “explode” if the user so desires. The movement inside the scene is limited to the edges of the ground and if the user tries to go beyond that point, they are penalyzed with the appearance of a big red bat that tries to catch the user. If the user is caught, a sound effect is played by the program. To escape from the chase of the red bat, the user has to move close to the lamp inside the house.

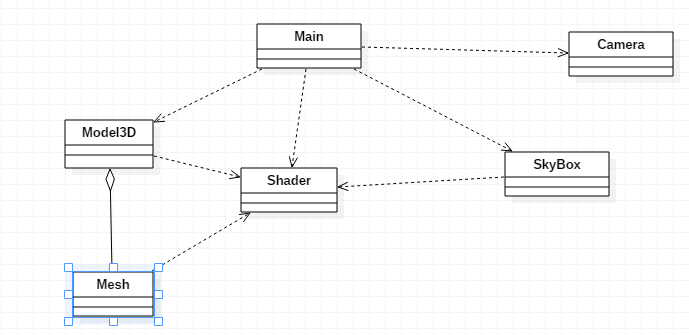
1. Implementation details
   1. Functions and special algorithms

In the main program, the most important functions are arguably initUniforms, processMovement and renderScene. The first function, initUniforms, is used to initialize the uniforms from the various shaders. All the uniforms that don’t change their value should be initialized here. The proper shader program must be selected before initializing the uniforms. The second function, processMovement, is used to process the input from the keyboard. Some functionality can be added to each key through this function. The third and most important function is renderScene. This function is called in a loop and is used to update the scene. It first calls processMovement, then continues by placing the objects in the scene in oreder to create the depth map (for shadow computation reasons). After that the objects are placed in the scene again in order to be drawn, also taking into account the depth map. Finally, the function draws the skybox. Any new object that one wants to place in the scene should be drawn inside this function twice: once for the depth map and again for the actual drawing.

Other interesting algorithms in the program include the mouse movement algorithm, the bat following algorithm, the collision detection and the limitation of movement. For the mouse movement I wanted to create a feeling similar to video games. Each time the mouse is moved, the vertical and horizontal offsets (pitch and yaw, respectively) are computed and sent to the Camera class, specifically the rotate function. The Camera class has three vectors that represent the direction of the coordinate axes which are used for simple movement (up-down, left-right, front-back). When a rotation is applied, one of the 3 vectors remains constant while the other 2 are changed accordingly. The change of the vector representing the up direction (y axis) brings the unexpected but welcome feature of the camera being able to execute rolls by performing repeated pitches and yaws (moving the mouse in a circular motion). While this algorithm is fine for this program, it could be improved with quarternions, which guarantee that no gimbal lock can happen (although none has happened to me using this algorithm either). The bat following algorithm computes two things: the position of the bat and the angle of its rotation. The position is computed by adding to the actual position a fraction of the vector that represents the difference between the camera position and the bat position. The rotation angle is computed as the angle between the original direction of the bat and the vector representing the difference between the camera position and the bat position. The algorithm currently only rotates the bat in the Oxz plane, an improvement would be to also rotate the bat on the Oy axis. Collision detection between the camera and the bat is done in a very basic manner, by considering the distance between the camera position and the bat position. If the distance is too small, a collision is considered to have happened. This algorithm could be improved by employing an actual collision detection algorithm. Limitation of movement is done by checking the camera position coordinates against the hardcoded “walls” and it could be improved by replacing hardcoded values with dynamic ones.

Algorithms used from external sources include the shadow computation algorithm, the explosion algorithm and the summation of lights algorithm. Shadow computation[[2]](#footnote-2) is done as a two-step process: first the scene is rendered from the light’s point of view, creating a depth map texture. Then, this texture is passed to the actual shader which uses it to attenuate the colours of the pixels which are in shadow. The explosion algorithm[[3]](#footnote-3) uses a geometry shader to move each vertex along its normal vector with a distance proportional to the elapsed time. The light summation algorithm[[4]](#footnote-4) is used to compute the final colour of each fragment taking into consideration that there are multiple light sources. It does so by adding to the three values of the light (ambient, diffuse, specular) the values computed for each light source.

* 1. The project, being an OpenGL application, uses shaders for various graphic computations. There are three sets of shader used: the skyboxShader, the simpleDepthMap and the houseShader. Out of these 3, the shaderHouse does most of the heavy lifting.
     1. The skyboxShader is used to display the surrounding skybox. It has 2 stages: vertex shader and fragment shader. The vertex shader is used to set the gl\_Position output parameter and to pass the texture coordinates to the fragment shader. The fragment shader uses the texture coordinates received as input and the samplerCube received as uniform to output the color of the pixels composing the skybox. It also has a uniform called neg which can trigger the display of negative colors.
     2. The simpleDepthMap shader is used to create the depth map needed by the shadow computation algorithm. The vertex shader sets the gl\_Position output parameter as the input position transformed into light space (from the light point of view). The fragment shader simply sets the output colour as some value (it doesn’t matter since the output of this shaders is not displayed).
     3. The shaderHouse is the most complex of the three, having three stages: vertex, geometry and fragment shaders. The vertex shader is used to set the gl\_Position output variable, to transform some of the inputs to other spaces and pass the others to the next shaders. For example, the input vertex position is transformed into the world space and also separately in the light space. The geometry shader is used to perform the explosion algorithm described above (moving each vertex along its normal) and to pass the rest of the inputs to the fragment shader. The fragment shader first computes the shadow value by taking into account the depthMap texture received as uniform. The three components of light (ambient, diffuse and specular) are then computed for each of the light source and summed up. The colour is attenuated with the shadow factor and then it may be negated if the neg parameter is true. Finally, the fog is computed by taking into account the distance between the viewer and the fragment and the fragment colour represents the final output.
  2. The main data structures used in the project are the classes that compose it. Others include vectors and matrices, but these are all handled using the glm library. The classes of this project are:
     1. Camera: is used to represent the view point, direction and target and also modifications of these by means of movement and rotation (it implements the rotation algorithm described above). It also returns the view matrix, using the glm lookAt function.
     2. Mesh: represents a collection of vertices with their associated textures and materials that together form a 3D object or part of one. Its most important functionality is to draw itself.
     3. Model3D: represents a collection of Mesh objects and their textures that together form a complete 3D object. It can read a .obj object from its file, together with the associated textures and materials and create its list of Mesh objects. It can also draw its list of Mesh objects.
     4. Shader: represents an abstraction of the shaders used by the program. It can read, compile and link shaders from a given path and also set its shader program as the current used one. A Shader object can have two or three shader stages (it might not include a geometry shader).
     5. SkyBox: is used to represent the skybox. It can load its textures (“faces”) from files and draw the skybox
  3. Class hierarchy:



1. User manual: after starting the program (double click on OpenGL\_4\_Application\_VS2015.exe) the user has the following options:
   1. Move mouse to look around the camera: up-down for pitch, left-right for yaw and in a circular motion for roll.
   2. Move the camera: A for left, D for right, W for front, S for back, Z for up, X for down
   3. Rotate around the scene: Q for clockwise, E for counter-clockwise
   4. Open and close the door: T to open the door, R to close the door
   5. View the scene as wireframe objects: M, point objects: N switch back to initial view: I
   6. Float around the scene: F, stop floating: G
   7. View negative colours: O, view normal colours again: P
   8. Explode the scene: Y, stop the explosion: U
   9. Turn off the point light: B, turn it on: V
2. Conclusions and further developments

The application is a pretty good example of what an OpenGL application is capable of. While this is only scratching the surface of the possibilities, I believe that my program is interesting enough, although it can be further developed (as mentioned in the algorithms section). The main improvement of this program would be to include better collision detection, and also to fix the objects with problems (texture or otherwise).

1. http://cgis.utcluj.ro/teaching/mod/page/view.php?id=214 [↑](#footnote-ref-1)
2. http://cgis.utcluj.ro/teaching/pluginfile.php/2921/mod\_resource/content/3/Laboratory\_work\_10.pdf [↑](#footnote-ref-2)
3. https://learnopengl.com/#!Advanced-OpenGL/Geometry-Shader [↑](#footnote-ref-3)
4. https://learnopengl.com/#!Lighting/Multiple-lights [↑](#footnote-ref-4)