COMP3811 Computer Graphics - Coursework 2

Al Yaqdhan Al-Maskari (fy19ayyn), Asen Markov (sc20a2m)

In this report, we list the features that were developed for the sample scene for this coursework. Before we begin, we first introduce the main development philosophy that we adopted for this project. Although the coursework task is focused on scene creation (i.e. a one-time task), we believe that good software design must provide clear interfaces and support reusability. For this reason, we aimed higher than simply delivering the "bare" OpenGL code that would render the scene. Instead, we started the development of a small OpenGL wrapper which makes high-level tasks such as "load mesh" or "draw mesh" easier by encapsulating away OpenGL implementation details. We believe that even on the small scale we are working at, such scalability efforts are important. On a larger scale, however, they would be mandatory.

Tasks list:

Band 1:

- **Create a visual scene:** to create a scene, we focused on all bands' tasks to ensure that we covered a good variety of models and features. The scene represents an indoor paintball arena that includes a variety of static and dynamic objects. Each object has an associated material with one or more textures. We acquired the assets from two main sources: either freely available online assets, or custom-built assets in Blender which we assembled ourselves.
- **Complex object constructed in code:** although most of the meshes were created in Blender or acquired through Internet, we chose one of the objects to be created in code instead of using Blender. The object is one of the hiding/cover places that can be found in paintball games. (Can be found in element1.hpp)
- **Implement a perspective projection that adapts to window size when resized:** We handle all window-related tasks in a separate Window class (window.h/.cpp), which takes care of glfw initialization and window set-up. We provide functions for several glfw callbacks in two main categories – input callbacks (key press, mouse movement, etc.) and window callbacks (framebuffer resize, minimization, etc.). On window resize, we read the new sizes of the framebuffer and use them to resize the OpenGL viewport appropriately. This ensures that our application is adaptive, and objects do not get distorted when we shrink the window. Additionally, we implemented fullscreen mode, which can be toggled on/off by pressing the 'F' key.
- **A first-person style 3D camera:** We used arguably the most popular camera navigation controls – WASD for moving forward-backward and left-right. The buttons EQ allow moving up and down. Moving the mouse rotates the camera.

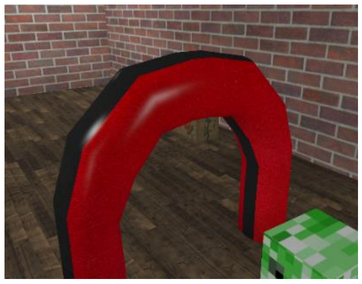
  We implemented the camera by storing its position and current orientation (given in polar coordinates). Transforming the azimuthal/longitudinal angles to cartesian coordinates, we can extract the forward vector of the camera. Then, assuming a constant UP direction (we allow no camera roll), we can perform two cross products to compute the camera right and true up vectors. That way, we can extract the camera view matrix from our camera class on demand. We update the camera position and orientation through input callbacks as already mentioned.

- **Implement and use the vector and matrix classes:** We implement the most common vector and matrix algebraic operations and provide functions for constructing different types of transformation matrices.
- **Implement diffuse and ambient shading originating from a point light:** As mentioned below, we implement the full Blinn-Phong model, which also includes diffuse and ambient shading. We provide code for three different light sources – point lights, spotlights, and directional lights. For the point and spotlights, we use an exponential attenuation function because the ordinary inverse-square law function would never truly reach zero (making the reach of each positional light infinite, which is impractical). Our interface allows setting a maximum illumination radius, beyond which the light is assumed to have zero contribution.

band 2:

- **At least one animated object:** there are several animated objects on the scene. Shooting targets on the back on the floor move in predefined path, other shooting targets on the back wall move in sin and cos functions in x-axis and from left to right in the y-axis. You can pause/resume the targets by pressing enter. A Minecraft diamond sword that flooding above an opened mystery box. Additionally, we created a Minecraft creeper with a hierarchical animation, and it moves in a predefined path. More details in band 4.

Blinn-Phong lighting model: We implement the full Blinn-Phong lighting model (classical version). For each light in the scene, we compute its diffuse and specular contributions to the fragment, modulated by any textures if present. Additionally, we add any emissive contribution and a final ambient colour for the scene. The screenshots below display one object of each type – a brick wall (diffuse), a specular arc object, and a lightbulb (emissive).



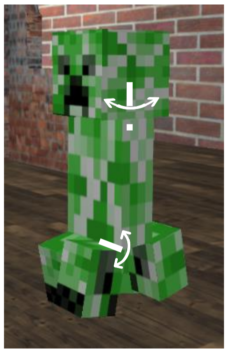
band 3:

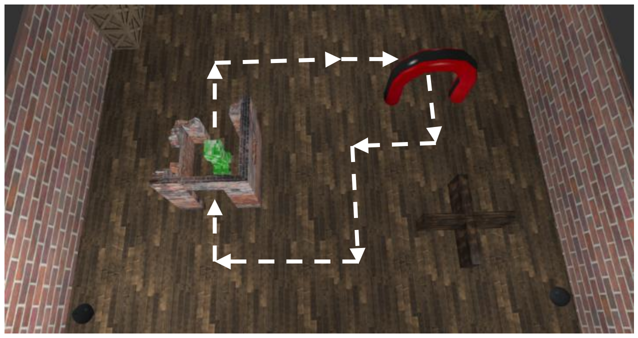
- **Implement texture mapping:** We handle textures in a custom OpenGL wrapper class, which automatically handles texture destruction. We store all created textures in a self-expanding array. Each mesh gets associated with a separate material object, which contains all parameters required, including any associated textures. However, if a loaded mesh has no uv coordinates (or in general, if the material object has a ‘nulllptr’ value for some of its textures), we load a default texture object instead. We have prepared several default textures – a white 1x1 texture (used as a substitute in all places where colour contribution is required), a black 1x1 texture (for metallic textures in the PBR pipeline), and a 1x1 blue texture which acts as a substitute for bump maps (the blue colour corresponds to an unperturbed normal).

- **Use multiple light sources (three or more):** there are five different lights sources in the scene, they can be found in the corners of the arena and the centre. For the scene, we decided on using spot lights only but our framework supports point lights and directional lights as well. For each light we store position/direction (depending on light type), as well as attenuation parameters for point lights and spot lights.
- **Include at least one object loaded from an external Wavefront .obj file (include texture coordinates and normals):** We load .obj files with the use of the rapidobj library. We load and store the files in an indexed format, where we keep vertices and indices in separate arrays. A vertex consists of unique position, normal, and texture coordinates. To identify the unique vertices (and avoid duplicates), we use hash maps during the load. We implement several custom functions to allow us to hash a whole vertex (consisting of several integer parameters – position index, normal index, etc.). We also support per-face materials – we create as many sub meshes as materials there are and generate a separate index array for each. When drawing a mesh, we draw all its sub meshes in sequence, binding the associated material for each one.
- **Include at least one “transparent” (alpha blended) object:** We include a glass window (a very thin cube) with a semi-transparent texture on top. We read the texture's alpha and perform blending with the background.

band 4:

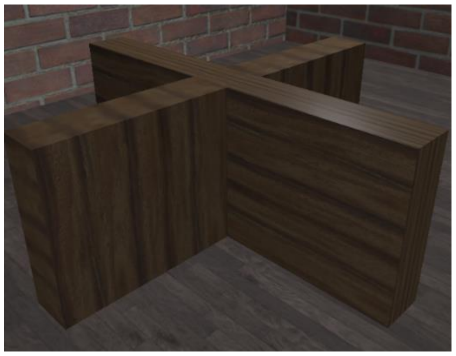
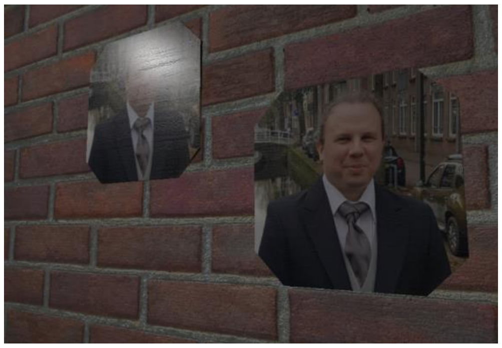
- **Include one or more objects that require hierarchical modeling and transformations:** the four legs, body and head of Minecraft creeper has hierarchical modeling and transformations. The legs and the head have their own animation and they have been combined with the body. So, any transformation that applied to the body will be applied to the legs and the head as well.
- **an object that moves along a complex predefined path:** the creeper above moves in predefined path; it moves in the arena and through some objects like the red arch. The below picture shows the predefined path.





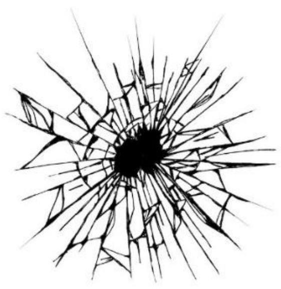
- **Use multi-texturing, where an additional texture controls the emissive color or specular exponent of a material:** We allow a lot of custom control through texturing. Apart from diffuse textures, we also use specular maps to control the specular reflectivity and emission maps for emissive materials (e.g. the lightbulb). When implementing the Physically-Based pipeline, we provided additional support for a variety of PBR textures such as roughness, metallic texture, etc.
- **A custom (not Blinn-Phong) shading model and implement it:** We implemented a physically based rendering pipeline, based on the Cook-Torrance shading model, which uses microfacets to

simulate light in a more believable way. Contrary to the Blinn-Phong model, here we use a BRDF to determine the amount of light reflected in the view direction. The model also handles conservation of energy correctly. The light contribution is split between diffuse and specular components based on some material parameters. For the specular part, we compute three terms – a Fresnel factor (determines the amount of light reflect vs absorbed), a microfacet distribution function, and a geometrical attenuation factor. Screenshot of the more realistic specular reflection can be seen below.

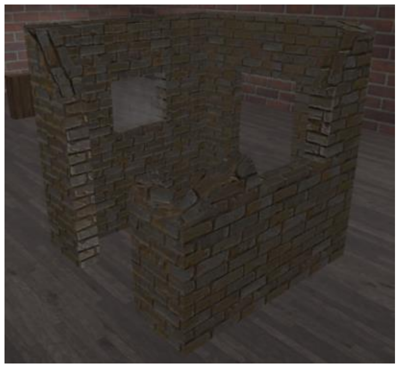
The PBR pipeline requires the use of some PBR materials, mostly texture based. For each object with such material, we load an albedo map (i.e. diffuse map), metallic map (determines whether the fragment is part of a metallic or dielectric material), a roughness map (one of the main parameters for the PBR pipeline), and an ambient map.

- **Integrate a third-party UI library:** Through the ImGui library, integrated on the software, the game allows users to choose the level of complexity where they can change the targets' speed by choosing easy, medium, or hard levels. Also, we have added a button to make the creeper jump. Moreover, for user convenience, the UI lists all key bindings - the buttons for controlling the camera, taking a screen shot, or pausing the targets.
- **Use a material with alpha masking:** In our Cook Torrance shader, we also implemented alpha masking functionality. Contrary to alpha blending, here we use a different texture's colours to determine which fragments should be visible and which not. We assume a black-and-white texture mask, where white texture stands for visible fragments, and black texture show that fragments should be transparent. In our implementation, alpha masking is independent of alpha blending. To demonstrate this, we add an alpha-masked bullet hole to our window glass. This object contains both alpha blending (transparent glass) and alpha masking (bullet hole).
-



An alpha masked bullet hole, superimposed on an alpha-blended glass window. The mask image is shown on the right.

- **Implement a function to take screen shots on request:** We use the glReadPixels functions to read data from the current framebuffer. We then write the data in a .ppm file (which allows us to write the colour data directly). We begin writing from the top row and go down.
- **Bump mapping (advanced feature):** To make full use of the PBR materials available, we implemented bump mapping. During mesh loading, we compute tangent and bitangent vectors to construct the tangent coordinate space. We traverse all mesh faces and compute uv differences between its vertices. We plug these differences in a linear system, which, if solvable, gives us the tangent vector. We distribute the tangent to the three vertices of the face. At the end, we iterate all vertices, normalizing their accumulated tangents. Passing the tangents to the vertex shader (together with a handedness parameter) allows us to compute the TBN matrix (which converts from tangent to world space). Although it would be more efficient transform the camera and light vectors to tangent space in the vertex shader, this goes against our forward rendering approach where the fragment shader may have a variable number of lights set. To simplify things, we simply pass the TBN matrix to the fragment shader, where we transform the TBN normal (fetched from the bump map) to world space.



- **Two or more different shader program:**
  Finally, we implemented two different versions of our rendering pipeline – one with bump mapping and one without. We have configured our application to automatically determine if a mesh has an associated bump map. If this is the case, the bump map shaders are loaded. If such a texture is missing, the ordinary Cook-Torrance shaders are used instead.

Appendix:

	Work split	
Task	Al Yaqdhan	Asen
Create sketch of the scene in Blunder (custom-built assets and downloaded assets)	100%	
Create the scene in code (place the objects, translate, scale, or rotate them)	90%	10%
Construct an object in code	100%	

Implement a perspective projection that adapts to window size when resized		100%
A first-person style 3D camera		100%
Implement and use the vector and matrix classes	100%	
Implement diffuse and ambient shading originating from a point light		100%
Animated objects	100%	
Blinn-Phong lighting model		100%
Implement texture mapping		100%
Light sources (and code)		100%
Object loader code		100%
Transparent object		100%
Hierarchical modeling and transformations	100%	
An object moves along a complex predefined path	100%	
Use multi-texturing		100%
Integrate a third party UI library	100%	
Use a material with alpha masking (code)		100%
Function to take screen shots		100%
Bump mapping		100%

Elements we have implemented:

- The arena (walls roof and floor)
- The pictures below (see on the scene the scene)