Graphics Processing Project: City

Student: Tudor Giuroiu

January 18, 2024

Contents

1 Subject Specification

The Graphics Processing Project represents a simple way to display the capabilities of OpenGL and GLSL for the creation of a dynamic and visually captivating 3D scene. In pursuit of this objective, the implementation will traverse a diverse array of graphics techniques, using fundamental aspects such as shading, lighting intricacies, the intricacies of shadow mapping, texture mappings, object modelling, matrix computations, and the crucial element of user interaction. By integrating these techniques, the ultimate goal is to build a compelling and realistic virtual environment.

This serves as an application and synthesis of the theoretical foundations learned from laboratory studies, where the basics of graphics programming were presented. As such, the project stands not just as an exercise in application but as a demonstration of the practical application of the learned concepts, demonstrating the ability to translate theoretical knowledge into a tangible 3D visual experience.

2 Scenario

2.1 Scene and Objects description

The depicted scene presents the focal point of a city's downtown area during the noon hours, characterized by discreet and exotic activities transpiring behind closed doors.

Situated upon a textured brick pavement resembling real-life pedestrian areas, the scene serves as the foundation for the environment. The central locale is surrounded by six distinct buildings, each varying in shape, size, and texture, some comprising multiple objects.

The interiors of all buildings remain unoccupied, with the exception of one structure featuring a door frame and a door that grants access to an enclosed room. This room comprises three distinct objects: a patterned floor, textured white-painted walls, and a simple plane serving as the ceiling.

Functioning as an exotic dance club, the room houses characteristic elements such as a miniature stage for performances, accompanied by two iconic pole objects suspended above. Additional elements include couches, each composed of two separate objects: the couch itself and legs textured with a different material. A strategically placed nightstand facilitates visitors in securely leaving their belongings while enjoying the vibrant atmosphere, contributing to an overall comfortable ambiance.

Central to the scene's atmosphere is the meticulously modeled dancer, crafted in Blender using a foundational T-pose human model. The intricate process involved the incorporation of a skeletal structure, its binding to the model, and morphing the figure into a suitable and dynamic position, resulting in the captivating main attraction.

2.2 Functionalities

The scene incorporates numerous functionalities that contribute to a heightened level of realism and entertainment. One notable feature involves the manipulation of light direction, affording users the ability to dynamically rotate and translate the light source based on keyboard inputs. Complementing this, the camera animation initiates automatically upon the player's entry into the scene, concluding upon user interaction. An additional facet involves the capacity to adjust fog density, providing users with the option to enhance or diminish atmospheric effects.

An integral functionality is the diverse rendering modes for objects and the overall scene, encompassing solid, wireframe, polygonal and smooth surface representations. Of paramount importance is the interactive capability allowing the opening and closing of a door, a process exclusively accessible when the player is in close proximity. Furthermore, an internally positioned red light source may be activated or deactivated at the user's discretion.

A convenience-oriented functionality permits users to assume either a first-person perspective or adopt a spectator role, with seamless toggling between these modes. Additionally, users can opt to display the depth map, facilitating a comprehensive understanding of object placements within the depth buffer. Notably, the camera's movement is facilitated through both mouse and keyboard inputs, ensuring a versatile and user-friendly interaction experience.

3 Implementation Details

The code uses the OpenGL graphics library for rendering 3D graphics and GLFW for handling window creation and user input.

GLEW (OpenGL Extension Wrangler) is used with the `#define GLEW_STATIC` directive to manage OpenGL extensions statically.

GLM (OpenGL Mathematics) is utilized for handling mathematical operations related to graphics, including matrix transformations, vectors, and other mathematical functionalities.

The application uses shader programs to manage the rendering pipeline, including vertex and fragment shaders. Several shaders are employed for different rendering purposes, such as basic rendering, depth map generation, screen quad rendering, and skybox rendering.

The code implements a camera system allowing the user to navigate the 3D scene. The camera supports both first-person and spectator modes.

Various 3D models are loaded into the scene, representing buildings, ground, furniture, and other elements. These models are rendered using the provided shader programs.

A skybox is incorporated into the scene, providing a background and enhancing the overall visual experience.

The application features dynamic lighting, including a directional light source that can be adjusted by the user. Additionally, a point light source

is positioned in the scene.

The code implements shadow mapping to achieve realistic shadows in the rendered scene. A depth map is generated to simulate the interaction of light with the objects in the scene.

The code includes animations for a rotating dancer model and a door opening and closing. A camera animation sequence is also implemented to guide the user through the scene.

The code incorporates error-checking mechanisms to detect and log OpenGL errors, aiding in debugging.

3.1 Functions and Special Algorithms

The code comprises several functions and special algorithms to achieve its functionality. Key components include:

Functions such as `'renderObjects'`, `'renderWoman'`, `'renderDoor'`, `'renderLightCube'` handle the rendering of different elements in the scene using specific shader programs.

Functions like `'initOpenGLWindow'`, `'initOpenGLState'`, `'initModels'`, `'initShaders'`, `'initSkybox'`, `'initUniforms'`, and `'initFBO'` initialize various components, including the window, OpenGL state, 3D models, shaders, skybox, uniforms, and frame buffer objects.

Functions like `'keyboardCallback'` and `'mouseCallback'` manage user input through GLFW callbacks, facilitating keyboard and mouse interaction.

The `'cameraAnimation'` function orchestrates a predefined camera movement sequence to guide the user through the scene.

The `'renderSceneToDepthBuffer'` function is responsible for rendering the scene from the light's perspective to generate a depth map for shadow mapping.

he `'computeLightSpaceTrMatrix'` function calculates the light space transformation matrix used in shadow mapping.

The `'glCheckError'` function is a utility for detecting and logging OpenGL errors during runtime.

One special algorithm I used for computing the shadows is Shadow Mapping. The algorithm is the following: render the scene from the light's point of view. It does not matter how the scene looks like (color information); the only relevant information at this point are the depth values. These values are stored in a shadow map (or depth map) and can be obtained by creating a depth texture, attaching it to a framebuffer object and rendering the entire scene (as viewed by the light) into this object. This way, the depth texture is directly filled with the relevant depth values.

Next render the scene from the final point of view (the camera's position) and compare the depth of each visible fragment (projected into the light's reference frame) with depth values in the shadow map. Fragments that have a depth greater than what was previously stored in the depth map are not directly visible from the light's point of view and are, thus, in shadow.

The algorithm designed to handle mouse input for controlling the orientation of a virtual camera within a 3D scene is the following: the main function, which serves as a callback for mouse movement events, begins by checking if it is the first time the function is called. If so, it initializes variables to store the last known mouse position and updates the firstMouse flag.

The algorithm calculates the change in mouse position (xoffset and yoffset) since the last frame. These offsets are then scaled by a sensitivity factor to control the responsiveness of the camera to mouse movements. The yaw and pitch angles of the camera are updated based on the calculated offsets.

To prevent unrealistic camera orientations, the algorithm applies bounds to the pitch angle, ensuring it stays within the range of [-89.0f, 89.0f]. The camera's orientation is then used to update the view matrix, likely influencing the rendering of the 3D scene.

The implemented algorithm for generating the skybox utilizes a technique commonly known as cube mapping. In this process, six textures, each representing a different face of a cube, are mapped onto the inner surface of a cube surrounding the scene. These textures are typically captured from different viewpoints, such as the six directions (up, down, left, right, front, back) from the center of the environment. The cube map is then applied to the skybox geometry, creating the illusion of an encompassing background. This approach allows for the simulation of distant scenery or atmospheric effects. The skybox algorithm often involves shaders to handle the mapping and rendering of the cube map onto the skybox geometry.

3.1.1 Possible Solutions

The Shadow Mapping technique in the code uses a depth map framebuffer object (FBO) with an associated depth map texture. Initialization is handled in the `'initFBO'` function, creating the FBO and attaching the depth texture. Rendering the scene to the depth buffer is done in the `'renderSceneToDepthBuffer'` function, utilizing the `depthMapShader` to update the depth map texture from the light's viewpoint.

The `'computeLightSpaceTrMatrix'` function calculates the light space transformation matrix, crucial for transforming fragment coordinates into the light's view space. In the main rendering loop, this matrix is passed to shaders. In the `'renderScene'` function, the shadow map texture is bound, and the main shader (`myBasicShader`) uses it to determine shadow presence. The fragment shader compares light perspective depth with the shadow map depth, adjusting fragment color accordingly. This approach effectively simulates realistic shadows in the rendered scene based on the light source's perspective.

The camera movement algorithm was implemented in the following way: at the beginning, some variables are initialized. `firstMouse` is a boolean flag used to determine if this is the first time the mouse callback function is called. `yaw` and `pitch` represent the horizontal and vertical angles of the camera, with yaw

initially set to -90 degrees. textttlastX and textttlastY store the last known position of the mouse and are initialized to default values (400, 300).

The `'mouseCallback'` function is likely registered as a callback to handle mouse movement events. It checks whether it's the first mouse movement and, if true, initializes the `lastX` and `lastY` variables with the current mouse position, setting `firstMouse` to false. Subsequently, it calculates the offset of the mouse movement (`xoffset` and`yoffset`) since the last frame.

A sensitivity factor is applied to these offsets to control how much the camera orientation changes in response to mouse movement. The yaw and pitch angles are then updated based on the calculated offsets. Additionally, there are bounds checks to ensure that the pitch angle stays within the range [-89.0f, 89.0f].

The skybox generation employs a cube map technique, loading textures specified in the faces vector onto the `mySkyBox` object using the Load function. The cube map is rendered within the main loop using the `skyboxShader` on the `mySkyBox` object, creating a captivating background with depth and atmosphere. The shaders handle the mapping of the cube map onto the skybox geometry, seamlessly integrating it into the 3D scene.

3.1.2 The Motivation of the Chosen Approach

The motivation for using the Shadow Mapping Algorithm lies in the ability to enhance the visual realism of rendered scenes in computer graphics. Shadow mapping enables the simulation of realistic lighting effects by accurately representing the interaction of light sources with objects in a 3D environment. Here are some key motivations for using shadow mapping:

The motivation for implementing the skybox generation algorithm is to enhance visual realism and immersion in 3D environments. It provides a dynamic backdrop, improving the overall atmosphere and user experience in applications like games or simulations.

3.2 Graphics Model

The overall graphics model is composed of the object models and the shaders that compute the colors.

Blender was used to create and design the 3D objects. This involved shaping, texturing, and defining the geometry and materials of the models. Then textures were applied to the 3D models. I ensured that the textures were UV-mapped correctly to your geometry. Then the objects were exported from Blender as `.obj`. When exporting the .obj file, Blender will typically create a corresponding `.mtl` (Material Template Library) file. This file contains information about the materials and textures applied to the objects. Then I placed the `.obj`, `.mtl` and textures files in one folder. The models are loaded into OpenGL application using `LoadModel` method of the `gps::Model3D` class.

3.3 Data Structure

Matrices (`glm::mat4`): Matrices play a crucial role in graphics programming, representing transformations such as model, view, and projection matrices. The GLM library provides the `glm::mat4` type for these transformations, allowing efficient manipulation and application of transformations to vertices and objects in the 3D space.

Vectors (`glm::vec3`): Vectors in GLM, specifically `glm::vec3`, are employed for storing and manipulating 3D spatial coordinates. They are used to represent positions, directions, colors, and other attributes in the scene. The code leverages vectors extensively for handling camera positions, light directions, and various object positions.

GLFW Data Structures: GLFW is a library for creating windows with OpenGL contexts and managing input. The code uses GLFW data structures, such as `GLFWwindow`, to handle the application window, `GLFWcallbacks` for event handling, and GLFW's input processing structures like `GLFW_KEY_XXX` for keyboard keys.

OpenGL Framebuffer Objects (FBO): The code utilizes OpenGL's Framebuffer Objects (FBO) to implement off-screen rendering for shadow mapping. FBOs involve multiple OpenGL data structures, including GLuint variables representing framebuffer and texture objects.

Shader Uniform Locations (`GLint`): Uniform locations are stored using GLint data types. These locations are used to efficiently update and retrieve values from shaders. The code maintains uniform locations for model, view, projection matrices, light parameters, fog density, and other shader variables.

Boolean Flags and Variables: The code uses boolean variables (e.g., `showDepthMap`, `animationOn`, `headUp`) to control various aspects of the application, such as toggling the display of a depth map or controlling camera animations.

Arrays (`GLboolean pressedKeys[1024]`): An array of `GLboolean` values is used to keep track of the state of keyboard keys. This array enables handling multiple key presses and releases effectively.

Custom Data Structures (`gps::Camera`, `gps::Model3D`, `gps::Shader`, `gps::SkyBox`): The code defines several custom data structures such as `Camera`, `Model3D`, `Shader`, and `SkyBox` using classes. These structures encapsulate functionality and data related to camera movements, 3D models, shaders, and skybox rendering.

3.4 Class Hierarchy

The code features a well-organized class hierarchy for a 3D graphics application:

Shader (`gps::Shader`): Manages shader operations, including compilation, linking, and uniform variable setup.

Camera (`gps::Camera`): Handles camera position, orientation, and perspective within the scene.

Model3D (`gps::Model3D`): Represents 3D models, managing data loading, transformations, and rendering using OpenGL.

SkyBox (`gps::SkyBox`): Renders the skybox by loading textures, setting up shaders, and rendering geometry.

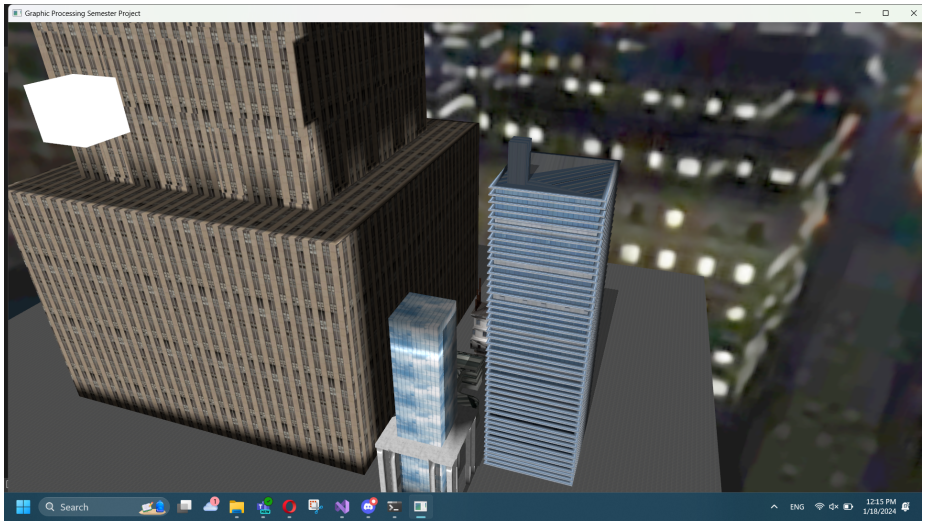
Lighting (`gps::Lighting`): Potentially manages lighting effects, handling light sources, illumination, and shadow maps.

Application (`App`): Top-level class orchestrating the entire graphics application, handling initialization, window creation, user input, scene updates, and rendering coordination.

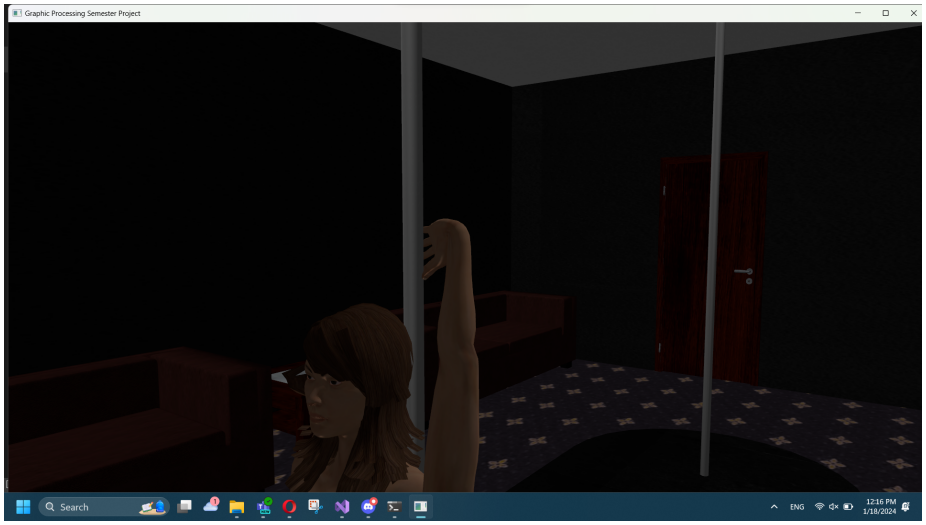
4 User Manual / Graphical User Interface Presentation

W Key moves the player forwards.
S Key moves the player backwards.
A Key moves the player to the left.
D Key moves the player to the right.
SPACE Key stops the beginning animation.
Q Key rotates the camera to the left.
E Key rotates the camera to the right.
F Key increases the fog.
G Key decreases the fog.
L Key toggles the red light inside the room.
T Key toggles the door opening/closing (only if the player is in close proximity).
M Key toggles the depth map view.
1 Key sets the object rendering to polygonal and smooth surfaces.
2 Key sets the object rendering to vertex view.
3 Key sets the object rendering to wireframe objects.
J Key rotates the light direction to the left.
K Key rotates the light direction to the right.
B Key moves the light direction downward.
V Key moves the light direction upward.
C Key toggles the spectator mode.


Graphic Processing Semester Project
Search
ENG
12:15 PM
1/18/2024

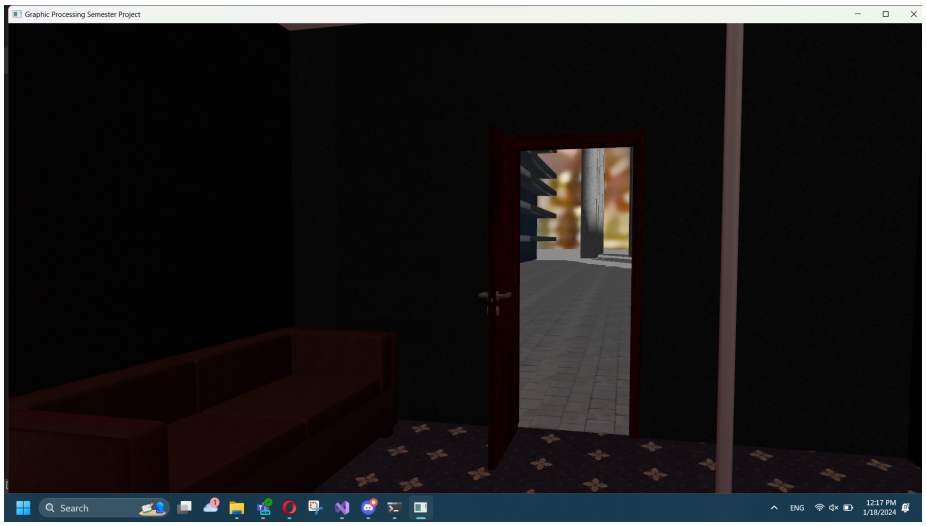

Graphic Processing Semester Project
Search
ENG
12:15 PM
1/18/2024

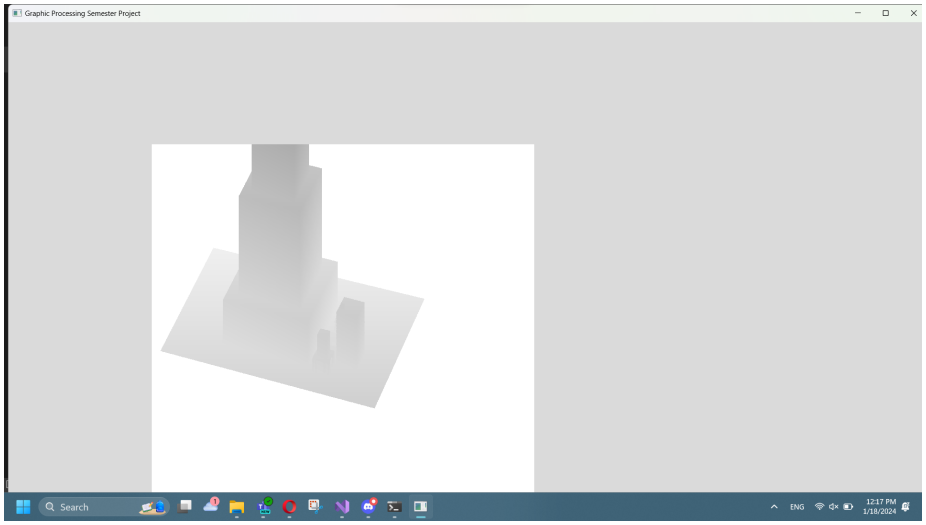

Graphic Processing Semester Project
Search
ENG
12:16 PM
1/18/2024


Graphic Processing Semester Project
Search
ENG
12:16 PM
1/18/2024


Graphic Processing Semester Project
Search
ENG
12:17 PM
1/18/2024


Graphic Processing Semester Project
Search
ENG
12:17 PM
1/18/2024


Graphic Processing Semester Project
Search
ENG
12:17 PM
1/18/2024


Graphic Processing Semester Project
Search
ENG
12:17 PM
1/18/2024

5 Conclusions and Further Developments

In conclusion, the Graphics Processing Project has successfully demonstrated the utilization of OpenGL and GLSL to create a captivating 3D scene, showcasing a lot of graphics techniques. By integrating shading, lighting intricacies, shadow mapping, texture mappings, object modeling, matrix computations, and user interaction, the project has achieved its ultimate goal of constructing a compelling and realistic virtual environment.

This project not only serves as an application of theoretical foundations acquired through laboratory studies but also stands as a testament to the practical application of these learned concepts. The ability to translate theoretical knowledge into a tangible 3D visual experience is exemplified through the intricately designed downtown scene, portraying discreet and exotic activities within the city.

Some further developments could include:

Exploring opportunities for shader optimization to enhance rendering performance. This may involve investigating more advanced shading techniques or implementing shader programs tailored to specific rendering scenarios.

Incorporating more sophisticated lighting models can be done, such as physically-based rendering (PBR), to achieve more realistic and visually appealing results. Also a spot light can be added.

Extending the application by implementing user interaction and interface components, allowing users to manipulate and interact with the 3D scene dynamically.

Improving texture mapping to add richness and detail to rendered objects.

Enhancing documentation to provide comprehensive insights into the codebase, making it easier for developers to understand and contribute. Implement

thorough testing procedures to ensure the stability and reliability of the application.

Adding a task or a goal to the scene.

6 References

Stack Overflow

Tutoriale Blender - Cosmin Nandra

Moodle Graphics Processing Labs