

COMP3011 Computer Graphics

Assessment 3

Report Sheet (v6)

Use this table to help you prepare for your demo. **Submit this report to Moodle.**

Student Name: Cheng Qin

Student ID: 20216424

Username: scycq2

I agree for my code & report to be published, with my name, to future students as an example (yes/no): yes

Introduction		
<i>Please explain why you implemented this scene</i>	<i>Describe your inspirations</i>	<i>Provide a general description of the scene.</i>
Aspiring to incorporate and showcase the entirety of the knowledge and skills I acquired in the labs, I envisioned a scene where each element is testament to my learned mastery. And to keep up with Sample1, I added some extra content including sky boxes, terrain with height, light reflection, etc.	My inspiration came from a well-loved cinematic masterpiece, 'Up'. This whimsical movie led me to imagine a house that defied gravity, buoyantly floating in the air. I used this as the cornerstone of my concept, and gradually added complexity based on it, such as the landmark 'Q' (also my last name), moving cars, controlled lights, and so on.	I have manually created an object resembling the alphabet 'Q' and a skybox. Additionally, I imported an OBJ file representing a house and procedurally generated a car with its accompanying headlights. Furthermore, I procedurally created two floor lights to illuminate the house, where the headlights utilize spot light, and the floor lights employ positional light. The headlights and floor lights can be individually toggled on or off, whereas the environmental light, represented by a directional light, remains constant and cannot be turned off. The car can be controlled to move along a path generated by a Bezier Curve featuring nine control points. Moreover, I have implemented two cameras within the scene. The first is a fly-through camera, allowing for arbitrary observation of the scene. The second camera, the model-viewer camera, is specifically designed to focus on the car. It is possible to switch between these two cameras during the scene. All objects are positioned on a procedurally generated terrain, created using Perlin noise to establish varying heights for a randomized appearance. And encapsulated into a skybox. To enhance visual quality, I have implemented shadow, texture, and anti-aliasing techniques with the provided lab code. Furthermore, I have upgraded the lighting system by incorporating light reflection and light blending capabilities.

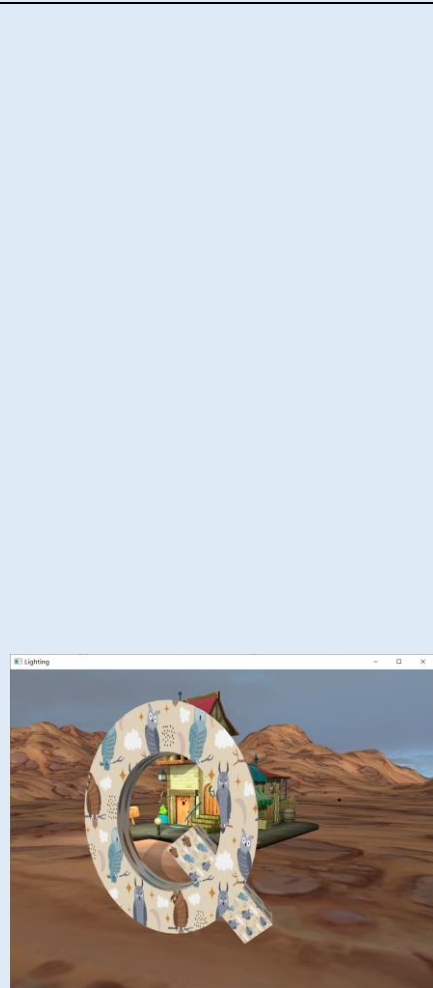
TR 2 – 3D Modelling

Object 1 - modelled by hand

Please give a screenshot

reference specific code (filename and line)

Description of object



node.h: line 8-10
mesh.cpp:
line 315-344: create function
material.cpp:
line 3-10: default reflection parameters

q.cpp:
line 14-872: uses manually created vertices array to generate the shape of 'Q', which is the first alphabet of my surname.

skybox.cpp: line 29-74

First, I want to give you some basic information about my program. OOP design is considered in my program. Thus, q.cpp and all other 3D models inherit from node.cpp. The node class incorporates a mesh attribute for object storage and a create function within the mesh class that accepts a vertices array, subsequently binding buffers and delegating them to OpenGL. Additionally, the node class encapsulates a material attribute to retain material reflection data, such as Ka, Kd, Ks, and Ns. For instance, in the hand model, I applied default values of Ka=1, Kd=1, Ks=0.3, and Ns=32 to enhance the realism of the generated Q. The node class also contains a texture attribute for object texturing, which I will elaborate on in the texture section.

For the hand model Q, it consists of a ring formed by connecting cylinders and a cube. This class inherits from node class, so it passes the vertices array to the mesh attribute and then to OpenGL to create the model. The default material is used above to increase the reality since my program implemented light reflection. Apparently, Q's vertices array uses more than 800 vertices, which is impossible to write them one by one. But I did not use any software, instead I wrote my own functions to implement them (you can find the original generating function in line 874-914 of q.cpp), and then I copied the implemented vertices and manually set the indices to complete this complex handmade model (I think it should be counted as handmade now, since I have provided a full vertices array).

skybox is also a handmade model, and it is a cube that can automatically change its size based on the camera's zoom parameter.

If you have used software, e.g. Blender, to model your object, please provide a screenshot here showing your model rendered in that software.

If you have not used software to model your object, please state that here.

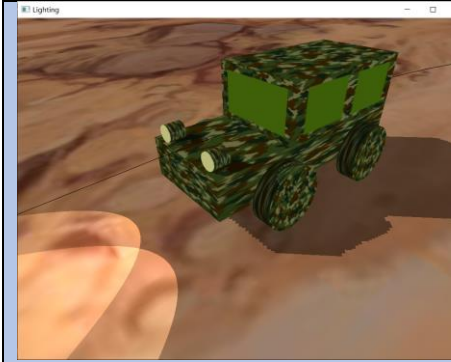
I am not using software modelling

Object 2 - procedurally generated

Please give a screenshot

reference specific code (filename and line)

Description of object



jeep.cpp: line 15-146

mesh.cpp:
line 30-278: code to create some basic 3D shapes.

Jeep's main components include the wheels, chassis, body, and headlights, all of which are created as separate node objects.

The createWheel function constructs the wheel nodes, each as a cylinder object. It sets the wheel's radius, width, position, and rotation, then adds each wheel as a child node of the jeep.

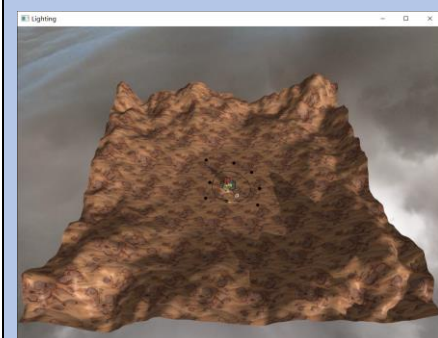
The createLight function forms the headlights, each of which comprises a lampshade and a light node. The lampshade is a cylinder with a camouflage texture, and the light node is also a cylinder but with a luminous texture. Both the lampshade and the light are placed into a container node and added to the jeep.

The create function initializes the jeep's construction. It constructs a jeep object by using the above functions. Additionally, you can find the cylinders, cube, sphere and etc. creation functions in the mesh class.



floorLamp.cpp: line 13-38

The floor lights use a similar approach to the jeep car, creating a sphere and a cylinder to generate.



terrain.cpp: line 16-195

terrain is formed by a square plane and random heights generated using Perlin noise. But this is not covered in the course, so I will explain this algorithm in the R&D section.

Object 3 - OBJ parser

Describe memory allocation and freeing, data structures, how you read the file and parsed the vertex attributes.

Provide the URL for the OBJ file you submitted

Describe how you transferred the vertex attributes to OpenGL

<p>Memory Allocation and Freeing: Dynamic memory allocation is achieved through the new keyword when creating instances of ObjMaterial, Material, Texture, Mesh, and ObjModel. Deallocations are handled using delete in the ~ObjFile() destructor, thus preventing memory leaks.</p> <p>Data Structures: The code utilizes vectors (std::vector) for storing vertices, face indices, and meshes. Maps (std::map) are used to associate material properties and textures with string keys.</p> <p>File Reading and Parsing Vertex Attributes: The OBJ file is read using std::ifstream in the load and loadMaterial functions. The std::getline function fetches each line for processing. Vertex attributes are parsed by fetching data based on line type identifiers (like "v", "vn", "vt", "f"). For faces ("f"), the loadFace function tokenizes the face line, storing each vertex's indices as a glm::ivec3. These indices are then organized into triangular faces for polygonal faces.</p>	<p>https://3d.znzm.com/cgmoxing/168901150.html?ipcity_1136=eq0xR7eDqxuD9DRx0v8tztD%3DKPBIKj5Nx&alichlgref=https%3A%2F%2F3d.znzm.com%2Fmfcgmx.html%3Fkeyword%3D%25E6%2588%25BF%25E5%25AD%2590%26format%3Dobj</p>	<p>My code transfers vertex attributes to OpenGL through a series of processes in mesh class, which is used in every 3D objects (Because every 3D object class inherits node class, and node class contains a mesh object as its private attribute). During the initialization of each type of mesh, the vertex attributes - position, normal, and UV coordinates - are computed and stored in an array or vector. These computed vertices and indices are then passed to the Mesh::create() function which creates deep copies of this data into member variables for future reference. The function then generates a VAO, a VBO, and an EBO using OpenGL functions. It binds and populates the VBO and EBO with the vertex and index data, respectively. Vertex attributes are then specified via calls to glVertexAttribPointer() which dictate how OpenGL should interpret the vertex data in the VBO. These attributes are then enabled using glEnableVertexAttribArray(), after which the VAO is unbound. When it's time for rendering, the Mesh::draw() function binds the necessary VAO and calls glDrawElements() to render the vertex data.</p>
--	--	---

Describe which elements are parsed from your OBJ file, e.g. vertex pos, normal, UV, number of textures or sub-objects etc.

"v": Vertex position; "vt": Vertex texture coordinates (UV coordinates); "vn": Vertex normal; "f": Face indices; "mtllib": Material library reference; "usemtl": Material usage; "Ka", "Kd", "Ks", "Ns": Material properties including ambient, diffuse, and specular color coefficients, and specular exponent respectively; "map_Kd": Diffuse texture map

TR 3 – 3D Transformations

Object 1 - modelled by hand

Please give a screenshot of transformed object	reference specific code (filename and line)	Description of transformations
	<p>node.cpp: line 28-52, line 229-240</p> <p>node.h: line 29-31</p>	<p>Because the program adopts OOP design, so the transformation is implemented using the same logic for every object. The key idea is shown below: Two matrices are saved in the base class Node, a matrix m_LocationMatrix relative to the parent node, and a world matrix m_WorldMatrix. When the position, rotation, and scaling of the node are changed, $m_LocationMatrix = (translate_matrix) \times (rotation_matrix) \times (scaling_matrix)$ is calculated in the function Node::resetLocationMatrix(). Then use the Node::resetWorldMatrix() function to calculate $m_WorldMatrix = parent_node_world_matrix \times$</p>



CW3.cpp: line 201-204

m_LocationMatrix, and finally call Node::resetWorldMatrix() of all child nodes to calculate the child node's world matrix.
Thus, we only need to know the translate_matrix, rotation_matrix and scaling_matrix for each object, then we can calculate the transformation based on the explanation above.
Specifically, Q's translate matrix is (9,2.7,13), rotation matrix is (90,30,0), and scaling matrix is (4,6,4)



skybox.frag and skybox.vert

CW3.cpp: line 287, 288

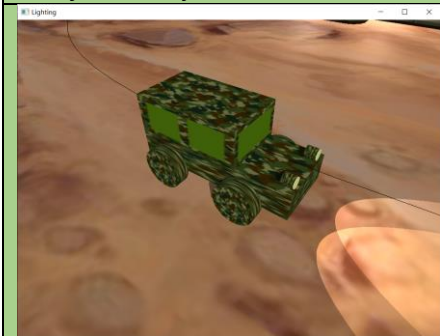
The skybox's view and projection matrices are passed as uniform variables to the shader program with glUniformMatrix4fv(). Also, the camera position (camPos) is passed as a uniform to simulate the skybox's infinite distance. The skybox stays centred around the camera position, giving the illusion of an infinitely large environment.

Object 2 - procedurally generated

Please give a screenshot of transformed object

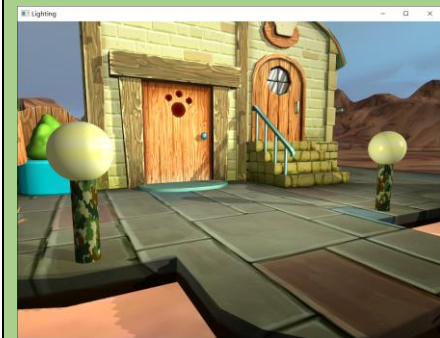
reference specific code (filename and line)

Description of transformations



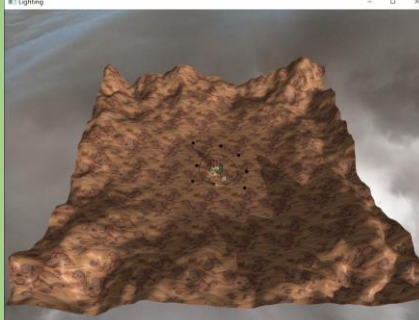



jeep.cpp: line 47-68, line 107-109, line 127-129, line 174-176

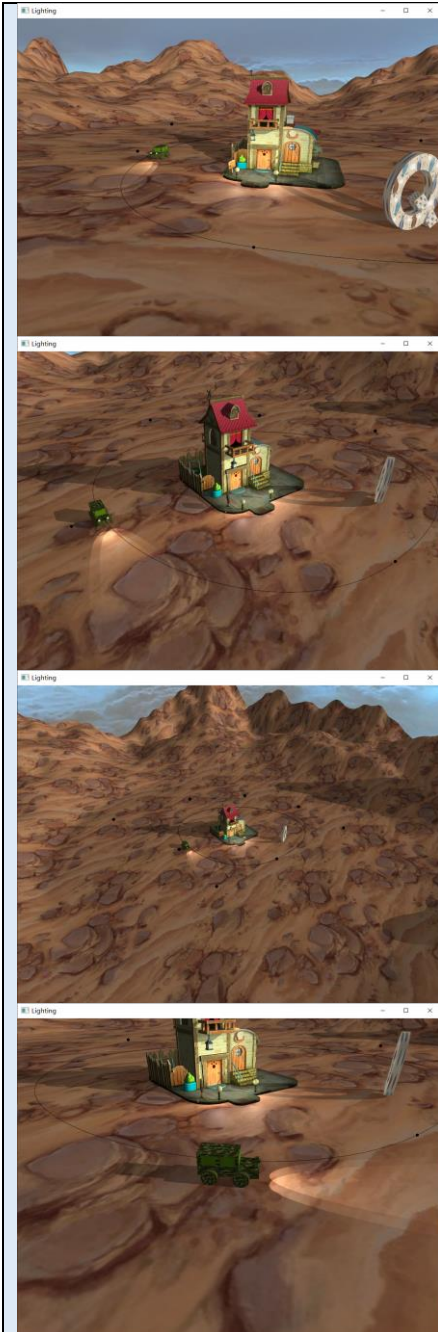
The transformation method is the same as the described in handmade model. For the jeep car, we just set the three matrices for the car body, the chassis, and two lights respectively. You can find the exact values in the lines marked in the code on the left blank. The only notable thing is that this car can move, so the position matrix is changing (I will introduce in the animation part).



CW3.cpp: line 224-235

Floor lights' transformation is also the same as before. You can find the values for the three matrices in the code on the left.

	terrain.cpp	Terrain' transformation is quite complex since I divide it into 240x240 tiles. But this is not covered in the course, so I will explain this algorithm in the R&D section.
Object 3 - OBJ parser		
Please give a screenshot of transformed object	reference specific code (filename and line)	Description of transformations
	CW3.cpp: line 218-220	Floor lights' transformation is the same as handmade model. You can find the values for the three matrices in the code on the left.
TR 3 – Animation		
Please give a screenshot of animated object	reference specific code (filename and line)	Description of animation
	jeep.cpp: line 144-179: add translation to every component, and rotation to the wheels. casteljau.h (will be introduced in curve part, used to create path for moving) CW3.cpp: line 252-261: use the time function to update the jeep car frame by frame	By pressing "1", users can toggle the value of "m_IsMoving" in the jeep class to 'true', initiating the jeep's movement along the Bezier curve I created (path is evaluated in the <i>casteljau.h</i>). This translational motion is applied to the entire jeep object to ensure the synchronized movement of all its components. To enhance the realism of this motion, I also added rotation to the jeep's four wheels and fine-tuned their rotational speed to maintain congruity with the vehicle's linear movement. Moreover, to further ensure that the driving path of the jeep car is my curve rather than straight line movement between the control points, I also use the evaluate function created in lab9 to make the jeep move along the curve.
TR 4 – Camera		
Please give a screenshot	reference specific code (filename and line)	Description of camera
	camera.cpp and camera.h: camera has been encapsulated as a camera class, you can set the initial target point, yaw, pitch, position, distance, zoom etc. in the cpp file. CW3.cpp: line 42-46: use the model-viewer camera, changing to the follow mode. line 125-150: handle the mouse input.	In this project, I incorporated two distinct types of cameras. The primary camera is my default fly-through camera. This allows users to navigate forwards, backwards, left, and right within the scene using the 'W, A, S, D' keys. These movements occur within a horizontal plane (specifically the x-z axis plane), as illustrated in the second figure, when compared to the first one. Camera rotation can be achieved by holding down the right mouse button and dragging the



line 166-167: initialize a camera object.


line 263-285: handle the keyboard input.


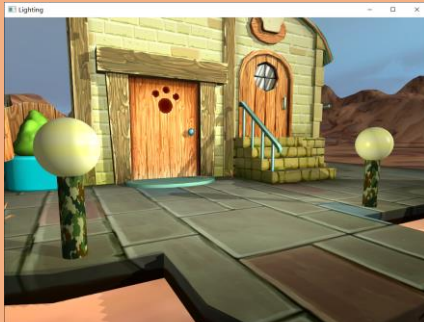
cursor, and its effect is shown in third figure. Importantly, this action does not alter the camera's focus target like the key input. To zoom in or out, users can simply scroll the mouse wheel like the fourth figure. Again, this operation does not affect the camera's focus target. Furthermore, to move on the Y-axis, you should adjust the camera's focus to create a downward angle. Then modifying the camera's distance from the scene through scrolling mouse wheel simulates movement along the Y-axis.


The second camera is a modified version of the model-viewer camera from lab6a. Unlike in the fly-through camera, mouse actions do not shift the camera's focus target. Consequently, if only mouse inputs are used, my fly-through camera functions as a model-viewer camera. However, it was ineffective as it could focus on scene without objects. To address this, I specifically tailored the model-viewer camera to the jeep car. By pressing '2', users can shift the camera's target point to the jeep, ensuring that the camera follows the vehicle even during motion like the last figure. In this mode, the camera maintains the jeep at the screen's center, while the field of view can still be altered via mouse swipes and scrolls. Keyboard inputs are disabled in this mode to preserve the fixed camera focus target.

TR 5 – Texture

Object 1 - modelled by hand


Please give a screenshot	reference specific code (filename and line)	Description of texture
	<p>texture.h and texture.cpp: explain in the right blank.</p> <p>node.cpp: line 60-72: use to set texture, material, and mesh (vertices, norm and etc.) line 204-206: use to render the texture with object.</p> <p>q.cpp: line 870: the hand made Q is a subclass of node. My project has implemented good OOP design. node is the parent class of most objects to manage its mesh, material, texture etc.</p>	<p>The texture.cpp file encapsulates the texture as a class. When instantiating the class, it calls init function to import all textures needed for the project into s_Textures array. load function in texture.cpp is the combination of setup_texture function and setup_mipmaps function implemented in lab6b. Therefore, load function imports the texture with mipmap to enhance the quality of the image. Specifically, The data information of texture in load function is read using loadbitmap in bitmap.h implemented in this lab.</p> <p>For the Q class's texture, its texture ID is stored as TEXTURE_ID_Q (check line 870</p>

	<p>skybox.frag: bind the texture for skybox</p> <p>CW3.cpp: line 290-292: pass the texture to the shader</p>	<p>in q.cpp), and you can call the texture stored in s_Textures with this variable. Then it passes the texture to the m_Texture variable in node class (q class inherits this variable from it) and bind it when calling the render function inherited from node class.</p> <p>Skybox set its texture in CW3.cpp, and then pass the texture into its shader to render. And the texture it used is TEXTURE_ID_SKYBOX.</p>
Object 2 - procedurally generated		
Please give a screenshot	reference specific code (filename and line)	Description of texture
  	<p>texture.h and texture.cpp: class used to manage every texture.</p> <p>node.cpp: line 60-72: use to set texture, material, and mesh (vertices, norm and etc.) line 204-206: use to render the texture with object.</p> <p>Jeep.cpp: line 17-23, 40-46, 52-58, 97-106, 113-119, 126: jeep is also the subclass of node.</p> <p>floorLamp.cpp: line 15-18, 28: floor lamp is the subclass of light, and light is the subclass of node.</p> <p>terrain.cpp: line 193: terrain is also the subclass of node.</p> <p>CW3.cpp: line 208: creating the terrain in main function and specify the tileSize here.</p>	<p>As described in my hand made texture, I've initialized and stored all the textures used in the project. A jeep car uses different textures for different components since it's a complex object consists of two cubes, six cylinders and two lights.</p> <p>Specifically, TEXTURE_ID_CAMOUFLAGE is used for the texture of the wheels (line 17-23 in jeep.cpp), the texture of the lamp bodies (line 40-46)), the texture of the chassis (line 97-106), and the texture of the roof (line 114). TEXTURE_ID_LIGHT is used for the luminous part of the top of the car lamps (line 52-58). TEXTURE_ID_JEEP_SIDE_WINDOW and TEXTURE_ID_JEEP_FRONT_WINDOW are used in the texture around the body to simulate the windows (line 115-118, 126).</p> <p>Floor lamps use TEXTURE_ID_CAMOUFLAGE as their bodies, and TEXTURE_ID_LIGHT as the bulbs.</p> <p>The terrain specifies a texture using TEXTURE_ID_TERRAIN and it is tiled to the entire terrain, and you can specify how far to cycle the texture in the main function (described in the modelling part).</p>
Object 3 - OBJ parser		

Please give a screenshot	reference specific code (filename and line)	Description of texture
	<p>objFile.cpp: line 103-115: read the textures according to the material file.</p> <p>objFile.h: line 49: specify a texture object in the objFile class.</p>	<p>Read the filename of the texture from the obj's material file, then load its texture. The only notable thing here is that obj's textures are stored separately from other textures. When obj is created, a separate texture object (the texture class) is created for it.</p>



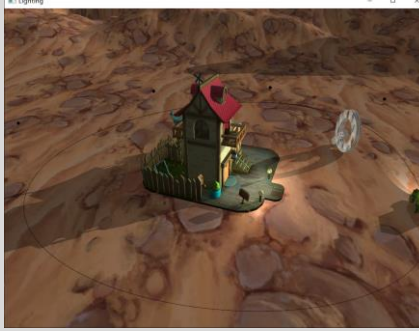

TR 6 – Lighting

Light 1

Please give a screenshot	reference specific code (filename and line)	Description of light
	<p>light.h and light.cpp: explain in the right blank.</p> <p>phong.frag: line 72-78: calculate the color contribution of directional light.</p> <p>CW3.cpp: line 196: create the environmental light, set the position based on the texture of the skybox and strength of this light.</p>	<p>I encapsulated light I implemented in lab7 to the light class (you can set attenuation, colour and on/off etc. in this class). This encapsulation makes my project easy to add a new light. By modifying the shader provided in lab8, I implemented light reflection and the blending of multiple lights. These enhancements facilitate a more sophisticated and realistic lighting system than the one found in lab8.</p> <p>The blending of multiple lights is achieved through a straightforward approach: the main function in phong.frag calculates the sum of the contributions of all lights in the scene (see lines 107-128 in phong.frag). This method, which contrasts the lab8's single-light approach, allows for complex interactions among different lights in the scene. Light reflection, a relatively intricate component not covered in the course, will be explained in the R&D section.</p> <p>While the above-mentioned concepts provide important context, let's back to light 1. This light is a directional light, introduced in lab7, utilized in my program due to the implementation of a skybox texture featuring a sun. Consequently, I positioned a directional light to align with the sun in the skybox, enhancing the natural appearance of the scene's illumination.</p>


Light 2

Please give a screenshot	reference specific code (filename and line)	Description of light
--------------------------	---	----------------------

	<p>phong.frag: line 89-104: calculate the color contribution of spot light.</p> <p>jeep.cpp: line 61: create the car lights, set the position based on the car and other parameters like attenuation, theta, phi and etc.</p>	<p>As I described in the jeep car, this object consists of two lights. The type of the two lights is spot light, which is implemented in lab7. You can press '3' to turn the headlights on or off. Moreover, as I mentioned above, light blending is implemented. Therefore, some area is super bright because of the blending of this spot light and environmental light (directional light).</p>
Light 3		
Please give a screenshot	reference specific code (filename and line)	Description of light
	<p>phong.frag: line 79-88: calculate the color contribution of positional light.</p> <p>floorLamp.cpp: line 31: create the floor lights, set the position and attenuation.</p>	<p>As I described in the floor lamp, we have two floor lights in the scene. The type of the lights is positional light mentioned in lab7. You can press '4' or '5' to turn on the left/right light. Moreover, as I mentioned above, light blending is implemented. Therefore, some area is super bright because of the blending of this positional light and environmental light (directional light).</p>
TR 7 - Shadow		
Please give a screenshot	reference specific code (filename and line)	Description of shadow
	<p>CW3.cpp: line 111-120: generate the depth map. line 104: pass the depth map to the main shader (phong.frag)</p> <p>shadow.h: the same as the file implemented in lab8, which is used to calculate the shadow map.</p> <p>shadow.frag and shadow.vert: the same as the file implemented in lab8, use this shader to generate shadow since we don't want any light effect.</p>	<p>The idea is the same as lab8: First create a depth texture, then render the scene into the depth texture from the light perspective, and finally convert the vertex to the light perspective when rendering the scene and compare it with the z in the depth texture to determine whether the position is in the shadow.</p>
TR 8 - Interactive object		
Please give a screenshot	reference specific code (filename and line)	Description of interactive object
	<p>CW3.cpp: line 34-57: handle keyboard input. line 59-67: change the window size. line 122-150: handle the mouse input.</p>	<p>As described in the above animation and camera parts, many of the objects including camera, jeep car and all of the lights in my scene possess interactive capabilities. I won't go into the details of their implementation in this context since I have mentioned in the above parts, and here I just I would like to provide a concise summary:</p> <p>Key 1: runs or stops the jeep car. Key 2: change the camera mode to fly-through camera or model-viewer camera (follow the jeep car). Key 3: turn on or turn off the car lights. Key 4: turn on or turn off the left floor light.</p>

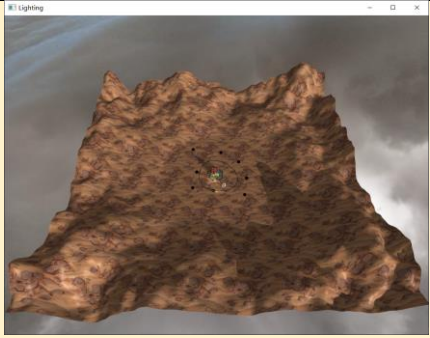
		<p>Key 5: turn on or turn off the right floor light.</p> <p>Key 'Esc': close the application.</p> <p>Scroll mouse wheels: zoom in or zoom out the scene.</p> <p>Right click mouse button: rotate the camera.</p> <p>Window size can be changed to just click the maximize button or drag the edge of window.</p>
--	--	--

TR 9 – Curves

Please give a screenshot	reference specific code (filename and line)	Description of curve(s)
 <p>PS: The curve uses the black line color, and the black control points color, so it doesn't look obvious in the picture.</p>	<p>curve.cpp and curve.h: divide the lab9's main function into create, createBuffer and render function (Because curve is a 2D shape, it can't inherit the render function from node class).</p> <p>casteljau.h: the same as the lab9's header file, which is used to generate the curve based on control points.</p> <p>curve.frag and curve.vert: specify a new shader to handle the 2D curve.</p> <p>CW3.cpp: line 304-308: use shader to render curve.</p>	<p>I encapsulated the curve implemented in lab9 into the cruve class. The create function generates nine control points in a sequential manner and employs the evaluate function from the casteljau.h file to connect these control points into a curved form. Then, the createBuffer function is utilized to bind the buffers for the curve. Lastly, the rendering aspect is implemented in the main function using a specialized shader designed for 2D curves. Additionally, the main objective of the curve is to serve as a path for the jeep car. This is primarily achieved by passing the points evaluated from the curve to the jeep car. By doing so, the car is able to move gradually along the curve, frame by frame, based on these specified points.</p>

R&D

Please provide details of any research and development you conducted, as additional technique not in the lecture notes.

Please give a screenshot	reference specific code (filename and line)	Description of Research including websites, articles, references, etc.
	<p>terrain.cpp: line 16-195</p>	<p>The create method generates a terrain grid with a specified tileSize and tileNum. The grid is essentially a plane divided into tileNum by tileNum square tiles, each of size tileSize. For each tile, it calculates the vertex position (pPosition), the vertex normal (pNormal), and the texture UV coordinates (pUV). For each tile, it also creates two triangles (idea from lab6b) with six indices and adds them to the index buffer (indices). The v0, v1, v2, v3 represent the indices of the four corners of each tile. Next, the terrain elevations are generated using a multi-frequency Perlin noise function. This noise function produces smooth and natural terrain shapes. The frequency and amplitude are specified in the frequency and amplitude arrays. The Perlin noise is sampled at each vertex position and the result is used to set the y value of the vertex, creating hills and valleys. Once the positions are defined, the normals for each face are calculated as the cross product of two</p>



phong.frag

material.h: line 12-15

edges of the triangle. Vertex normals are then calculated as the average of the normals of the faces sharing that vertex. These normals are important for shading calculations when the terrain is rendered. The terrain flattens at the center specified by centerRadius, thus creating a flat area at the center and elevated terrain in the outer regions. The texture coordinates for each vertex are calculated as the ratio of the vertex's x and z coordinates to the textureTileSize. This will allow a texture to be mapped onto the terrain, with the texture repeating every textureTileSize units. Finally, the vertices, normals, UVs, and indices are all stored in a new Mesh object, and the mesh is paired with a Material and a texture to complete the terrain model.

The creation of my terrain refers to a blog in a chines forum - CSDN (Chinese Software Developer Network):

<https://blog.csdn.net/birdflyto206/article/details/110730044>

I have upgraded the phong lighting from lab8, and introduced light blending and light reflection. Since I have mentioned light blending in the lighting part, here I would like to only focus on light reflection. For the light reflection, because my obj reads and processes various reflection coefficients in material, and thus I decided to handle them correctly.

The coefficients including ambient (K_a), diffuse (K_d), and specular (K_s), are considered when calculating the color. For each light source, my shader calculates the color contribution due to diffuse and specular reflections, where the Phong shading model is used. Specifically, the diffuse term represents the amount of light scattered uniformly in all directions, while the specular term captures the mirror-like reflection of light. The latter is controlled by the shininess coefficient (N_s) which impacts the spread of the specular highlight. By integrating these factors, my shader provides a realistic lighting effect with reflectance.

The implementation of it refers to the learnopengl website:

<https://learnopengl.com/Lighting/Materials>

The skybox class includes methods for creation and rendering of the skybox. Initially, in the 'create' function, the



vertices for the six faces (front, right, back, left, top, bottom) of the skybox cube are defined with respect to a size parameter. Specifically, It was defined using the vertices array and buffer binding skill I learned from the labs. But for the transformation part, the render function renders the skybox to the screen, which appears as an infinite expanse due to the camera's transformation being applied only to the rotation (not the translation). This gives the impression that the skybox is always distant, regardless of the camera's position in the world. Again, the idea is from learnopengl website: <https://learnopengl.com/Advanced-OpenGL/Cubemaps>

Conclusion		
<i>Please describe what you perceive to be the strengths and weaknesses of your project</i>	<i>Describe what aspect of it you are particularly proud of, and what you think would need to be improved.</i>	<i>Reflect on what you have learned during this project that you can apply in future projects to improve your performance.</i>
<p>Strengths:</p> <ol style="list-style-type: none"> OOP Design: I adopted OOP design and encapsulated the code I implemented in lab into classes (textures, camera, curve, mesh), and created node class to manage them, which increase the reusability and made my program structure clearer. OBJ Loader: My obj loader can handle most obj file since I have read and stored not only the vertices, textures, norms, faces (v, vt, vn, f), but also the materials information (Ka, Kd, Ks, Ns) Lighting System: I self-studied some lighting skills to add light reflection into the shader to correctly handle the obj materials' information; and combine different lights together in the scene to render a complex lighting. 3D Terrain: I self-studied how to create a 3D terrain with hills and valleys using mathematics calculation for norm and Perlin noise. Camera System: I self-studied how to create a fly-through camera and made the target variable updated in the render loop to create a upgraded model-viewer camera which could follow the car even it is moving. Skybox: I self-studied how to create a skybox, and bind the camera position with it to make a infinite space. 	<p>Proud:</p> <ol style="list-style-type: none"> I'm super proud of my OOP design, I can easily reuse my code, for example you can create a new car with 2 cubes, 4 cylinders and 2 spotlights directly using my jeep class to create a node (which is a parent class for every object in the scene except curve). Node class contains the mesh class to bind buffer, the texture class to bind texture (if you want), and material class to bind material information (if you want). Then, the jeep node can be rendered directly using node's render function. Therefore, OOP gives my program the possibility to generate a new complex object in just a few second. Furthermore, OOP made the objects I created easy to manage. For example, I added a unified motion to the jeep class in the program and added a separate rotation to the wheels to simulate the driving of the car. I encapsulated light in the light class (you can set attenuation, colour and on/off etc. in this class) and added light reflection and light combination by changing the shader provided in lab8. These two operations made my program have a more realistic lighting system than lab8, and easy to add a new light. 	<p>This project has significantly deepened my understanding of various concepts such as Object-Oriented Programming, 3D model rendering, complex lighting systems, procedural generation of 3D terrains, dynamic camera systems, and the creation of comprehensive interactive systems and mimic animation. These skills will undoubtedly prove essential in future work involving game development or large-scale environment creation. Additionally, I have honed my 3D modeling skills through the creation of both procedural and handmade models. However, this project has also highlighted the need for further development of my artistic skills to create visually compelling scenes. In the future, I will strive to balance technical proficiency and artistic creativity for a more holistic approach to game and graphics development.</p>

<p>7) Handmade/Procedural Models: I do believe that both handmade Q and procedure-generated jeep cars, terrain and floor lights are highly complex.</p> <p>8) Interactive System: I implemented a rich interactive system, including car motion control (translation and rotation at the same time), light control, camera control and switching.</p> <p>9) Anti-aliasing and mipmap: I implemented those techniques to enhance the quality of textures. Anti-aliasing refers to the line 155 and line 172 in CW3.cpp, and mipmap refers to the bitmap.cpp, which is developed during the lab.</p> <p>Weaknesses:</p> <p>1) Lack of Art: I don't think I have enough art skills. Although I implemented solid code, I didn't generate a scene with really good visual effects. As someone who wants to work in the game industry, I think this is where I need to improve in addition to code technology in the future.</p>	<p>3) Since the floor in lab8 was poor and it was obtained by scaling down a cube, which made it look like a 2D image. I wanted to create a real terrain. And I got an inspiration from lab6b's grid. Therefore, I created a lot of grids as a basic floor, and each grid was handled by Perlin noise to add height for them. However, to ensure that there is no peak in the centre of the scene that blocks my other objects, I also set a reasonable radius to control the vertices height to 0. And I increased the height of the vertices according to the distance from the center of the image, which made it more natural for the image to transition from flat to peak.</p> <p>4) I'm proud of my procedurally generated jeep car, Its composition is more complex, and the realization of multiple motion, and multiple lights. And I believe the final effect is even better than some loaded obj.</p> <p>5) I am also proud of my handmade model, Skybox, cameras, obj loader and etc. I tried to do everything at the highest possible standard (refers to the sample1), and I do think I achieved a similar effect as it. There is not enough space to go on those parts separately, and I've covered this in more detail in the previous sections. So let's just stop here.</p> <p>Improvement plan:</p> <p>1) I would enrich the scene using fit obj. file or other procedurally generated objects (since handmade objects are really hard to make it complex), maybe increase my art taste in the future.</p>	
--	---	--