UNIVERSITY OF LEEDS

School of Computing: Assessment brief

Module title	Foundations of Modelling and Rendering
Module code	COMP5812M
Assignment title	A1.1 Raytracing and A1.2 Rasterisation.
Assignment type and description	Programming assignment.
Rationale	You will be implementing the two most common approaches to rendering 3D scenes, and applying most of the foundational content from this module.
Word limit and guidance	This is a pass/fail assignment. You will complete it after a successful demo at the lab class. Attendance is encouraged.
Weighting	50%
Submission deadline	15/12/2023
Submission method	In-person Demo + Q&A + Gradescope.
Feedback provision	In-person feedback.
Learning outcomes assessed	LO1, LO2, LO3, LO4, LO5, LO6, LO7, LO8, LO9, LO10
Module lead	Rafael Kuffner dos Anjos
Other Staff contact	

1. Assignment guidance

This assignment is divided into two tasks, the Raytracing and Rasterisation components. Please see the individual description of tasks for each one.

2. Assessment tasks:

A1.1 – Raytracing

This is the first of the two exercises that need to be completed as part of your first assignment. The starting point for this assignment is the end of the Raytracing exercise which is a fully guided tutorial. If you haven't so, please do it now. The final goal is to have a working raytracer with phong shading, hard shadows, and mirrors implemented. This will be the output of correctly implementing the tasks described ahead

Task 1: Transformations
Your first task will be finishing the code that will perform the transformations. Implement a function that should return your modelview Matrix on the Scene class. Here is a starting point.

```
Matrix4 Scene::getModelview()
{
    Matrix4 result;
    result.SetIdentity();
    //TODO: Grab all the necessary matrices to
    // build your modelview. And return from this function.
    return result;
}
```

Then, use this function to transform your vertices and normals inside the updateScene(). Example:

```
//TODO: apply modelview here, before adding vertex to triangle.
t.verts[vertex] = v;
```

Finally, add a "Scene" object to our RaytraceRenderWidget. Initialize it in the constructor, and call "updateScene()" when we press the raytrace button.
Does it work: Use the debugger to verify that the matrices are being updated accordingly when you use the interface.It will however be easier to tell in the when you are able to see things.

Task 2: Casting a ray
Implement a function on RaytraceRenderWidget that given a pixel position, casts a ray towards the scene.

```
Ray calculateRay(int pixelx, int pixely,bool perspective);
```

Implement this function by following the process described on Lecture 4, slides 34-38.
Call it inside your main raytracing loop, before calculating a colour for a given pixel.
Does it work: Use the breakpoints to see the coordinates your rays match the edges of your image plane. Those should be similar to what you have as the Left,Right,Bottom,Top from the "glFrustum" on RenderWidget.cpp. It will however be easier to tell in the next step when you are able to see things.

Task 3: Geometric Intersections
Calculate geometric intersections between camera rays and triangles in the input model. Use an auxiliary struct "CollisionInfo" to contain the intersection information, and implement the following function (starting point below).

```
//scene.h
 struct CollisionInfo{
        Triangle tri;
        float t;
    };

CollisionInfo closestTriangle(Ray r);

//scene.cpp
Scene::CollisionInfo Scene::closestTriangle(Ray r)
```

```
{
    //TODO: method to find the closest triangle!
    Scene::CollisionInfo ci;
    ci.t = r.origin.x; // this is just so it compiles warning free
    return ci;
}
```

This function should iterate the list of triangles in the current scene, and test for intersection with them. The closest triangle given a ray will have the smallest t (See Lecture 3, Slides 35-38).
To calculate intersections, implement a method in your Triangle Class.

```
float intersect(Ray r);
```

Follow the method described in the slides, and return the calculated t for a given intersection, as we can calculate the point of intersection given a ray's origin, direction, and t. We are only interested in cases where t > 0, as negative t would mean an intersection behind the camera. Use this to your advantage to encode "no collision" cases with a negative t value.
To verify if the intersection point with the plane formed by the triangle is inside the triangle (thus, a valid intersection) implement the half plane test. (See Lecture 3, Slides 41-44).
Does it work: Call closestTriangle inside your main raytracing loop. If t > 0, assign the color white to the pixel. Use the sliders and arcball to verify that your transformations are matching with the openGL renderer. Example outputs can be seen on Minerva.

Task 4: Barycentric Interpolation

Implement barycentric interpolation to obtain the barycentric coordinates at the position of the intersection calculated in the previous task. Add a function to your triangle class that returns the "alpha, beta, gamma" as a Cartesian3. Here is a starting point

```
//add to .h
Cartesian3 baricentric(Cartesian3 o);
//add to .cpp
Cartesian3 Triangle::baricentric(Cartesian3 o)
{
   //TODO: Input is the intersection between the ray and the triangle.
   //o = origin + direction*t;
   Cartesian3 bc;
   bc.x = o.x ; // Just to compile warning free :)
   return bc;
}
```

Does it work: Use this function inside your main loop. Use the barycentric coordinates to calculate the normal vector for that point. Then set the color of the output pixel as the following, which should show the value of the normal if we have the checkbox "interpolation" checked:

```
if(renderParameters->interpolationRendering)
     return Homogeneous4(abs(normOut.x),abs(normOut.y),abs(normOut.z),1);
```

Example outputs can be seen on Minerva.

Task 5: Blinn-Phong Shading

Calculate Blinn-Phong shading considering every light present in the scene. Use the lights in the vector that you populated in Exercise #2, on the RenderParameters class. Remember to apply the modelview matrix to them as well, as they are tied to a physical object in the scene and these are subject to the sliders and arcball.
Implement the code to calculate phong shading in a function of your Triangle class (Lecture 4, Slides 7-19). Parameters should be the light position, light colour, and the intersection barycentric coordinates. Use the material properties for that given triangle to calculate phong.
Research and implement quadratic attenuation to mimic what happens on the left side (see exercise 4).
Return the final colour as Homogeneous4 and apply it to the pixel.
Does it work: Compare it to the shading that happens on the Minerva examples. Change the mtl file of objects you try to see if your implementation responds properly to it.

Task 6: Shadow Rays

Include shadows in your shaded result, also considering every light source in the scene. Follow the instructions on the slides (Lecture 5, slides 4-8).
Use the "closestTriangle" function you implemented before to perform intersection tests, and the t value to check if there was a valid intersection.

Pay attention to shadow acne, correctly displacing the starting point of your shadow ray.
Adjust your phong shading function to only apply ambient and emissive colour if the object is in shadow.
Does it work: Compare to the Minerva examples again. Move the object around, zooming in, and using different angles, ensuring shadow acne never happens.

Task 7: Impulse Reflection
Use the material properties to verify if the object is mirrored. If it is, calculate the reflected value which should be combined with the object's colour accordingly (values are 0 to 1). This will require you to change your raytracer into a recursive function.
Move out all of your shading code after calculating the ray out of the main raytracing loop into a separate function:

```
Ray r = calculateRay(i,j,!renderParameters->orthoProjection);
Homogeneous4 color = TraceAndShadeWithRay(r,N_BOUNCES,1.0f);
//... applying to pixel, gamma correction etc
```

Where N_BOUNCES is defined in your .h file with the maximum number of recursions allowed in your raytracer.
An object is reflective if the mirror property in the material is different than 0. If that's the case, you should find the color at the end of this reflection. Implement a "reflectRay" function that reflects a ray according to a surface normal.

```
reflectRay(r,normal,hitPoint);
```

And recursively call TraceAndShadeWithRay, as described in the slides (Lecture 5, slides 10-13). In the material model we are using in this assignment, the "mirror" property in the material file is a floating-point value meaning how much of the total energy should be from the reflected ray. Use this to linearly combine the resulting color of the mirror ray, and the color of the current surface, making sure that no energy is lost, or added to the system.
Does it work: Again, check examples on Minerva to compare. Change the material properties to verify that when mirror = 1.0, all you see is a reflection. When mirror=0.0 you should not see any reflection. Anything else should be a combination of colours from phong shading and the reflection.

A1.2-Rasterisation

This is the second of the two exercises that need to be completed as part of your first assignment. The goal is to have a working terrain renderer using rasterization, applying phong shading, using materials stored in textures, normal mapping, and a smooth transition between materials according to the height of the terrain. This will be the output of correctly implementing the tasks described ahead.

Task 1: Terrain displacement map
You are provided with a heightmap texture (mountains_height.bmp). This textures encodes what is the height each pixel should have. We are going to use the UV coordinates of the terrain created in the tutorial to map this texture to it, and use it to set the vertex to the correct height in the vertex shader, given a certain scale constant.
For each pixel in the heighmap, you have a 24 bit integer encoded in the red, green, blue channels of your texture. As an example, let's say a pixel has the height 12345678 This will be encoded as such in RGB values according to the significance of that byte:

R = 10111100 = 188 = 0.73725	G = 01100001 = 97 = 0.38039	B = 01001110 = 78 = 0.30588

Meaning you can reconstruct the original value using shift operations!
12320768 + 24832 + 78 = 12345678

Add the required code to load this texture (similarly to what happens in the tutorial) paying attention to what is the sampling mode you are going to use (Lecture 9,

Does it work: You should be seeing mountains! Check the images on Minerva to see if it matches.

Task 2: Terrain normals
Calculate a normal per each vertex. This can typically be done by calculating derivatives along both tangents of the plane. There is a simple way of estimating it in our particular case. We can calculate the correct normal vectors by sampling the values around the current pixel and estimating their differences. Considering we are reading the pixel at position 4, this is one method to calculate it.

0	3	6

1	4	7
2	5	8

Nx = differences between 0-6, 1-7, 2-8
Ny = fixed value.
Nz = differences between 0-2, 3-5, 6-8

Where you can give a higher weight the central differences (1,7,3,5). You are free to research and implement an alternative version if you provide a source for it and can explain it clearly.
Implement this method in your vertex shader, and change the fragment shader to instead of outputting a colour from the texture, to output the normal vector as a colour, such as: vec3(abs(nx),abs(ny),abs(nz)). For that, change the outputs of the vertex shader, and inputs of the fragment shader so the value can go through.
Does it work: You should see varying colours around the cliffs, and they should keep the same value as you move the camera around.

Task 3: Terrain shading with texture materials
Implement Blinn-Phong shading in your terrain, using the normal vector calculated in the previous step. This should be implemented in your fragment shader. Perform this calculation in the View Coordinate System (See transformations lecture). This will require you passing more matrices as uniforms, and more variables from your vertex to your fragment shader. Set a new uniform with a directional light, which you will use in your calculations.

```
glm::vec3 lightPos = glm::vec3(0, -0.5, -0.5);
```

Remember that when using a directional light, the light vector is always the same!
I have provided a Diffuse and a Specular texture for each one of the different terrains. Continue using the rock and sample these textures to correctly set the materials. This will require adding more textures to the "load textures" method and passing them as uniforms. Be aware of the Filtering mode!
Implement tiling in how the textures are sampled, so they are at the right scale. This can be easily achieved by multiplying the UV coordinates by a constant and setting the boundary behaviour properly (GL_TEXTURE_WRAP_S and T).
Does it work: You should see parts of the rock texture being shiny, and others not. Check the example on Minerva for details.

Task 4: Terrain detailing with normal mapping
Use the normal map texture provided to apply normal mapping to your terrain. Using the normal vectors you have, and the UV coordinates, calculate the tangent vectors necessary for implementing normal mapping (See the advanced texturing lecture for the method).
You should implement this in your vertex shader. Remember that you can calculate the UV coordinates based on the number of points you have in your mesh. You don't necessarily need to pass them as arguments. Remember to do Graham-Schmidt process to make sure your basis is orthogonal. Implement the actual normal mapping then in your fragment shader.
Does it work: Your shading should be a lot more detailed now, even without increasing the resolution of the mesh.

Task 5: Terrain material interpolation
You have a texture of grass, rock, and snow. Make it so the fragment shader can choose which texture to apply according to the height of what it is shading. There should be a smooth transition between different textures being used. Pass the height as a parameter between the vertex and the fragment shader, and use it to mix the materials from the different textures.
Does it work: Transitions should be smooth! See example on minerva.

Task 6: Alternative rendering modes and useability
As a final task, I would like you to implement some functionalities in the renderer to allow some implementation of interaction in the C++ side.

- Implement rendering using wireframe that is activated while you press space. Research the function "glPolygonMode" to accomplish it.
- Recompile your shaders when you press R. That would allow you to change the shader and recompile them in real time to see the changes in effect.
- Allow the WASD keys to rotate your directional light.
- Use the T and G keys to control the scale of the terrain.

Does it work: All the buttons should be functional!

3. General guidance and study support

Consult the Minerva page to obtain the slides for this module, and references in the reading list. Your main source of support for this assignment is the laboratory sessions.

4. Assessment criteria and marking process

Marking will happen during the laboratory classes, thus, the deadline for completing this exercise is the last scheduled laboratory class. However, it is strongly encouraged that you take an iterative approach to this, and work your way through the exercises as quickly and regularly as possible. There are no "gotchas" in these exercises that we cannot support you with. So: start working on it, as soon as you have a problem or don't know how to progress, ask for support during the labs, and we will help you continue.
Demo: When you believe you have completed all the exercises for either rasterization or raytracing, inform the module team, and we will verify that you have completed everything to specification by looking at your prototype running. If there are any problems, we will inform you so you can continue working on it and fix them.
Q&A: Then we will ask you a few questions about your code that may include explanation about what you did, theoretical questions about the material you had to use to implement it, and requests to change some functionality. The goal of this Q&A session is to verify that you understand the code thoroughly. If you fail to respond to any questions, we will stop the Q&A and resume when you are able to respond correctly. Expect questions related to any one of the implementation tasks, and also to the guided tasks in the exercise.

5. Presentation and referencing

Questions will be asked and answered in English. Comments on your code must be written in English or they will be ignored.

6. Submission requirements

Gradescope Submission: After each demo and Q&A steps are completed, you will be given a "digital receipt" that you have been assessed.
A1.1: Prepare a zip folder with the "Raytracing" project folder and the receipt file, but without the "objects" folder.
A1.2: . Prepare a zip folder with the receipt and the following files:
main.cpp, Basic.vert, Texture.frag.
Please do not submit anything else, as the total size of the project will be too large.

Submit both zip files to Gradescope as a single submission. I would encourage you to submit as soon as you are done for "backup", and update when you finish both exercises.

7. Academic misconduct and plagiarism

All of the code submitted will be verified using plagiarism detection software. You are not allowed to submit code that was not written by you. In this assignment, it is not allowed to use any external sources for the code you submitted, even with references. All the submitted code must be written from the start by you.

The Q&A sessions will be also used to verify that you wrote the code yourself, and that you are familiar with all aspects of it. If any member of staff suspects of plagiarism at this point, you will be warned at this point and your submission will be closely investigated after submission to Gradescope.

8. Assessment/ marking criteria grid

50 points: Everything works, Q&A completed successfully.