Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Centre for Language and Communication Studies

Assignment Submission Form

Student Name:	Eoin Brereton Hurley
Student ID Number:	19335447
Module Title:	CSU44052-202223 Computer Graphics
Please fill in one of the following as appropriate:	
MPhil course (Applied Linguistics, Linguistics, etc.)	
TCD undergraduate (indicate degree subject or TSM subjects)	X
Visiting student (or other student category)	
Lecturer(s):	Prof. Rachel McDonnell & Prof. Carol O'Sullivan
Date Submitted:	06/01/2023

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

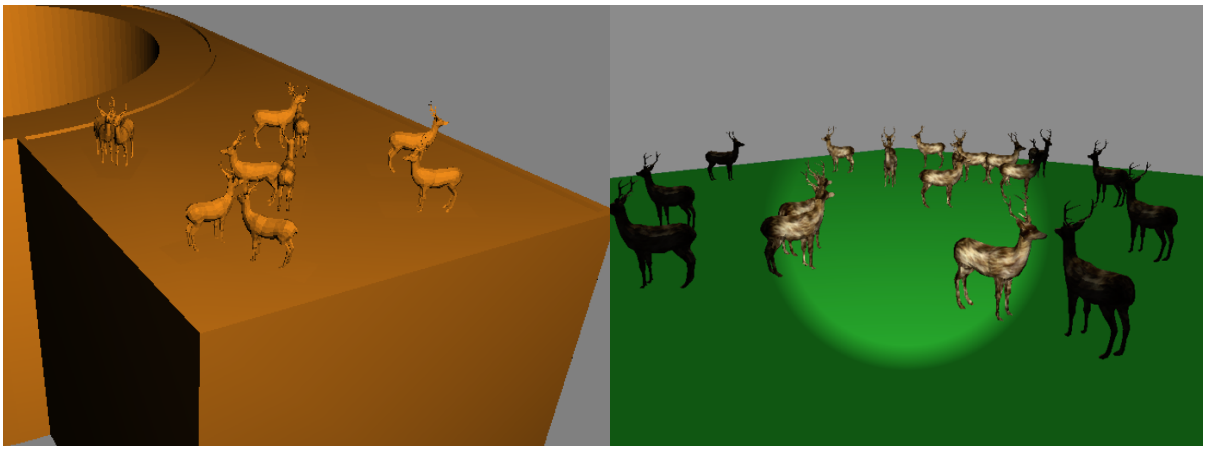
I declare that the assignment being submitted represents my own work and has not been taken from the work of others save where appropriately referenced in the body of the assignment.

Signed Eoin Brereton Hurley **Date:** 06/01/2023

Name:	Eoin Brereton Hurley
Student ID:	19335447
Declaration	See Page Above
YouTube Link 1: (Required Features)	https://youtu.be/a-nCgE2R3z4
YouTube Link 2: (Final Demo)	https://youtu.be/Z7AzUAgmBZ8

Required Feature 1 (Crowd of animated reindeer):

Screenshot(s) of feature:



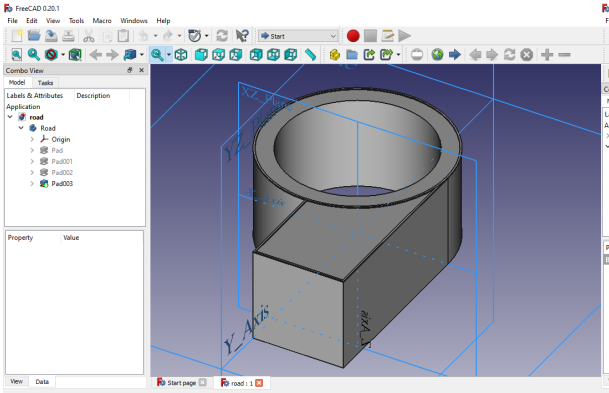
Preliminary Reindeer Crowd Rendering *Finished Reindeer Crowd Rendering*

Description of implementation:

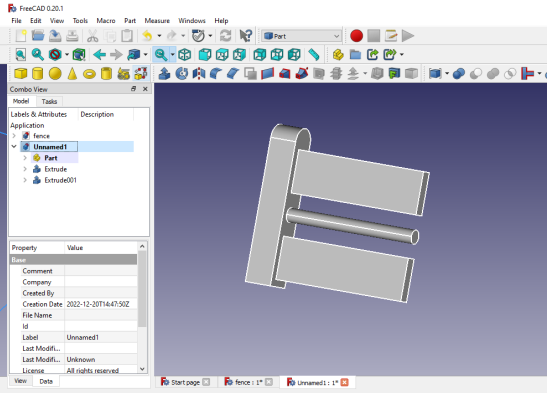
The first step in producing a crowd of reindeer involved finding a way to render multiple models on the screen. This was achieved using two custom made classes: `ModelMesh` and `Model`. (The code of these classes is provided below). The `ModelMesh` class allows 'base' objects to be instantiated for each model. These base objects contain the necessary information unique to each model, including all of the vertex information like position, normal and texture coordinates. This information is extracted using the `load_mesh` class of type `ModelData`. The implementation of a base class in this way is quite advantageous as it means that the mesh associated with a given model only needs to be loaded once. For example, multiple reindeer are rendered for the crowd without having to reload the mesh for each one. The `Model` class is instantiated after the `ModelMesh` class and contains a pointer to the appropriate `ModelMesh` object. This subclass provides all of the extra detail needed for each single object on the screen. Objects of this class have their own x, y and z rotation as well as their own x, y and z translation. With this feature, each object rendered on screen may be easily manipulated, thus facilitating easy animation.

The reindeer body and each of its legs are all separate models. During the render loop, these elements are formed hierarchically. The reindeer bodies are translated and rotated to different positions and then each of the legs undergo the same transformations. The reindeers are initially randomly scattered throughout the green field area with the help of the `initialise_deer_positions` method. A similar function exists for other models in the scene. All models were formatted as *Collada (.dae)* files before being imported into the project. Some of these models were downloaded from the internet, (reindeer and car),

whereas the rest were handmade, (‘road-field’ and fence). A Computer-Aided Design (CAD) software was utilised to create these handmade models as shown in the pictures below. (Note that the fence model ended up not being used in the final scene).



Handmade model of road and field area



Handmade model of fence, (not used in final scene)

Relevant Code Snippets:

```
typedef struct ModelData
{
  size_t mPointCount = 0;
  std::vector<vec3> mVertices;
  std::vector<vec3> mNormals;
  std::vector<vec2> mTextureCoords;
  std::vector<int> vert_indices;
} ModelData;
```

ModelData Class

```
class ModelMesh {
public:
    ModelData mesh_data;
    float* vertices;
    bool loaded;
    ModelMesh(const char* file_name);
};
ModelMesh::ModelMesh(const char* file_name) {
    vertices = NULL;
    loaded = false;
    if (!loaded) {
        mesh_data = load_mesh(file_name, &vertices);
        loaded = true;
    }
}
```

ModelMesh Class

```
class Model {
public:
    ModelMesh* modelMesh;
    GLfloat rotate_x;
    GLfloat rotate_y;
    GLfloat rotate_z;
    GLfloat translation_x;
    GLfloat translation_y;
    GLfloat translation_z;
    GLfloat theta;
    glm::vec3 translation;
    Model(ModelMesh* modelMesh);
};
Model::Model(ModelMesh* m) {
    rotate_x = 0.0f;
    rotate_y = 0.0f;
    rotate_z = 0.0f;
    translation_x = 10.0f;
    translation_y = 25.0f;
    translation_z = 30.0f;
    translation = glm::vec3(translation_x, translation_y, translation_z);
    theta = 0.0f;
    modelMesh = m;
}
```

Model class

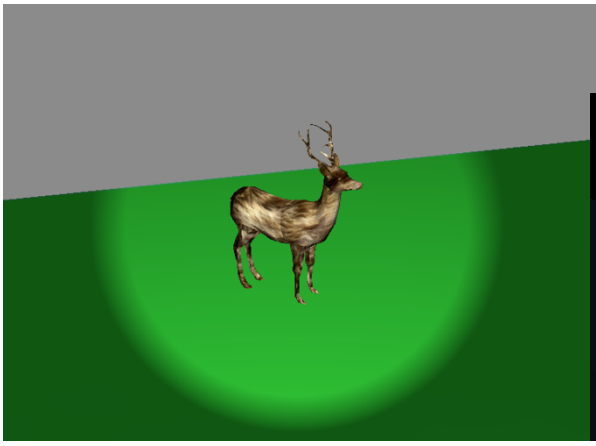
```
void initialise_deer_positions(std::vector<Model>& deer) {
    for (int i = 0; i < deer.size(); i++) {
        int r_tx = rand();
        int r_tz = rand();
        int r_ry = rand();
        deer[i].translation_x = ((3.0f + fmodf(r_tx, 19.0f)) / adjustment);
        deer[i].translation_y = (25.0f / adjustment);
        deer[i].translation_z = ((28.0f + fmodf(r_tz, 19.0f)) / adjustment);
        deer[i].rotate_x = 0.0f;
        deer[i].rotate_y = fmodf(r_ry, 360.0f);
        deer[i].rotate_z = 0.0f;
    }
}
```

initialise_deer_positions function

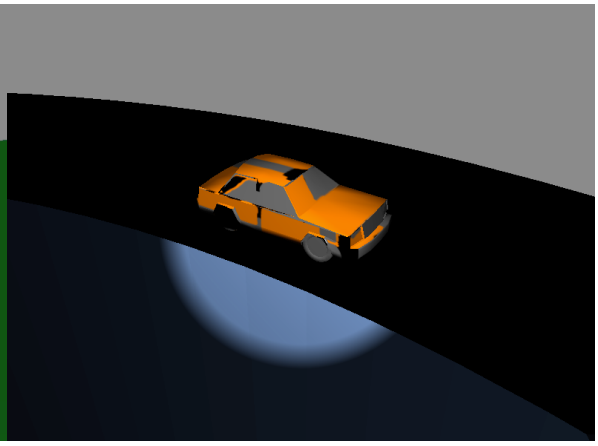
Credits:
Software used for modelling: *Blender, FreeCAD*
Reindeer and car models downloaded from: *free3d.com*
Shaders / shader class used (except in preliminary scene) from: *learnopengl.com*

Required Feature 2 (Texture mapping using image files):

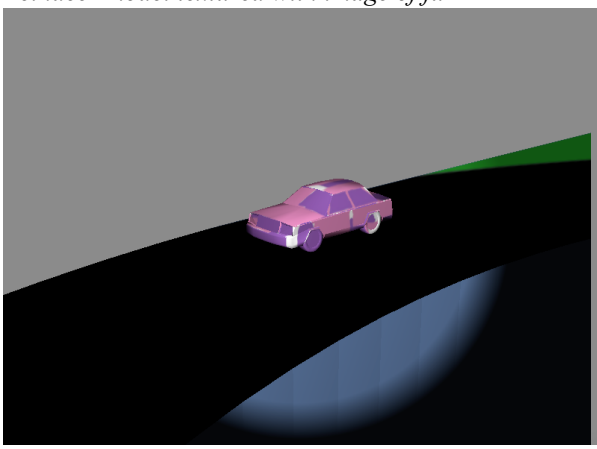
Screenshot(s) of feature:



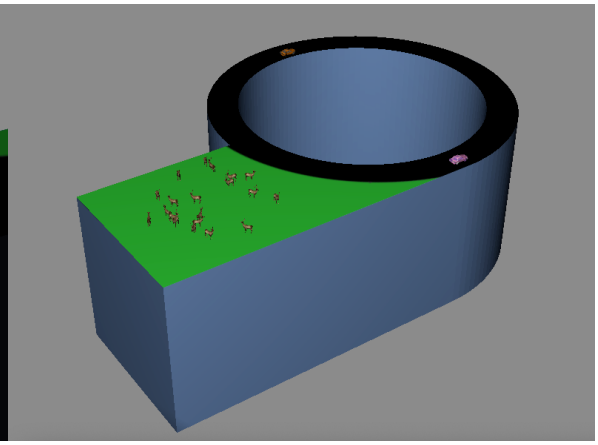
Reindeer model textured with image of fur



Car model textured with handmade texture



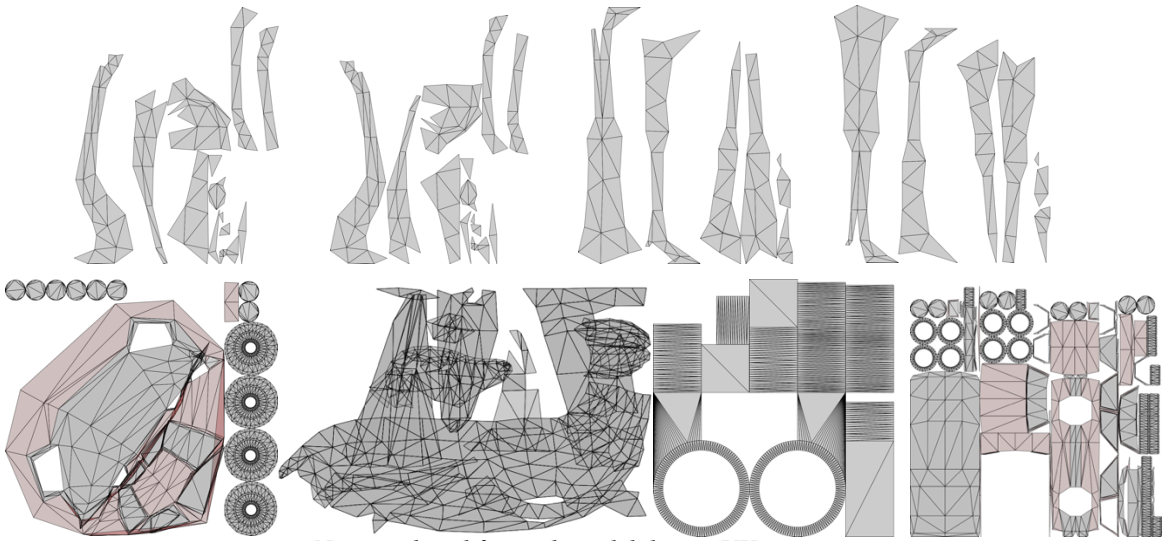
Car model textured with different handmade texture



Scene model 'road-field' textured with handmade texture

Description of implementation:

In order to map a 2D image onto a 3D model, there are a number of steps which must be followed. The model must be unwrapped and projected onto a 2D plane as a net. There are a many algorithms for accomplishing this, depending on what needs to be optimised. This step was done for each model with the aid of UV Editing functionality in Blender. The UV are texture coordinates and are stored at each vertex in a mesh. It is based on the unwrapped model, or net, that these coordinates are determined. The values for these coordinates range from 0 to 1. Blender also allows for the superimposition of an image file on top of the generated net. It is then more easy to adjust the coordinates to improve the quality of the eventual rendered model. The pictures below depict the nets (texture coordinates) generated for each model.



Nets produced for each model during UV mapping

It is also possible to handmake an image for texture mapping by using the net as a template. This process was employed for the car and 'road-field' (scene) models. However, it proved to be quite difficult to achieve the desired effects in these cases. The images below correspond to these handmade images produced by this process.

Handmade images created for each cars and 'road-field' (scene)

Once the mesh has texture coordinates at its vertices, they must be extracted within the program. This is done during the `load_mesh` function. Specifically, as shown in the code below, the program must appropriately skip the other information stored at vertices when extracting texture coordinates.

Relevant Code Snippets:

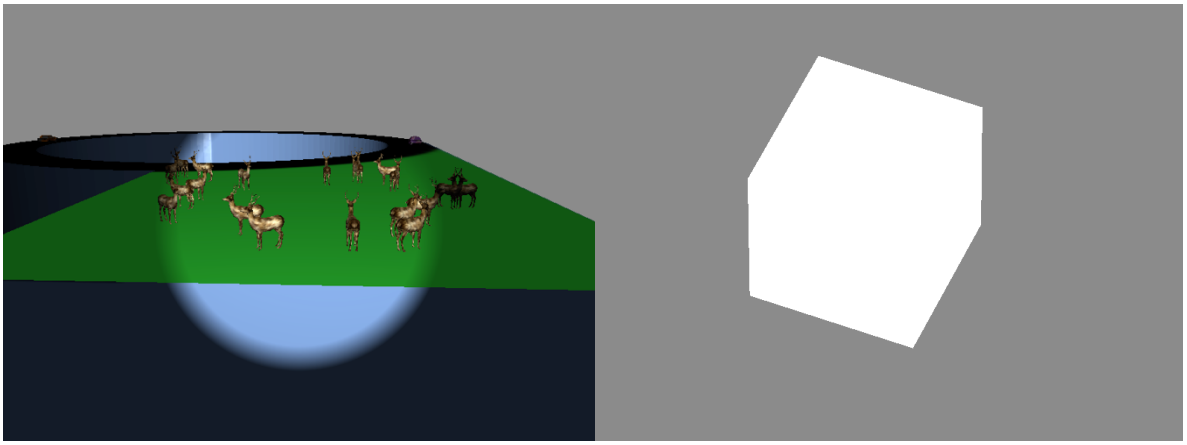
```
float* vertices = new float[modelData.mVertices.size()*(3+3+2)];
int j=0;
for (int i=0; i < modelData.mVertices.size(); i++) {
  vertices[j] = modelData.mVertices[i][0]/200;
  vertices[j+1] = modelData.mVertices[i][1]/200;
  vertices[j+2] = modelData.mVertices[i][2]/200;
  vertices[j+3] = modelData.mNormals[i][0];
  vertices[j+4] = modelData.mNormals[i][1];
  vertices[j+5] = modelData.mNormals[i][2];
  vertices[j+6] = modelData.mTextureCoords[i][0];
  vertices[j+7] = modelData.mTextureCoords[i][1];
  j += 8;
}
j=0;
```

Texture coordinates extraction at vertices

Credits:
`load_mesh` function based on `load_mesh` function supplied in Lab04

Required Feature 3 (Lighting implemented with the Phong illumination model):

Screenshot(s) of feature:



Spotlight *Point light*

Description of implementation:

The lighting was implemented through the use of a specific shader which calculates the sum of the contributions of each type of light source in the scene. Subsequently, objects rendered with this shader are appropriately illuminated based on any lights in the scene.

There is a small amount of ambient lighting in the scene as well as multiple point light sources. The point light sources take the form of cubes and are eventually used as headlights for the car, (see below). There is also a spotlight facing in the same direction as the camera.

Relevant Code Snippets:

```
lightingShader.setVec3("spotLight.position", camera.Position);
    lightingShader.setVec3("spotLight.direction", camera.Front);
    lightingShader.setVec3("spotLight.ambient", 0.0f, 0.0f, 0.0f);
    lightingShader.setVec3("spotLight.diffuse", 1.0f, 1.0f, 1.0f);
    lightingShader.setVec3("spotLight.specular", 1.0f, 1.0f, 1.0f);
    lightingShader.setFloat("spotLight.constant", 1.0f);
    lightingShader.setFloat("spotLight.linear", 0.09f);
    lightingShader.setFloat("spotLight.quadratic", 0.032f);
    lightingShader.setFloat("spotLight.cutOff", glm::cos(glm::radians(12.5f)));
    lightingShader.setFloat("spotLight.outerCutOff", glm::cos(glm::radians(15.0f)));
```

Spotlight code – setting shader

```
lightingShader.setVec3("pointLights[0].position", pointLightPositions[0]);
lightingShader.setVec3("pointLights[0].ambient", 0.05f, 0.05f, 0.05f);
lightingShader.setVec3("pointLights[0].diffuse", 0.8f, 0.8f, 0.8f);
lightingShader.setVec3("pointLights[0].specular", 1.0f, 1.0f, 1.0f);
lightingShader.setFloat("pointLights[0].constant", 1.0f);
lightingShader.setFloat("pointLights[0].linear", 0.09f);
lightingShader.setFloat("pointLights[0].quadratic", 0.032f);
```

Point light code example – setting shader

Credits:
As mentioned above, shaders used (except in preliminary scene) from: *learnopengl.com*

Advanced Feature 1 (Hierarchical crowd):

Description/name of feature:
The reindeers and their legs are in a hierarchy

Advanced Feature 2 (Moving light sources):

Description/name of feature:
The headlights of the cars are light sources and are moving around the track.

Advanced Feature 3 (Crowd features):

Description/name of feature:
Reindeers are randomly scattered and avoid falling off the edge.