DECLARATION: I understand that this is an **individual** assessment and that collaboration is not permitted. I have read, understand and agree to abide by the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar. I understand that by returning this declaration with my work, I am agreeing with the above statement.

1 Implementations

This section will discuss my implementations of hierarchy, camera controls, and other relevant aspects for this deliverable. The reiterate my project title is: "Manta Ray Migration - Simulate a large school of manta rays gliding through a dynamic ocean, interacting with smaller fish".

1.1 Basic, Scene Setup and Object Loading

In my scene there are Manta Rays, small fish, starfish, seaweed, and textured environments with movements using Boids [1]. The video attached to the deliverable shows a video demonstration of the scene loading and running. The written explanation, is I have functions which wrap opengl calls, in object_create, object_draw, and object_destroy, which have a dependency on shaders. Shaders are attached during initialisation at the beginning and in a loop is drawn with the models. In certain cases, a generic model structure is not used, and instead structures like "Mantaray", and "Fish", exist which in practice extends the functionality of what a model is. Below is an example of my Model struct followed by Mantaray struct:

```
typedef struct {
    vec3 position;
    vec3 normal;
    vec2 texCoords;
} Vertex;

typedef struct {
    unsigned int id;
    char* type;
    char path[255];
} Texture;

typedef struct {
    Vertex* vertices;
    unsigned int* indices;
    Texture* textures;
    unsigned int VAO, VBO, EBO;
} Mesh;

typedef struct {
    Mesh* meshes;
    Texture* texturesLoaded;
    char directory[255];
    size_t directoryLength;
} Model;
```

Figure 1: Structure definition of Model

```
typedef struct {
    Model body;
    Model leftHandOne;
    Model rightHandOne;

    Model leftHandTwo;
    Model rightHandTwo;

    unsigned int bufferBody;
    unsigned int bufferLeftHand;
    unsigned int bufferRightHand;
} Mantaray;
```

Figure 2: Structure definition of Manta Ray

My Model struct is based of the material I referenecd for model loading from LearnOpenGL, [2]. My Manta Ray struct is defined in a specific way to allow for hierarchical animation. I split the model into multiple parts. I also create multiple buffers, to draw these models using instancing, a way of drawing multiple equivalent objects in one draw call.

1.2 Camera

My camera's implementation is based on euler angles, where, yaw, pitch, are used. Notice how roll isn't used, the camera wouldn't need to roll as I am mimicking a first person perspective. I have a mouse field which I set as well to move the camera. The structure definition of the camera can be seen below: An important part of my camera implementation, is that I had additional structure, called "Cameras". This allows me to pass in multiple Camera structures and in turn give my code the ability to change cameras with a click of a button. Additionally zoom functionlity exists, which increases and decreases the field of view using perspective.

```
typedef struct {
    vec3 pos;
    vec3 front;
    vec3 up;
    vec3 right;
    vec3 worldup;
    float speed;
    float zoom;
    Mouse mouse;
    EulerAngle angle;
    float deltaTime;
} Camera;

typedef struct {
    Camera* camera; // a list of cameras
    size_t focus;
    size_t size;
} Cameras;
```

Figure 3: Structure definition of Camera and Cameras

```
void cg_control_camera_create(Camera* camera, float speed, vec3 pos) {
    vec3 up = {0.0f, 1.0f, 0.0f};
    vec3 front = {0.0f, 0.0f, -1.0f};
    glm_vec3_copy(pos, camera->pos);
    glm_vec3_copy(front, camera->front);
    glm_vec3_copy(up, camera->worldup);
    camera->speed = speed;
    camera->zoom = 45.0f;
    camera->angle.pitch = 0.0f;
    camera->angle.yaw = -90.0f;
    camera->mouse.xpos = CG_SCREEN_X / 2.0;
    camera->mouse.ypos = CG_SCREEN_Y / 2.0;
    camera->mouse.focus = 0;
    camera->mouse.sensitivity = 0.1f;
    cg_control_angle_update(&camera->angle, camera->front, camera->worldup, camera->up, camera->right);
}
```

Figure 4: Creation of Camera

```
Camera camera[2]; // Two cameras
{
    vec3 cameraOne = {0.0f, 0.0f, 3.0f}; // In ocean
    cg_control_camera_create(&camera[0], 10.0f, cameraOne);

    vec3 cameraTwo = {0.0f, 90.0f, 0.0f}; // outside ocean
    cg_control_camera_create(&camera[1], 10.0f, cameraTwo);
}
camera[1].angle.pitch = -89.0f;
```

Figure 5: Structure initialisation of camera

```
void cg_control_angle_update(EulerAngle* angle, vec3 front, vec3 worldup, vec3 up, vec3 right) {
    front[0] = cos(glm_rad(angle->yaw)) * cos(glm_rad(angle->pitch));
    front[1] = sin(glm_rad(angle->pitch));
    front[2] = sin(glm_rad(angle->yaw)) * cos(glm_rad(angle->pitch));
    glm_normalize(front); // normalize front vector

    glm_vec3_cross(front, worldup, right); // cross product between the worlds up and right vector insert into right vector
    glm_normalize(right); // normalize the result

    glm_vec3_cross(right, front, up); // cross product between right and front, and that becomes up vector
    glm_normalize(up); // normalize the result
}
```

Figure 6: Structure initialisation of camera

```
static void mouse_callback(GLFWwindow* window, double xposIn, double yposIn) {
    Cameras* cameras = glfwGetWindowUserPointer(window); // extract cameras from GLFW
    Camera* camera = &cameras->camera[cameras->focus];

    float xpos = xposIn;
    float ypos = yposIn;
    if (!camera->mouse.focus) { // If the camera 1 is enabled
        camera->mouse.xpos = xpos;
        camera->mouse.ypos = ypos;
        camera->mouse.focus = 1;
    }

    // set sensitivity of camera
    float xoffset = xpos - camera->mouse.xpos;
    float yoffset = camera->mouse.ypos - ypos;
    camera->mouse.xpos = xpos;
    camera->mouse.ypos = ypos;

    xoffset *= camera->mouse.sensitivity;
    yoffset *= camera->mouse.sensitivity;

    // update yaw, pitch. roll is not needed for camera
    camera->angle.yaw += xoffset;
    camera->angle.pitch += yoffset;

    if (camera->angle.pitch > 89.0f) {
        camera->angle.pitch = 89.0f;
    }
    if (camera->angle.pitch < -89.0f) {
        camera->angle.pitch = -89.0f;
    }
    cg_control_angle_update(&camera->angle, camera->front, camera->worldup, camera->up, camera->right);
}
```

Figure 7: Mouse movement for camera

Camera switching is achieved by calling a GLFW function. This function allows the user to pass in a pointer to a GLFW window, this window will hold the cameras structure, and anywhere in the code where I pass my window, I can extract the cameras and modify the camera. This is important when we update the camera at each iteration of the render loop. I have a focus variable which determines which camera is on. Therefore in practice I have two cameras which I can switch, and the angles of the cameras stay consistent when rapidly switching. My cameras are set to be inside the "ocean", and outside, looking at the skybox.

1.3 Hierarchical Animation

Two of my models use hierarchical animation, the Manta Ray and the small fish. I will use Manta Ray as the example for my explanation. My Manta Ray, is split into separate models, this is to ensure I can rotate, and therefore animate the fish. The models are split into: left hand 1, left hand 2, body, right hand 1, right hand 2. I use hands as a generic term to refer to what makes them flapping motion. It is split into two, which I animate in a hiearchical way. My "left/right hand 1" rely on "body", I do this by multiplying the "left/right hand 1" by "body". I do the same for "left/right hand 2" which multiplies by "left/right hand 1". Below is a code snippet which I have also commented. It goes through the hierarchical animation and the movement of the mantaray for it's left hands. The same approach is used for the right hands, but it is not shown here for the sake of brevity.

```
for (size_t i = 0, j = 0; i < boids->size; i+=2, j++) {
    glm_mat4_identity(modelBody[j]); // identity matrix

    glm_translate(modelBody[j], boids->boid[i].position); // translate boid to specific position

    mat4 rotation; // rotation matrix
    cg_boid_angle_update(&boids->boid[i], rotation, forward); // make boid face forward
    glm_mat4_mul(modelBody[j], rotation, modelBody[j]); // rotation * body to rotate body

    float time = glfwGetTime(); // timer to flap the hands of the Manta Ray
    float degrees = 10.0f;
    float angle = degrees * sin(time + boids->boid[i].time); // sin wave to give a constant flapping

    glm_mat4_identity(modelLeftHandOne[j]); // identity matrix
    glm_mat4_mul(modelBody[j], modelLeftHandOne[j], modelLeftHandOne[j]); // hierarchical animation (body * left hand 1)
    glm_rotate(modelLeftHandOne[j], glm_rad(angle), (vec3) {0.0f, 0.0f, 1.0f});

    glm_mat4_identity(modelLeftHandTwo[j]); // identity
    glm_mat4_mul(modelLeftHandOne[j], modelLeftHandTwo[j], modelLeftHandTwo[j]); // hiearchical animation (left hand 1 * left hand 2)
    glm_rotate(modelLeftHandTwo[j], glm_rad(angle / 10), (vec3) {0.0f, 0.0f, 1.0f});
```

Figure 8: Mantaray Hierarchical animation Code

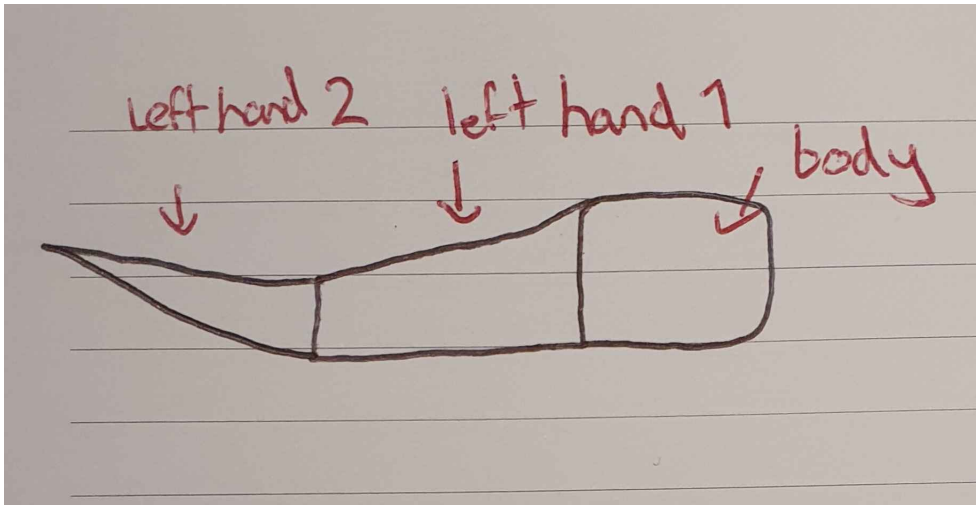

Figure 9: Mantaray Hierarchical structure

1.3.1 Other Animation aspects

Animations outside of hierarchical were also used. Quaternions using builtin cglm linear algebra functions [3], were used to rotate my models. My Models "swim" based on the boids algorithm [1]. Whenever a wall is hit, the Manta Ray and small fish would rotate. I used SLERP, to spherically linearly interpolate the positions and give a smooth rotation of my models. I use a lot of the builtin cglm functions given to me to achieve this. A visual demonstration might make sense, so reference the video for how the animation behaves.

```c
void cg_boid_angle_update(Boid* b, mat4 model, vec3 forward) {
    vec3 normVelocity;
    glm_vec3_normalize_to(b->velocity, normVelocity);

    vec3 rotation;
    glm_vec3_cross(forward, normVelocity, rotation);
    glm_vec3_normalize(rotation);

    float dot = glm_vec3_dot(forward, normVelocity);
    float angle = acos(dot);

    versor quaternion; // unit vector of norm one, x, y, z, w (unlike glm)
    glm_quatv(quaternion, angle, rotation);

    float slerp = 0.01f;
    versor interpolation;
    glm_quat_slerp(b->orientation, quaternion, slerp, interpolation);

    glm_quat_mat4(interpolation, model);
    glm_quat_copy(interpolation, b->orientation);
}
```

Figure 10: Spherical lerp animation

References

[1] Boids: Artifical life Simulation *https://en.wikipedia.org/wiki/Craig_Reynolds_(computer_graphics) Craig Reynolds*

[2] LearnOpenGL: Modern Opengl references *https://learnopengl.com/ JoeyDeVris*

[3] cglm: an optimized 3D math library written in C99 (compatible with C89). It is similar to the original glm library, except cglm is mainly for C. *https://cglm.readthedocs.io/en/latest/ recp*