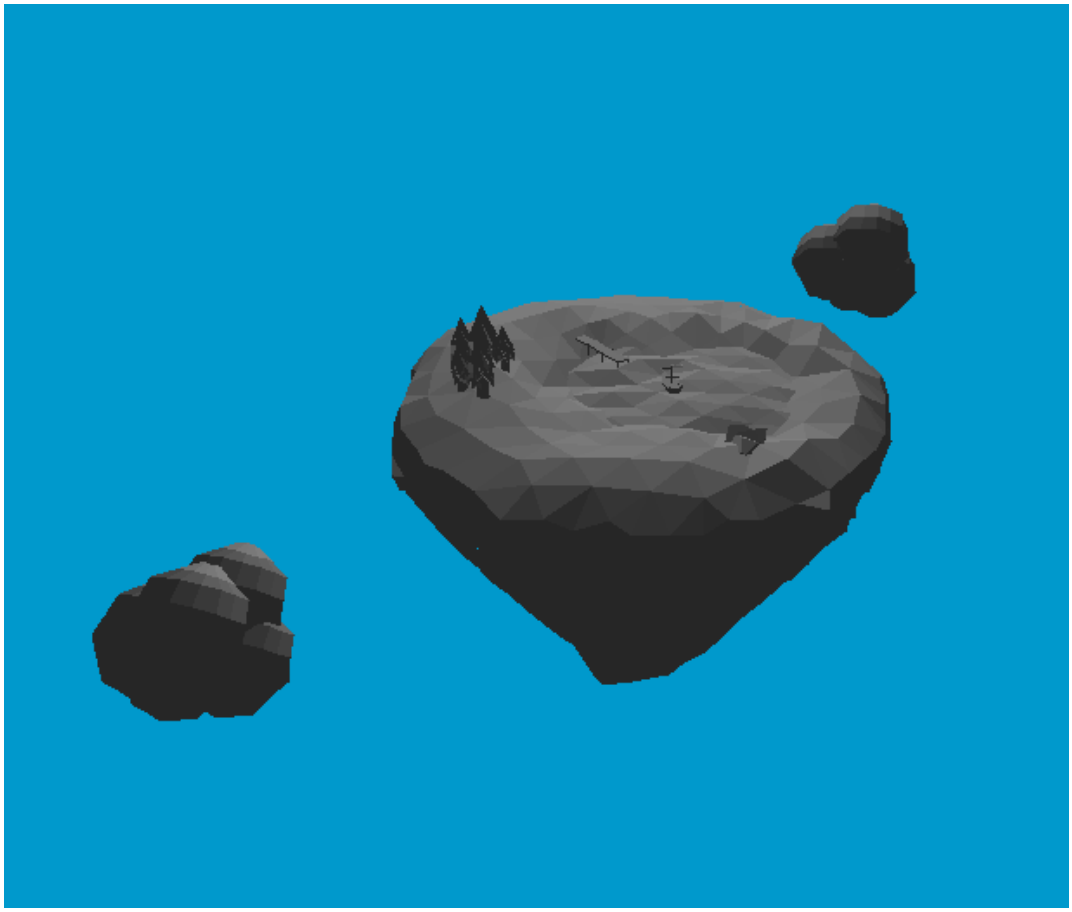
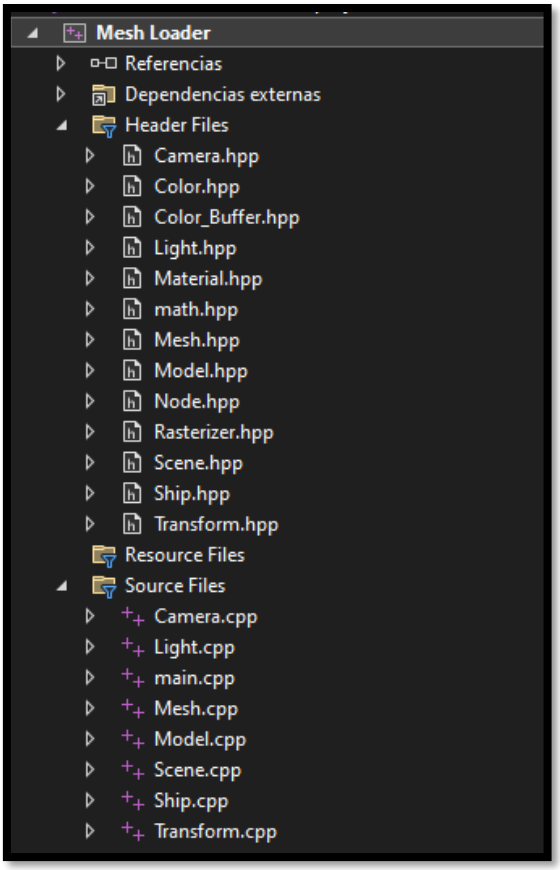
PG – Escenario Documentación – Práctica 1





Para esta práctica se ha optado por una composición de escena y nodos.

La escena es la encargada de guardar una lista de nodos, recoger el input principal de la ventana, actualizar los nodos y llamar a sus respectivos métodos de renderizado. Cada nodo se guarda en la escena con un identificador único. Contiene también la lógica necesaria para cargar y borrar el buffer de pantalla en cada Update.

Los nodos se dividen en tres componentes hijos:

- **Modelo**: Los modelos con composiciones de mallas.
- **Cámara**: La cámara especifica el punto desde el cual se ha de renderizar la escena y establece algunos parámetros de visualización como FOV, Near Plane o Far Plane.
- **Luz**: La luz/ces se utilizan para iluminar la escena en base a la distancia entre estas y los modelos. Se utiliza el método de iluminación difusa de Lambert.

Todos los nodos contienen una referencia a la **escena que los guarda** y un **Transform**. Un transform guarda los datos de posición, rotación y escala. Se encarga de gestionar y actualizar la matriz de transformación del modelo y en caso de que hubiera de calcularlo en base a su padre, estableciendo movimiento jerárquicos en la escena.

Los nodos contienen dos funciones que pueden heredar Update y render.

- **Update**: Se utiliza para actualizar la posición de los nodos en pantalla. Si quisiéramos mover un modelo o la cámara su input se recoge aquí y se actualiza su transform aquí.
- **Render**: Se utiliza para poder dar pie a renderizar los objetos que necesitaran de ello, por ejemplo las mallas de los modelos. Se pasan como parámetros en esta función ciertas matrices importantes como la de visión o la de proyección al igual que la fuente de luz para poder calcular la iluminación.

Las mallas son objetos que no heredan de nodo. Contienen una serie de parámetros que son necesarios para cargar y calcular sus transformaciones.

```
vector<Point4f>     original_normals;       ///< Original normals of the given mesh.
Vertex_Buffer       original_vertices;      ///< Original vertices of the given mesh.
Index_Buffer        original_indices;       ///< Original indices of the given mesh.
Vertex_Colors       original_colors;        ///< Original colors of the given mesh.
Vertex_Colors       transformed_colors;     ///< New colors of the mesh based with lightning operations applied.
Vertex_Buffer       transformed_vertices;   ///< New vertices positions in projection coordinates.
vector<Point4i>     display_vertices;       ///< New vertices positions in display coordinates.

Matrix44 render_transformation; ///< Display transformation matrix.
bool render_matrix_calculated; ///< Flag indicating whether render matrix is calculated so we only have to calculate it once.
```

Los modelos se encargan de cargar un archivo desde memoria usando la librería Assimp y crear tantas mallas como contenga el modelo. Luego a cada malla se le pasa una serie de parámetros para que pueda cargar sus respectivos buffers.

Las mallas contienen también un método render que les permite hacer en ese momento sus transformaciones desde coord. locales a coord. de display, usando las matrices comentadas previamente.

```
void render(Rasterizer<Color_Buffer>& rasterizer, const Matrix44& transform_matrix, const Matrix44& model_view_matrix, Light& light_source);
```

Se necesita también un rasterizador, ya aportado para esta práctica, para poder dibujar triángulos y asignarles un color a los vértices.

Los cálculos se luz se hacen usando las normales de los vértices y una función que contiene la clase de luz. Esta función aplica el método de Lambert para devolver una intensidad en base al ángulo entre la luz y la normal sumando algunos parámetros de iluminación.

```
//LAMBERT MODEL  L ^ N

Vector3f& l = glm::normalize(transform->get_position() - point);

float dot_product = glm::dot(l, normal);

float total_intensity = ambient_intensity + (intensity * dot_product);

return glm::clamp(total_intensity, ambient_intensity, 1.f);
```

Para el recorte simple de triángulos de la escena como tal no es recorte si no descarte. Se emplea una función que determina si los vértices están o no dentro de la ventana, si

no lo están no se dibuja todo el triángulo. Es algo más temporal realmente para poder moverse por la escena sin que rompiera el programa constantemente. Lo ideal sería tratar de añadir el algoritmo de recorte de Sutherland-Hodgman.

Se enseñan por escena también algunos objetos en movimiento como es el barco del centro del lago. Este barco hereda de la clase modelo y crea su propio Update para poder gestionar su rotación por separado. Hace un movimiento de subida y bajada simple y un poco de rotación.

Para los Assets visuales se usaron modelos cogidos de la página web de sketchfab. En concreto este paquete de "CatgirlNotLive":

https://sketchfab.com/3d-models/low-poly-floating-island-7a35716ec71b4bf8aadf40cd139e1902

Se separaron los Assets individualmente en Blender en 3 componentes, la isla principal, el barco y la nube. La nube se replica dos veces y todos son hijos de la isla principal.

Para los cálculos matemáticos se usó la librería de GLM.

Existen algunos problemas de visualización como los pequeños espacios que aparecen entre los triángulos al rotarse. La cámara también tiene algún bug visual que ocurre cuando te adentras mucho en un modelo haciendo zoom que consigues como darle la vuelta a la imagen con algún efecto raro del z-buffer supongo.