PROYECTO
CASA VACACIONAL CUAUTLA

Profesor: Ing. Carlos Aldair Román Balbuena

317032392 Martínez Martínez Francisco David

Grupo: 6
Clave: 6590

DESCRIPCIÓN DEL PROYECTO Y MANUALES TÉCNICOS

REQUERIMIENTOS DEL PROYECTO
CASA VACACIONAL CUAUTLA

1.1 PROPÓSITO DEL PROYECTO

El objetivo principal de este proyecto es combinar los diversos elementos de aprendizaje de la asignatura "CGEIHC" en un proyecto general, funcional e implementable. Y no sólo se elabora para demostrar los conocimientos adquiridos en el curso, sino también poner en práctica de manera efectiva las habilidades básicas, tanto prácticas como teóricas.

1.2 OBJETIVOS DEL PROYECTO

Los objetivos del escrito son los siguientes:

- Elección de la fachada y el espacio: los estudiantes deben elegir una fachada y un espacio reales o ficticios como base para su proyecto. Esto incluye determinar el entorno para la representación 3D en OpenGL.

- Proporcione imágenes de referencia: los estudiantes deben buscar y proporcionar imágenes de referencia de la habitación seleccionada. Estas imágenes sirven como guías para la replicación en 3D y deben reflejar fielmente los aspectos visuales y estéticos del espacio.

- Replicación 3D en OpenGL: El objetivo principal es recrear virtualmente el espacio seleccionado utilizando el lenguaje de programación OpenGL. Se desarrolla un modelo tridimensional que representa con precisión la fachada y las habitaciones seleccionadas.

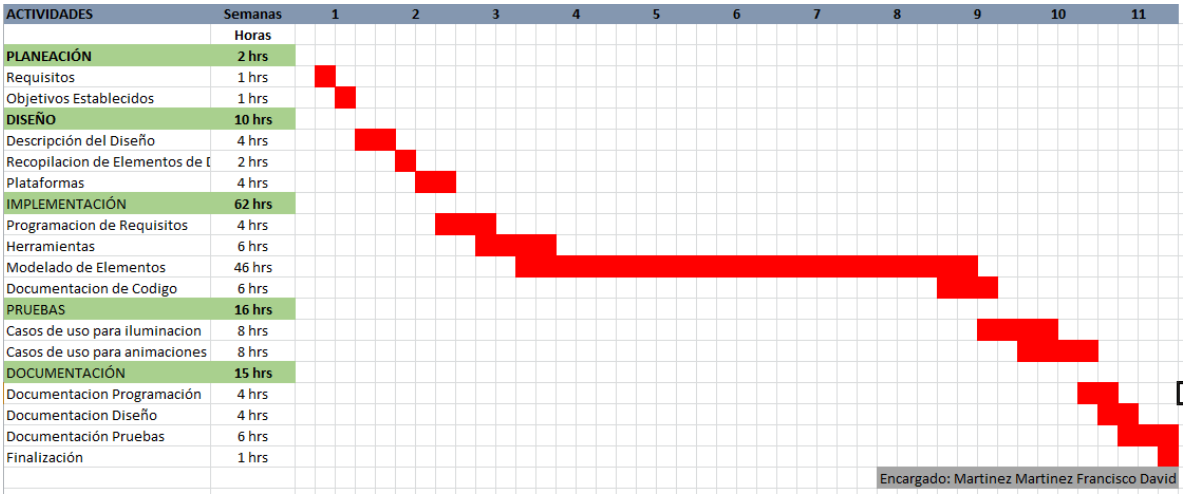
- Visualización de 7 Objetos: Dentro de la réplica 3D debe haber 7 objetos específicos que los estudiantes replicarán virtualmente. Estos objetos deben parecerse lo más posible a la imagen de referencia tanto en apariencia como en detalle.

- Diseño de habitaciones: además de recrear objetos, los estudiantes también deben trabajar en el diseño de habitaciones. Es importante crear la atmósfera adecuada teniendo en cuenta la iluminación, los colores, los materiales y otros factores que contribuyen a una representación realista del espacio elegido.

1.3 RESTRICCIONES

El proyecto se debe entregar de forma individual, con un manual de usuario en donde se explique cada interacción dentro del ambiente virtual recreado, así como un manual técnico que contenga la documentación del proyecto, que incluya objetivos, diagrama de flujo del software, diagrama de Gantt, alcance del proyecto, limitantes, metodología de software, la documentación del código, (no solo es copiar y pegar código y comentar algunas líneas de él) y conclusiones.

1.4 DIAGRAMA DE GANTT



1.5 COSTOS

CATEGORÍA	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	**TOTAL**
Sueldos	$1500.00	$1500.00	$1000.00	$1000.00	$1500.00	$1500.00	$1500.00	$1000.00	$1000.00	$1000.00	$1000.00	$13500.00
Plan de Propuestas	$2000.00											$2000.00
Gastos de Desarrollo		$1000.00	$1000.00	$1000.00	$1000.00	$1000.00	$1000.00	$1000.00	$1000.00	$1000.00	$1000.00	$10000.00
Administración de Herramientas			$1000.00	$1000.00	$1000.00							$3000.00
Material y Mantenimiento de Equipos						$1500.00				$1500.00	0	$3000.00
Desarrollo de entregables	$500.00				$500.00				$500.00	$500.00	$1500.00	$3500.00
										COSTO TOTAL		**$35000.00**

1.6 PROPUESTA PARA EL PROYECTO

Fachada seleccionada

Cuarto seleccionado

CUARTO A RECREAR:

El cuarto seleccionado para este trabajo sería una Sala que se encuentra ubicada del lado derecho de la fachada de nuestra casa vacacional, en este cuarto los objetos que principalmente lo conforman es un juego de Sillones Artesanales, Una Mecedora, Cuadros de decoración y también cuenta con una Pantalla plana junto con su mueble.

FACHADA:

La Fachada de la casa seleccionada para elaborar, consta de dos pisos, en el primer piso se encuentra el cuarto seleccionado mencionado en el párrafo anterior, que se encuentra del lado derecho de la puerta de entrada

OBJETOS

- Sillones y Mecedora Artesanales:

  Parte fundamental del cuarto que abarca un espacio considerable del escenario, y son piezas fundamentales para la recreación de la escena ya que se trata de transmitir la idea de comodidad.

  - Sillones Artesanales:

    - Individual: Se pueden encontrar como parte del juego de sala dos sillones artesanales individuales.

- ❖ Grupal: El objeto que más destaca por sus proporciones ya que este es el objeto más grande del juego de la sala y tal y como su nombre lo indica está pensado para ser usado por varias personas.

- ❖ Sillón Antiguo: Forma parte de la decoración del cuarto seleccionado.

- ➢ Mecedora Artesanal: Objeto seleccionado para animaciones simple, por su naturaleza de movimiento.

- Elementos de decoración

  Son utilizados para proporcionar una vista agradable a la estancia, colocados en la pared y evitando una sobrecarga de objetos. En nuestro proyecto consta de dos cuadros, mueble de televisión y pantalla.

  - ➢ Cuadros: elementos artísticos para dar una sensación de paz y tranquilidad.

  - ➢ Mueble de Televisión y Pantalla: Utilizados para la recreación y entretenimiento durante la estancia en nuestra Sala.

- Elementos de animación

  Son objetos en movimiento que le dan profundidad a la Casa Vacacional.

  - ➢ Dron: Objeto seleccionado para realizar una animación compleja y también para el uso de jerarquización.

  - ➢ Papalote: Objeto seleccionado utilizado para animaciones complejas con ayuda de funciones trigonométricas.

Casa Vacacional Cuautla

Manual Técnico Español

PLATAFORMAS DE TRABAJO.

Para el desarrollo de nuestro entorno de la casa vacacional, fue necesario trabajar con el entorno de desarrollo integrado de Microsoft Visual Studio (IDE), que se utiliza para crear programas, aplicaciones y servicios informáticos. En este caso, seguimos las operaciones de OpenGL, que es una API (interfaz de programación de aplicaciones) que nos ofrece un conjunto de funciones para manipular gráficos e imágenes. Aunque OpenGL es una especificación mantenida por el Grupo Khronos, no es una API en sí misma, sino simplemente una especificación.

La especificación de OpenGL define el resultado y el funcionamiento de cada función. Luego, los desarrolladores, como nosotros, debemos implementar estas especificaciones y encontrar la mejor solución para su funcionamiento. Dado que la especificación de OpenGL no proporciona detalles de implementación, las versiones desarrolladas de OpenGL pueden tener diferentes implementaciones, siempre y cuando los resultados cumplan con la especificación y sean consistentes para los usuarios.

Es importante destacar que para este proyecto de recreación de la casa vacacional, trabajamos con las bibliotecas de OpenGL escritas en C, las cuales permiten diversas derivaciones en otros lenguajes, aunque en esencia sigue siendo una biblioteca de C. Como muchas de las construcciones de lenguaje de C no se traducen directamente a otros lenguajes de nivel superior, OpenGL se desarrolló con varias abstracciones en mente. Una de esas abstracciones son los objetos en OpenGL.

Se definieron múltiples objetos para este proyecto de recreación de la casa vacacional, siguiendo las especificaciones de los requerimientos funcionales mencionados anteriormente. En OpenGL, un objeto es una colección de opciones que representa un subconjunto del estado de OpenGL, Donde podemos visualizar un objeto como una estructura en lenguaje C.

La ventaja de utilizar estos objetos es que podemos definir más de uno en nuestra aplicación, configurar sus opciones y, cada vez que realizamos una operación que utiliza el estado de OpenGL, vinculamos el objeto con nuestra configuración preferida. Por ejemplo, hay objetos que actúan como contenedores de datos para modelos 3D, como una casa o un personaje. Cada vez que queremos dibujar uno de estos modelos, vinculamos el objeto que contiene los datos correspondientes al modelo que deseamos dibujar. Esto nos permite especificar varios modelos y, cuando queremos

dibujar un modelo específico, simplemente vinculamos el objeto correspondiente sin tener que volver a configurar todas sus opciones.

PARÁMETROS BÁSICOS DEL ENTORNO

A) Uso de la Ventanas

Al adentrarnos en la creación del entorno gráfico para la casa de vacaciones, nos encontramos con la necesidad de configurar los parámetros básicos que nos permitieran establecer un entorno visual atractivo y funcional. Uno de los primeros aspectos que debimos abordar fue la creación de un contexto OpenGL y una ventana de aplicación adecuada para dibujar. Sin embargo, estas operaciones están estrechamente ligadas al sistema operativo y OpenGL, consciente de ello, busca abstraerse de dichas tareas y permitir a los desarrolladores centrarse en la implementación de los gráficos y las interacciones.

Para lograr este cometido, recurrimos a la biblioteca GLFW. GLFW, escrita en lenguaje C, se ha convertido en una opción popular y confiable para el desarrollo de aplicaciones gráficas en OpenGL. Esta biblioteca se enfoca específicamente en brindar las funcionalidades básicas necesarias para trabajar con OpenGL y facilitar la creación de ventanas, gestión de contextos y manejo de eventos de usuario. En nuestro caso, resultó ser una elección acertada para satisfacer nuestras necesidades en la creación del entorno visual para la casa de vacaciones.

Con la ayuda de GLFW, pudimos establecer un contexto OpenGL adecuado para aprovechar al máximo las capacidades gráficas del sistema. Además, tuvimos la posibilidad de definir diversos parámetros de la ventana, como el tamaño, la posición y las propiedades visuales, lo que nos permitió adaptar la presentación visual a nuestras preferencias y requisitos específicos. Asimismo, la biblioteca nos brindó la capacidad de manejar la entrada del usuario, incluyendo eventos de teclado y ratón, lo que resultó fundamental para interactuar con el entorno gráfico y proporcionar una experiencia inmersiva y amigable.

La elección de GLFW como biblioteca para el manejo de la ventana y el contexto OpenGL se basó en su reputación, documentación exhaustiva y su amplia adopción en la comunidad de desarrollo de gráficos por computadora. Su naturaleza de código abierto y su activa comunidad de usuarios contribuyeron a que encontráramos recursos y soluciones a los desafíos que surgieron durante el proceso de creación del entorno visual para la casa de vacaciones.

B) Vinculación

Para garantizar el correcto funcionamiento del proyecto con GLFW, fue necesario establecer la conexión entre la biblioteca y nuestro proyecto. Para lograr esto, se realizaron algunas configuraciones en el enlazador, especificando que se utilizaría el archivo glfw3.lib. Sin embargo, surgía un nuevo desafío, ya que nuestras bibliotecas de terceros se almacenaban en un directorio diferente al del proyecto principal. Por lo tanto, era imprescindible agregar este

directorio al proyecto para que este pudiera ubicar y acceder a glfw3.lib sin dificultades.

Además de incluir el directorio correspondiente, también debimos informar al entorno de desarrollo integrado (IDE) acerca de esta ubicación, para que pudiera realizar las búsquedas de bibliotecas y archivos de manera adecuada. De esta manera, se aseguraba una correcta vinculación y disponibilidad de las bibliotecas necesarias para el proyecto.

Nuestra lista de enlaces incluye los siguientes directorios:

- **$(SolutionDir)/External Libraries/GLEW/lib/Release/Win32**
- **$(SolutionDir)/External Libraries/GLFW/lib-vc2015**
- **$(SolutionDir)/External Libraries/SOIL2/lib**
- **$(SolutionDir)/External Libraries/assimp/lib**

Estos directorios indican las ubicaciones específicas donde se encuentran las bibliotecas relacionadas con GLEW, GLFW, SOIL2 y assimp respectivamente. Al añadirlos a la configuración del proyecto, nos aseguramos de que el compilador y el enlazador pudieran acceder a estas bibliotecas de manera correcta, facilitando el desarrollo del proyecto y garantizando su correcta ejecución.

C) Bibliotecas Utilizadas

Para facilitar la ubicación de las herramientas necesarias para el proyecto, se agregó manualmente la cadena de ubicación correspondiente. Esto permitió al IDE buscar en los directorios especificados cuando se necesitaban bibliotecas y archivos de encabezado. Al incluir las carpetas, como la de GLFW, se logró encontrar todos los archivos de encabezado relacionados con GLFW al utilizar <GLFW/..>, etc. Nuestra lista de bibliotecas incluye:

- **$(SolutionDir)/External Libraries/GLEW/include**
- **$(SolutionDir)/External Libraries/GLFW/include**
- **$(SolutionDir)/External Libraries/glm**
- **$(SolutionDir)/External Libraries/assimp/include**

Estos directorios especifican las ubicaciones donde se encuentran los archivos de encabezado necesarios para GLEW, GLFW, glm y assimp respectivamente. Al agregarlos a la configuración del proyecto, aseguramos que el compilador pudiera encontrar y utilizar correctamente estos archivos, lo cual resulta fundamental para el desarrollo del proyecto.

D) *Parámetros de Entrada*

Una vez que hemos establecido las cabeceras externas en nuestra IDE Visual Studio, se habilita la capacidad de localizar automáticamente todos los archivos necesarios para nuestro proyecto. Esto nos permite ahorrar tiempo y esfuerzo al evitar la búsqueda manual de cada archivo individual.

Después de configurar las cabeceras externas, el siguiente paso es vincular las bibliotecas requeridas para nuestro proyecto. Para hacer esto, nos dirigimos a la pestaña "Vinculador" y "Entrada" en la configuración del proyecto. En esta

sección, especificamos las bibliotecas que deseamos vincular.

En el caso específico mencionado, las bibliotecas que se vinculan son:

soil2-debug.lib, assimp-vc140-mt.lib, opengl32.lib, glew32.lib glfw3.lib.

Cada una de estas bibliotecas proporciona funcionalidades específicas y recursos necesarios para nuestro proyecto.

Al vincular estas bibliotecas, aseguramos que todas las dependencias requeridas estén disponibles para nuestro proyecto y que se puedan utilizar de manera efectiva durante la compilación y ejecución del mismo.

Este proceso de configuración y vinculación de bibliotecas es esencial para garantizar que nuestro proyecto tenga acceso a todas las herramientas y funcionalidades necesarias para su correcto funcionamiento. Una vez completada esta etapa, estamos listos para avanzar en el desarrollo de nuestro proyecto con la seguridad de que todas las bibliotecas están correctamente enlazadas y disponibles para su uso.

SHADERS

En el contexto de una casa vacacional, se ejecutan programas específicos para cada sección de la visualización gráfica. En esencia, los shaders son programas que transforman las entradas en salidas. Estos programas son independientes entre sí y no pueden comunicarse directamente; su única forma de interacción es a través de las entradas y salidas definidas.

En el caso de la casa vacacional, se utilizan shaders escritos en GLSL, un lenguaje similar a C diseñado específicamente para gráficos. GLSL ofrece características útiles para manipular vectores y matrices.

Cada shader comienza con una declaración de versión, seguida de una lista de variables de entrada y salida, así como de uniformes. Luego, se define la función principal del shader.

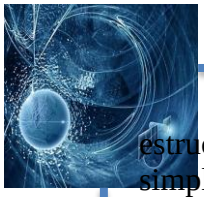
La función principal de cada shader es el punto de entrada, donde se procesan las variables de entrada y se generan los resultados en las variables de salida correspondientes.

En el proyecto de la casa vacacional, se emplearon shaders para cargar modelos, gestionar la iluminación, crear la ambientación (cubemaps) y realizar las animaciones. Estos shaders se encuentran en una carpeta específica llamada "Shaders" y su programación se ajusta a su función particular dentro del proyecto.

MODELOS

A) Biblioteca de Importación de Activos (Assimp)

Para importar modelos complejos a nuestra casa vacacional, utilizamos la biblioteca de importación de modelos Assimp. Esta biblioteca nos permite cargar diferentes formatos de archivo en una estructura de datos común, abstrayéndonos de los detalles técnicos de cada formato. Al cargar un modelo con Assimp, obtenemos acceso a todos los datos necesarios en un objeto de escena que contiene nodos que representan la



estructura del modelo. Esta solución simplifica la importación de modelos y nos permite trabajar con jerarquías y estructuras complejas de manera eficiente.

B) Representación de Mallas (Mesh)

En el contexto de nuestra casa vacacional, al cargar modelos con Assimp, obtenemos una gran cantidad de datos almacenados en las estructuras de Assimp. Sin embargo, para representar los objetos en OpenGL, necesitamos transformar esos datos a un formato comprensible por OpenGL. Una malla requiere al menos un conjunto de vértices, cada uno con información de posición, normal y coordenadas de textura. También incluye índices para el dibujo indexado y datos de materiales, como texturas difusas o especulares.
Mediante la construcción de la malla, obtenemos una lista de datos que se pueden utilizar para el renderizado. Configuramos los búferes adecuados y definimos el diseño de los atributos de vértice en el shader. Para completar la clase Mesh, implementamos la función "Draw", que se encarga de enlazar las texturas necesarias antes de llamar a glDrawElements. Sin embargo, esto puede resultar complicado, ya que no conocemos de antemano la cantidad o tipo de texturas que puede tener la malla.

C) Model Load

En esta etapa del desarrollo de nuestra casa vacacional, nos adentramos en la tarea de traducción y carga utilizando la biblioteca Assimp. Nuestro objetivo es crear una clase que represente un modelo completo, que incluya múltiples mallas y posiblemente varias texturas. Por ejemplo, podemos tener un modelo que contenga elementos como habitaciones, muebles y decoración, todos ellos cargados como parte de un único modelo.
Para lograr esto, aprovechamos Assimp para cargar el modelo y luego realizamos la traducción a varios objetos de malla que hemos creado previamente.

Es importante destacar que asumimos que las rutas de los archivos de textura mencionados en los archivos del modelo son locales al objeto del modelo en sí, es decir, se encuentran en el mismo directorio que el modelo.

Para obtener la ruta completa de las texturas, concatenamos la cadena de ubicación de la textura con la cadena del directorio que hemos obtenido previamente durante la función de carga del modelo. Esto nos permite tener acceso a las texturas adecuadas cuando sea necesario.

En algunos casos, los modelos obtenidos de fuentes externas pueden utilizar rutas absolutas para las ubicaciones de las texturas, lo cual no es compatible con nuestro enfoque actual. En estos casos, realizamos modificaciones manuales en los archivos para cambiar las rutas a rutas locales, asegurándonos de que las texturas se encuentren en el lugar correcto. También podemos realizar sustituciones de texturas y adaptarlas al modelo, según sea necesario.

ELABORACIÓN Y ADAPTACIÓN DE MODELOS

1. Modelos a utilizar

En su mayoría tanto los objetos como la fachada fueron creados y diseñados partiendo desde 0, los únicos modelos que fueron obtenidos de manera externa fueron el denominado Papalote y Televisor. Estos al ser de licencia gratuita no contenían texturas ni materiales.

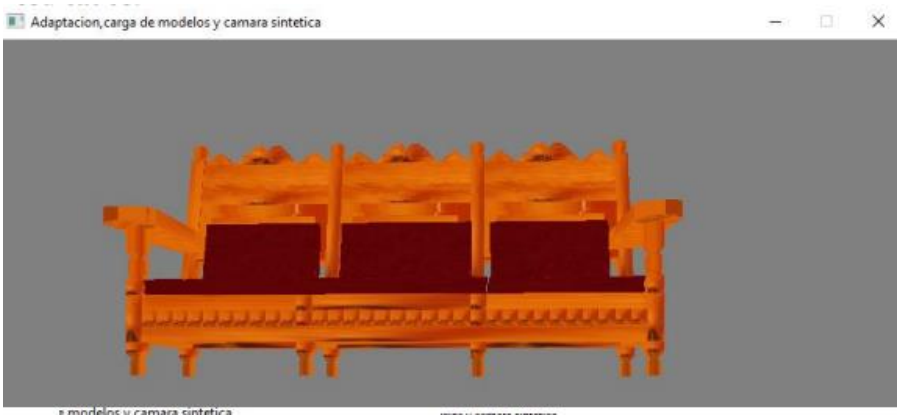


Figura 3 Diseño de Los Modelos 3D

2. Texturas y Materiales

Partiendo de los objetos previamente seleccionados para la fachada se buscó las mejores texturas para el diseño del proyecto. Una vez obtenidas las texturas se empezó a trabajar en la calidad de los materiales con ayuda de Maya y de GIMP, para poder realizarle acabados a las texturas para ser utilizadas en nuestros modelos y trabajarla en los mapas UV de nuestros objetos.

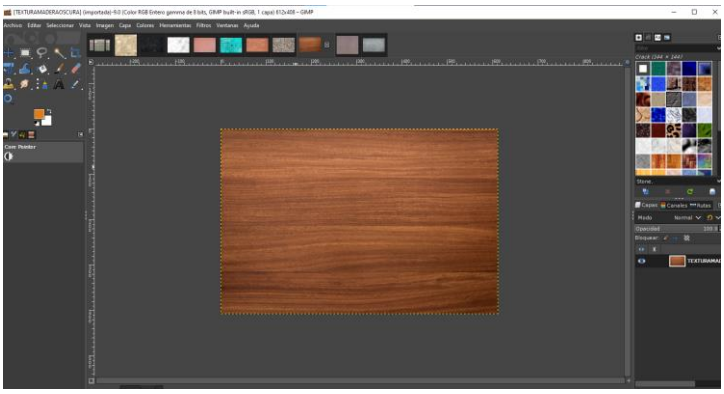


Figura 4 Diseño de las Texturas

3. *Finalización del modelado*

Una vez seleccionados y acoplados los materiales a utilizar, es de suma importancia corroborar las transformaciones básicas del objeto para así poder adecuarlos a su posición final en el entorno y de ser el caso trabajar con sus respectivos pivotes para la realización de interacciones, animaciones simples o complejas.

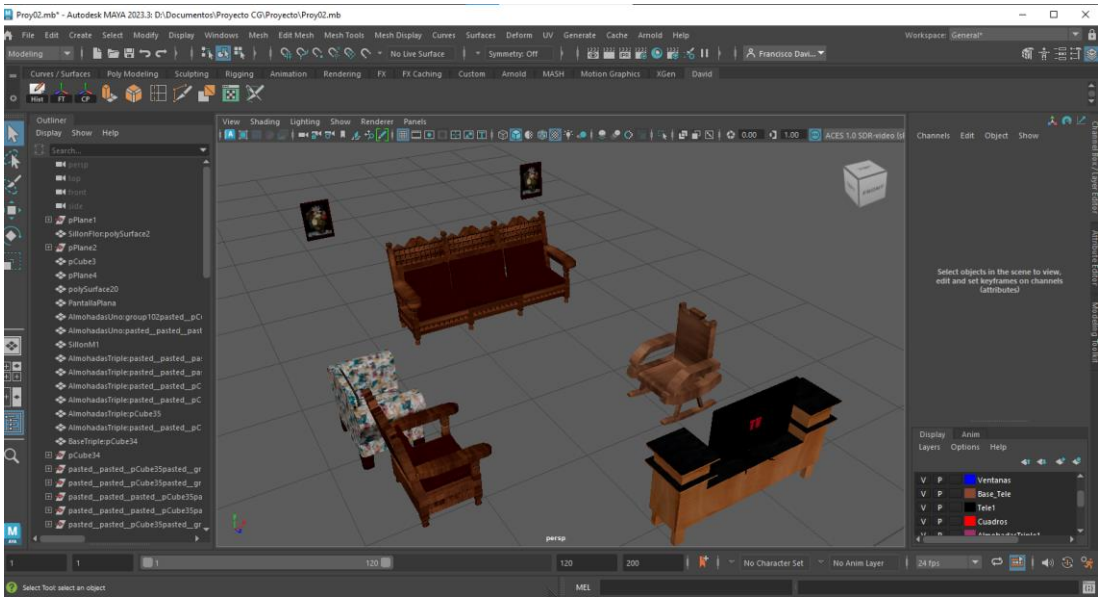


Figura 5

4. *Integración de los Modelos en el código del proyecto*

Una vez terminados los modelos de los objetos estos se cargan al proyecto para ser revisados (esto principalmente ya que están sujetos a cambios). Una vez cargados nos aseguramos que los objetos y sus transformaciones no afecten a la integridad del escenario.

Figura 6

5. Control de Versiones

Para asegurar la integridad del proyecto cada versión éxito generada a partir del acoplamiento de los modelos se sube al repositorio, esto con la finalidad de respaldar versiones funcionales.

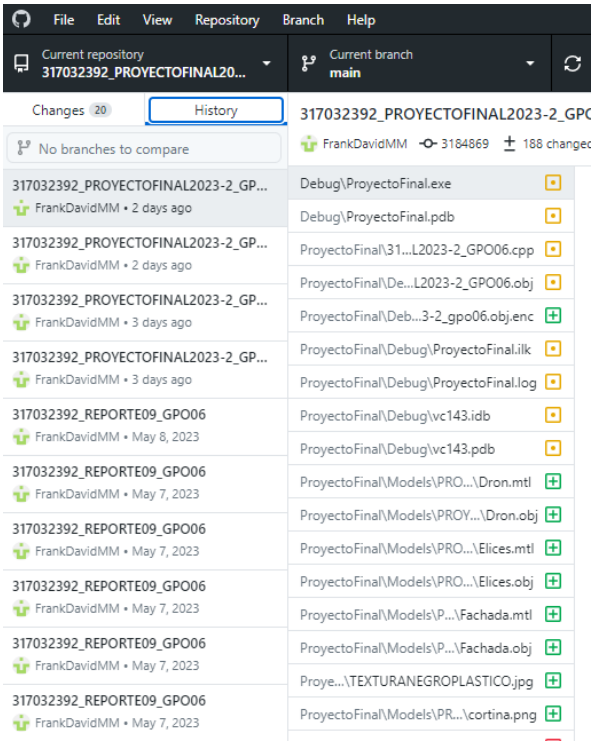


Figura 7

AMBIENTACIÓN E ILUMINACIÓN:

Para el desarrollo del escenario fue importante el uso de iluminación, las cuales fueron: directional, pointlight y spotlight. Por lo tanto en este apartado presentaremos las variables uniformes que utilizamos para estos tipos de iluminación: direccional, ambiental, difusa y especular, tal y como se muestra en las Figuras 7 y 8.

```
// Directional light

glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.direction"), -0.2f, -1.0f, -0.3f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.1f, 0.1f, 0.1f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.diffuse"), 0.1f, 0.1f, 0.1f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.specular"), 1.0f, 1.0f, 1.0f);



// Point light 1
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].position"), pointLightPositions[0].x, pointLightPositions[0].y, pointLightPositions
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].ambient"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].diffuse"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].specular"), 1.0f, 1.0f, 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].linear"), 0.009f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].quadratic"), 0.008f);



// Point light 2
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].position"), pointLightPositions[1].x, pointLightPositions[1].y, pointLightPositions
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].ambient"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].diffuse"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].specular"), 1.0f, 1.0f, 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[1].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[1].linear"), 0.009f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[1].quadratic"), 0.008f);


glm::vec3 lightColor;
lightColor.x = abs(Light1.x);
```

Figura 7

```
// Point light 3
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].position"), pointLightPositions[2].x, pointLightPositions[2].y, pointLightPositions
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].ambient"), lightColor.x, lightColor.y, lightColor.z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].diffuse"), lightColor.x, lightColor.y, lightColor.z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].specular"), 1.0f, 1.0f, 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[2].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[2].linear"), 4.4f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[2].quadratic"), 3.2f);

// Point light 4
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].position"), pointLightPositions[3].x, pointLightPositions[3].y, pointLightPositions
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].ambient"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].diffuse"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[3].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[3].linear"), 0.7f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[3].quadratic"), 0.8f);

// SpotLight
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.position"), camera.GetPosition().x, camera.GetPosition().y, camera.GetPosition().z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.direction"), camera.GetFront().x, camera.GetFront().y, camera.GetFront().z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.ambient"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.diffuse"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.linear"), 0.9f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.quadratic"), 0.32f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.cutOff"), glm::cos(glm::radians(12.5f)));
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.outerCutOff"), glm::cos(glm::radians(15.0f)));
```

Figura 8

La importancia de los valores uniformes que le mandemos a estas fuentes de iluminación recae en como el material de nuestros objetos reaccionará a estos cambios y que efecto tendrá en la vista final.

ANIMACIONES

Animaciones Simples:

La primera animación es la caída del cuadro, para lograrlo se utilizaron traslación y rotación, también se usó condicionales para verificar la posición del objeto, una vez que el cuadro se encontraba en el suelo este rotaria sobre el ejeX para estar totalmente en el piso y así simular su caída.

```
/*==================={ Simple Animations /Animaciones Simples }====================*/
model = glm::mat4(1);

model = glm::translate(model, glm::vec3(6.65f, 1.366f, -4.844f));
model = glm::translate(model, glm::vec3(0.0f, AnimCua01, 0.0f));
model = glm::rotate(model, glm::radians(rotCua01), glm::vec3(1.0f, 0.0f, 0.0));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
Cuadro01.Draw(lightingShader);
```

Figura 9 Carga del Modelo y sus transformaciones

```
/*============================================================================
* simple frame animation01 / Animacion01 Simple Picture
* ============================================================================ */

if (AnimCua01 >= -1.051f && rotCua01 == 0.0f) {

    AnimCua01 -= 0.05f;
}
else if (rotCua01  <= 85.0f ) {
    rotCua01 += 5.0f;
}
else if (ContAnims01 <= 100.0f ) {
    ContAnims01 += 0.5f;
}
else {
    AnimCua01 = 0.0f;
    rotCua01 = 0.0f;
    ContAnims01 = 0.0f;
}
```

Figura.10 Condicionales y variables responsables de la animación.

La segunda y tercera animación se caracterizan por realizar una rotación sobre el eje Z, en la primera la rotación tiene condicionales ´para poder controlar en

que posición acabara dándole realismo al movimiento como si de verdad un clavo se hubiera soltado de un lado y el cuadro se mantuviera con el otro.

En el caso de la mecedora se utilizaron igual condicionales, la diferencia es que la rotación de este objeto es algo más lineal ya que constantemente está decreciendo e incrementando el ángulo de la rotación, principalmente para no dejar fugas de memoria y también para dejar un movimiento más fluido.

```
/*=====================================================================
* simple frame animation02 / Animacion02 Simple Picture
* ===================================================================== */

if (rotCua02 >= -45.0f) {
    rotCua02 -= 2.5f;
}
else if(ContAnims02 <= 100.0f)
{
    ContAnims02 += 0.5f;
}
else {
    rotCua02 = 0.0f;
    ContAnims02 = 0.0f;
}
```

Figura 11 Manipulación de los valores encargados de la rotación del Cuadro02

```
22      }
23
24      /*=============================================================
25      * simple rocking animation03 / Animacion03 Simple Mecedora
26      * ============================================================= */
27      if (rotMece >= -5.0f && derMece == false) {
28          rotMece -= 0.2f;
29          if (rotMece < -5.0f) {
30              derMece = true;
31          }
32      }
33      else if(rotMece <= 5.0f && derMece == true){
34          rotMece += 0.2f;
35          if (rotMece > 5.0f) {
36              derMece = false;
37          }
38      }
```

Figura 12 Manipulación de los valores encargados de la rotación de la Mecedora.

Animaciones Complejas:

Vuelo del Dron:

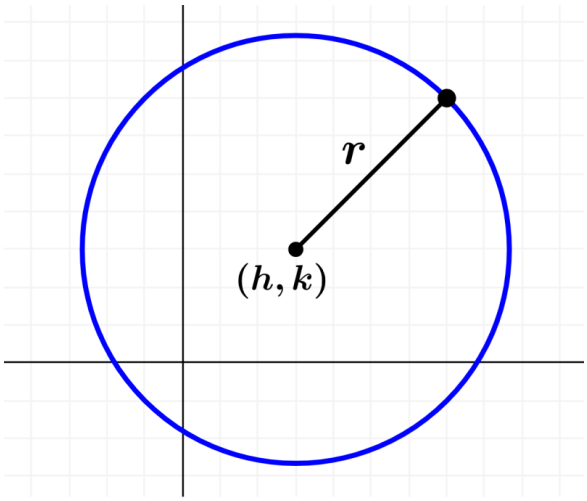
Para nuestra primera animación compleja se simula el vuelo de un Dron, para realizarla se utilizó jerarquización, esto ya que las hélices están separadas del objeto puesto que a cada una se le realiza una rotación para simular su funcionamiento, pero gracias a la jerarquización y respaldando la traslación del cuerpo del dron y posteriormente asignándoselo a las hélices estas siguen en todo momento la ubicación del cuerpo del dron. Lo anteriormente dicho es únicamente el modelado y acoplamiento de las partes del dron pero en si no la animación, la animación se caracteriza por utilizar la ecuación de la circunferencia con la finalidad de obtener las coordenadas en X y Z de la posición del dron para simular un círculo sin la necesidad de utilizar rotación. En resumen, para lograr la animación del Dron se utilizó jerarquización para poder modelar y mantener cada parte del mismo junta en cada momento y se utilizó la ecuación de la circunferencia para que este diera vueltas en un círculo a partir de los ejes X y Z sin el uso de rotación.

```
553
554         /* Hierarchy / Jerarquizacion */
555         Auxiliar = model = glm::translate(model, glm::vec3(PosicionX, 2.5f, PosicionZ));
556         Auxiliar = model = glm::rotate(model, glm::radians(rotDron), glm::vec3(0.0f, 1.0f, 0.0));
557
558         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
559         Dron.Draw(lightingShader);
560
561         model = glm::translate(Auxiliar, glm::vec3(0.066f, 0.015f, 0.052f));
562         model = glm::rotate(model, glm::radians(rotElices), glm::vec3(0.0f, 1.0f, 0.0));
563         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
564         Elices.Draw(lightingShader);
565
566         model = glm::translate(Auxiliar, glm::vec3(0.066f, 0.015f, -0.052f));
567         model = glm::rotate(model, glm::radians(rotElices), glm::vec3(0.0f, 1.0f, 0.0));
568
569         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
570         Elices.Draw(lightingShader);
571
572         model = glm::translate(Auxiliar, glm::vec3(-0.066f, 0.015f, -0.052f));
573         model = glm::rotate(model, glm::radians(rotElices), glm::vec3(0.0f, 1.0f, 0.0));
574
575         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
576         Elices.Draw(lightingShader);
577
578         model = glm::translate(Auxiliar, glm::vec3(-0.066f, 0.015f, 0.052f));
579         model = glm::rotate(model, glm::radians(rotElices), glm::vec3(0.0f, 1.0f, 0.0));
580
581         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
582         Elices.Draw(lightingShader);
583
```

Figura 13 jerarquización del Dron

```
875
876     /*=====================================================
877     * Drone animation / Animacion Dron
878     * =====================================================*/
879     if (AnguloDron <= 360.0f) {
880         AnguloDron += 0.04f;
881     }
882     else {
883         AnguloDron = 0.0f;
884         rotDron = 0.0f;
885     }
886
887
888     PosicionX = -2.9f + 1.1f * cos(AnguloDron);
889     PosicionZ =  1.5f + 1.1f * sin(AnguloDron);
890
```

Figura 14 Variables utilizadas para la animación.



$$(x-h)^2+(y-k)^2=r^2$$

Figura 15 Ecuación de la Circunferencia

Papalote:

Para la animación del papalote se utilizó funciones trigonométricas, exactamente la función Sen, ya que el recorrido que hace esta función es perfecta para realizar la animación, por lo tanto se agregaron valores extra a esta función para que la trayectoria recorrida por el mismo fuera la deseada.

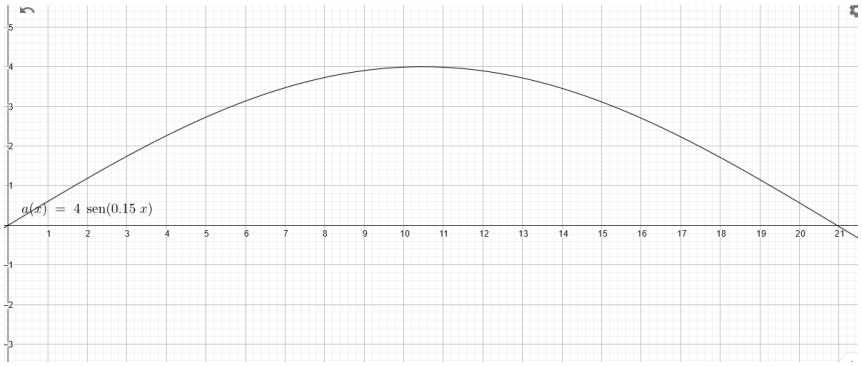


Figura 16. Ecuación y Recorrido deseado para nuestra animación

Conclusiones:

En resumen, la realización de un proyecto en OpenGL y modelado 3D implica comprender y aplicar una variedad de conceptos y procesos. Esto incluye la carga y manipulación de modelos utilizando bibliotecas como Assimp, la programación de shaders para efectos visuales y realismo, la configuración técnica de la aplicación, la gestión de la interfaz de usuario y la realización de pruebas exhaustivas. Además, es importante considerar la optimización del rendimiento para garantizar una experiencia fluida. En general, el desarrollo exitoso de un proyecto en OpenGL requiere conocimientos sólidos de gráficos 3D y habilidades técnicas para lograr resultados impactantes y de alta calidad visual.

Enlace:

https://github.com/FrankDavidMM/317032392_PROYECTOFINAL2023-2_GPO06.git

Casa Vacacional Cuautla

Manual Técnico Español

WORK PLATFORMS.

For the development of our vacation home environment, it was necessary to work with Microsoft Visual Studio Integrated Development Environment (IDE), which is used to create programs, applications, and computer services. In this case, we followed the operations of OpenGL, which is an Application Programming Interface (API) that provides a set of functions for manipulating graphics and images. Although OpenGL is a specification maintained by the Khronos Group, it is not an API itself, but rather just a specification.

The OpenGL specification defines the outcome and operation of each function. Then, developers like us must implement these specifications and find the best solution for their operation. Since the OpenGL specification does not provide implementation details, different developed versions of OpenGL can have different implementations, as long as the results comply with the specification and are consistent for users.

It is important to note that for this vacation home recreation project, we worked with OpenGL libraries written in C, which allow for various derivatives in other languages, although it essentially remains a C library. Since many of the language constructs in C do not directly translate to higher-level languages, OpenGL was developed with several abstractions in mind. One of these abstractions is objects in OpenGL.

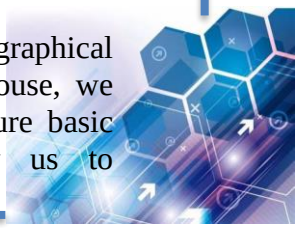
Multiple objects were defined for this vacation home recreation project, following the specifications of the mentioned functional requirements. In OpenGL, an object is a collection of options that represents a subset of OpenGL state, where we can view an object as a structure in the C language.

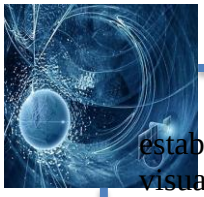
The advantage of using these objects is that we can define more than one in our application, configure their options, and whenever we perform an operation that uses the OpenGL state, we bind the object with our preferred configuration. For example, there are objects that act as data containers for 3D models, such as a house or a character. Whenever we want to draw one of these models, we bind the object that contains the corresponding data for the model we want to draw. This allows us to specify multiple models, and when we want to draw a specific model, we simply bind the corresponding object without having to reconfigure all its options.

BASIC ENVIRONMENT PARAMETERS

A) WINDOW USAGE

As we delved into creating the graphical environment for the vacation house, we encountered the need to configure basic parameters that would allow us to





establish an appealing and functional visual environment. One of the first aspects we had to address was the creation of an OpenGL context and a suitable application window for rendering. However, these operations are closely tied to the operating system, and OpenGL, being aware of this, seeks to abstract away from such tasks and enable developers to focus on graphics and interactions implementation.

To achieve this, we turned to the GLFW library. GLFW, written in the C language, has become a popular and reliable choice for developing graphics applications in OpenGL. This library specifically focuses on providing the essential functionalities needed to work with OpenGL, facilitating window creation, context management, and user event handling. In our case, it proved to be a fitting choice to meet our requirements in creating the visual environment for the vacation house.

With the help of GLFW, we were able to establish a proper OpenGL context to harness the full graphics capabilities of the system. Additionally, we had the flexibility to define various window parameters, such as size, position, and visual properties, allowing us to tailor the visual presentation to our preferences and specific requirements. Furthermore, the library provided us with the ability to handle user input, including keyboard and mouse events, which was crucial for interacting with the graphical environment and providing an immersive and user-friendly experience.

The selection of GLFW as the library for window and OpenGL context management was based on its reputation, comprehensive documentation, and widespread adoption within the computer graphics development community. Its open-source nature and active user community contributed to finding resources and solutions to the challenges that arose during the process of creating the visual environment for the vacation house.

B) Linking

To ensure the proper functioning of the project with GLFW, it was necessary to establish the connection between the library and our project. To achieve this, some configurations were made in the linker, specifying that the glfw3.lib file would be used. However, a new challenge arose as our third-party libraries were stored in a different directory from the main project. Therefore, it was essential to add this directory to the project so that it could locate and access glfw3.lib without difficulties.

In addition to including the corresponding directory, we also needed to inform the integrated development environment (IDE) about this location so that it could perform library and file searches properly. This ensured the correct linking and availability of the necessary libraries for the project.

Our list of links includes the following directories:

- **$(SolutionDir)/External Libraries/GLEW/lib/Release/Win32**
- **$(SolutionDir)/External Libraries/GLFW/lib-vc2015**
- **$(SolutionDir)/External Libraries/SOIL2/lib**
- **$(SolutionDir)/External Libraries/assimp/lib**

These directories indicate the specific locations where the libraries related to GLEW, GLFW, SOIL2, and assimp are located, respectively. By adding them to the project configuration, we ensured that the compiler and linker could access these libraries correctly, facilitating the development of the project and ensuring its proper execution.

C) Utilized Libraries

To facilitate the location of the necessary tools for the project, the corresponding location string was manually added. This allowed the IDE to search in the specified directories when libraries and header files were needed. By including folders such as GLFW, all the GLFW-related header files were successfully found using <GLFW/..>, etc. Our list of libraries includes:

- **$(SolutionDir)/External Libraries/GLEW/include**
- **$(SolutionDir)/External Libraries/GLFW/include**
- **$(SolutionDir)/External Libraries/glm**
- **$(SolutionDir)/External Libraries/assimp/include**

These directories specify the locations where the header files required for GLEW, GLFW, glm, and assimp are located, respectively. By adding them to the project configuration, we ensured that the compiler could find and correctly use these files, which is crucial for the development of the project.

D) *Input Parameters*

Once we have set up the external headers in our Visual Studio IDE, it enables the ability to automatically locate all the necessary files for our project. This saves us time and effort by avoiding the manual search for each individual file.

After configuring the external headers, the next step is to link the required libraries for our project. To do this, we navigate to the "Linker" and "Input" tabs in the project configuration. In this section, we specify the libraries we want to link.

In the specific case mentioned, the libraries being linked are:

soil2-debug.lib, assimp-vc140-mt.lib, opengl32.lib, glew32.lib glfw3.lib.

Each of these libraries provides specific functionality and resources required for our project.

By linking these libraries, we ensure that all the required dependencies are available for our project and can be effectively used during compilation and execution.

This process of configuring and linking libraries is essential to ensure that our project has access to all the necessary tools and functionalities for its proper functioning. Once this stage is completed, we are ready to proceed with the development of our project, with the assurance that all the libraries are properly linked and available for use.

SHADERS

In the context of a vacation house, specific programs are executed for each section of the graphic visualization.

Essentially, shaders are programs that transform inputs into outputs. These programs are independent of each other and cannot communicate directly; their only form of interaction is through the defined inputs and outputs.

In the case of the vacation house, shaders written in GLSL are used, which is a language similar to C designed specifically for graphics. GLSL offers useful features for manipulating vectors and matrices.

Each shader starts with a version declaration, followed by a list of input and output variables, as well as uniforms. Then, the main function of the shader is defined.

The main function of each shader is the entry point, where the input variables are processed, and the results are generated in the corresponding output variables.

In the vacation house project, shaders were used to load models, manage lighting, create the environment (cubemaps), and perform animations. These shaders are located in a specific folder called "Shaders," and their programming is tailored to their particular function within the project.
MODELOS

A) Asset Import Library (Assimp)

To import complex models into our vacation house, we used the Assimp asset import library. This library allows us to load different file formats into a common data structure, abstracting us from the technical details of each format. When loading a model with Assimp, we gain access to all the necessary data in a scene object that contains nodes representing the model's structure. This solution simplifies model importing and enables us to work with hierarchies and complex structures efficiently.

B) Mesh

In the context of our vacation house, when loading models with Assimp, we obtain a significant amount of data stored in Assimp's data structures. However, to represent objects in OpenGL, we need to transform this data into a format that is understandable by OpenGL. A mesh requires at least one set of vertices, each with position, normal, and texture coordinate information. It also includes indices for indexed drawing and material data such as diffuse or specular textures.

By constructing the mesh, we obtain a list of data that can be used for rendering. We set up the appropriate buffers and define the vertex attribute layout in the shader. To complete the Mesh class, we implement the "Draw" function, which is responsible for binding the necessary textures before calling glDrawElements. However, this can be challenging since we do not know in advance the quantity or type of textures that the mesh may have.

C) Model Load

At this stage of developing our vacation house, we delve into the task of translation and loading using the Assimp library. Our goal is to create a class that represents a complete model, which includes multiple meshes and possibly several textures. For example, we can have a model that contains elements such

as rooms, furniture, and decorations, all loaded as part of a single model.

To achieve this, we leverage Assimp to load the model and then perform the translation to various mesh objects that we have previously created.

It is important to note that we assume the texture file paths mentioned in the model files are local to the model object itself, meaning they are located in the same directory as the model.

To obtain the full path of the textures, we concatenate the texture location string with the directory string that we obtained during the model loading function. This allows us to access the appropriate textures when needed.

In some cases, models obtained from external sources may use absolute paths for texture locations, which is not compatible with our current approach. In these cases, we manually modify the files to change the paths to local paths, ensuring that the textures are in the correct place. We can also perform texture substitutions and adapt them to the model as needed.

MODELING AND ADAPTATION OF MODELS

1. Models to be Used

Most of the objects and the facade were created and designed from scratch. The only models obtained externally were the ones called "Papalote" and "Televisor." Since they were freely licensed, they didn't contain any textures or materials..

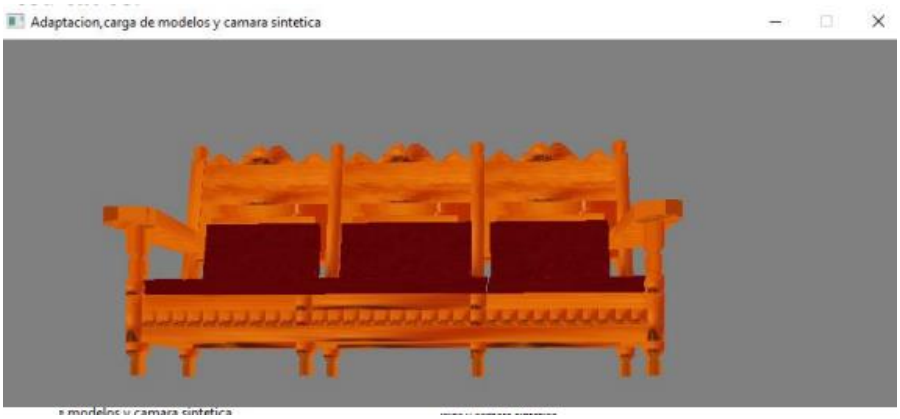


Figure 3: Design of 3D Models

2. Textures and Materials

Starting from the previously selected objects for the facade, we searched for the best textures for the project's design. Once the textures were obtained, we worked on improving the quality of the materials using Maya and GIMP. This allowed us to add finishes to the textures to be used in our models and work on the UV maps of our objects.

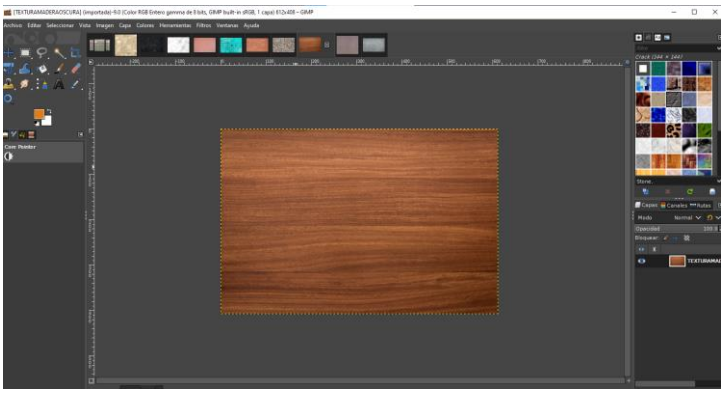


Figure 4: Design of Textures

3. *Completion of Modeling*

Once the materials to be used were selected and applied, it is crucial to verify the basic transformations of the object in order to adapt them to their final position in the environment. If necessary, we work with their respective pivots for the implementation of interactions, simple or complex animations..

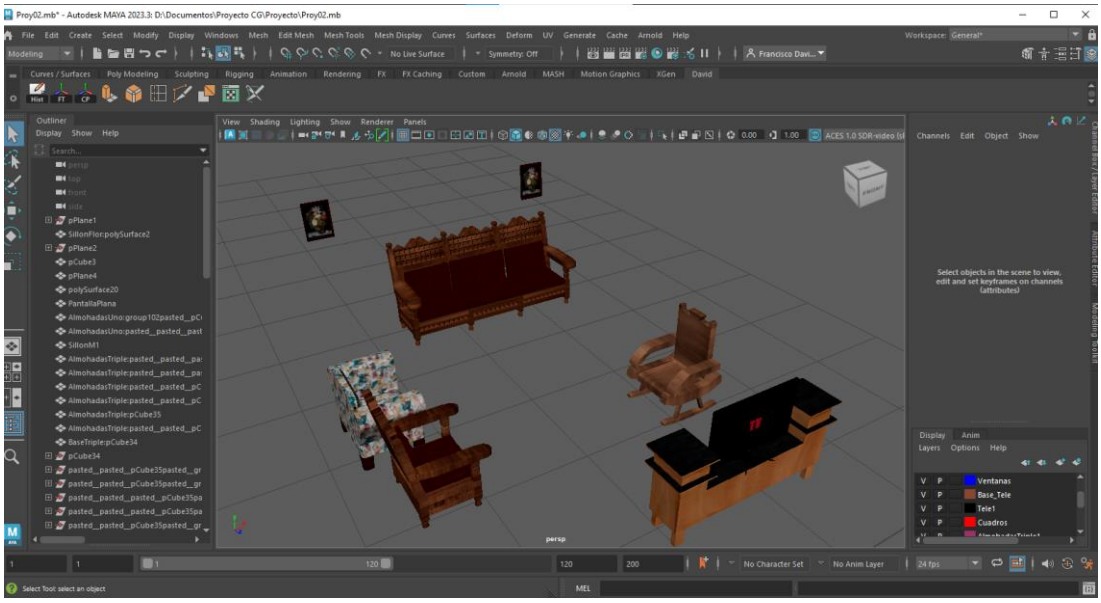


Figure 5

4. *Integration of Models into the Project Code*

Once the object models are finished, they are loaded into the project for review (mainly because they are subject to changes). After loading, we ensure that the objects and their transformations do not affect the integrity of the scene.

Figure 6

5. Version Control

To ensure the integrity of the project, every successful version generated from the integration of the models is uploaded to the repository. This is done to backup functional versions.

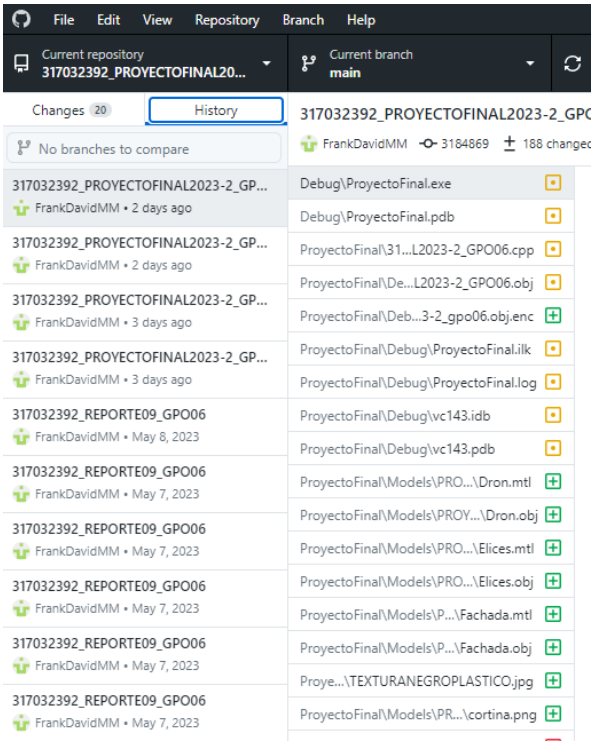


Figure 7

Vacation Home Cuautla

English Technical Manual

SETTING AND LIGHTING:

For the development of the scene, the use of lighting was important, including directional, pointlight, and spotlight. Therefore, in this section, we will present the uniform variables we used for these types of lighting: directional, ambient, diffuse, and specular, as shown in Figures 7 and 8.

```
// Directional light

glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.direction"), -0.2f, -1.0f, -0.3f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.1f, 0.1f, 0.1f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.diffuse"), 0.1f, 0.1f, 0.1f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.specular"), 1.0f, 1.0f, 1.0f);



// Point light 1
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].position"), pointLightPositions[0].x, pointLightPositions[0].y, pointLightPositions
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].ambient"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].diffuse"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].specular"), 1.0f, 1.0f, 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].linear"), 0.009f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].quadratic"), 0.008f);



// Point light 2
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].position"), pointLightPositions[1].x, pointLightPositions[1].y, pointLightPositions
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].ambient"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].diffuse"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[1].specular"), 1.0f, 1.0f, 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[1].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[1].linear"), 0.009f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[1].quadratic"), 0.008f);


glm::vec3 lightColor;
lightColor.x = abs(Light1.x);
```

Figure 7

```
// Point light 3
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].position"), pointLightPositions[2].x, pointLightPositions[2].y, pointLightPositions
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].ambient"), lightColor.x, lightColor.y, lightColor.z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].diffuse"), lightColor.x, lightColor.y, lightColor.z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[2].specular"), 1.0f, 1.0f, 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[2].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[2].linear"), 4.4f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[2].quadratic"), 3.2f);

// Point light 4
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].position"), pointLightPositions[3].x, pointLightPositions[3].y, pointLightPositions
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].ambient"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].diffuse"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[3].specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[3].constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[3].linear"), 0.7f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[3].quadratic"), 0.8f);

// SpotLight
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.position"), camera.GetPosition().x, camera.GetPosition().y, camera.GetPosition().z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.direction"), camera.GetFront().x, camera.GetFront().y, camera.GetFront().z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.ambient"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.diffuse"), 0.0f, 0.0f, 0.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.specular"), 0.0f, 0.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.linear"), 0.9f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.quadratic"), 0.32f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.cutOff"), glm::cos(glm::radians(12.5f)));
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.outerCutOff"), glm::cos(glm::radians(15.0f)));
```

Figure 8

The importance of the uniform values we send to these light sources lies in how the material of our objects will react to these changes and what effect it will have on the final view.

ANIMATIONS

Simple Animations::

The first animation is the falling of the painting. To achieve this, translation and rotation were used. Conditionals were also used to check the position of the object. Once the painting was on the ground, it rotated around the X-axis to be completely on the floor, simulating its fall.

```
/*==================={ Simple Animations /Animaciones Simples }====================*/
model = glm::mat4(1);

model = glm::translate(model, glm::vec3(6.65f, 1.366f, -4.844f));
model = glm::translate(model, glm::vec3(0.0f, AnimCua01, 0.0f));
model = glm::rotate(model, glm::radians(rotCua01), glm::vec3(1.0f, 0.0f, 0.0));
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
Cuadro01.Draw(lightingShader);
```

Figure 9 Model Loading and Transformations

```
/*============================================================================
* simple frame animation01 / Animacion01 Simple Picture
* ============================================================================ */

if (AnimCua01 >= -1.051f && rotCua01 == 0.0f) {

    AnimCua01 -= 0.05f;
}
else if (rotCua01  <= 85.0f ) {
    rotCua01 += 5.0f;
}
else if (ContAnims01 <= 100.0f ) {
    ContAnims01 += 0.5f;
}
else {
    AnimCua01 = 0.0f;
    rotCua01 = 0.0f;
    ContAnims01 = 0.0f;
}
```

Figure 10 Conditionals and Variables responsible for the animation.

The second and third animations involve rotating around the Z-axis. In the first one, the rotation includes conditionals to control the final position, adding realism as if a nail had come loose on one side and the painting remained on the other.

For the rocking chair, similar conditionals were used, but the rotation of this object is more linear as it continuously increases and decreases the rotation angle. This is mainly done to avoid memory leaks and to create a smoother movement.

```
/*=====================================================================
* simple frame animation02 / Animacion02 Simple Picture
* ===================================================================== */

if (rotCua02 >= -45.0f) {
    rotCua02 -= 2.5f;
}
else if(ContAnims02 <= 100.0f)
{
    ContAnims02 += 0.5f;
}
else {
    rotCua02 = 0.0f;
    ContAnims02 = 0.0f;
}
```

Figure 11 Manipulation of values responsible for the rotation of Painting02

```
23
24      /*=====================================================================
25      * simple rocking animation03 / Animacion03 Simple Mecedora
26      * ===================================================================== */
27      if (rotMece >= -5.0f && derMece == false) {
28          rotMece -= 0.2f;
29          if (rotMece < -5.0f) {
30              derMece = true;
31          }
32      }
33      else if(rotMece <= 5.0f && derMece == true){
34          rotMece += 0.2f;
35          if (rotMece > 5.0f) {
36              derMece = false;
37          }
38      }
```

Figure 12 Manipulation of values responsible for the rotation of the Rocking Chair.

Complex Animations::

Drone Flight:

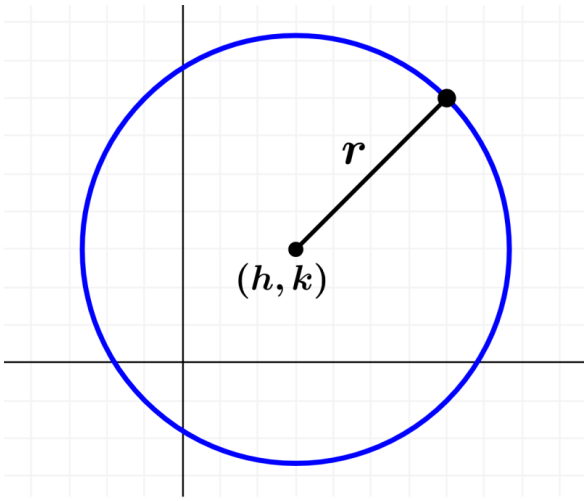
For our first complex animation, we simulate the flight of a drone. Hierarchy is used for this animation since the propellers are separate from the drone object. Each propeller is rotated to simulate its operation, but thanks to the hierarchy and by aligning the translation of the drone body with the propellers, they always follow the location of the drone body. The modeling and assembly of the drone parts involve hierarchy, but the animation itself is achieved by using the equation of a circle to obtain the X and Z coordinates of the drone's position, simulating a circular path without using rotation. In summary, hierarchy is used to model and keep each part of the drone together, and the equation of a circle is used to make it fly in a circular path based on the X and Z axes without using rotation.

```
553
554         /* Hierarchy / Jerarquizacion */
555         Auxiliar = model = glm::translate(model, glm::vec3(PosicionX, 2.5f, PosicionZ));
556         Auxiliar = model = glm::rotate(model, glm::radians(rotDron), glm::vec3(0.0f, 1.0f, 0.0));
557
558         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
559         Dron.Draw(lightingShader);
560
561         model = glm::translate(Auxiliar, glm::vec3(0.066f, 0.015f, 0.052f));
562         model = glm::rotate(model, glm::radians(rotElices), glm::vec3(0.0f, 1.0f, 0.0));
563         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
564         Elices.Draw(lightingShader);
565
566         model = glm::translate(Auxiliar, glm::vec3(0.066f, 0.015f, -0.052f));
567         model = glm::rotate(model, glm::radians(rotElices), glm::vec3(0.0f, 1.0f, 0.0));
568
569         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
570         Elices.Draw(lightingShader);
571
572         model = glm::translate(Auxiliar, glm::vec3(-0.066f, 0.015f, -0.052f));
573         model = glm::rotate(model, glm::radians(rotElices), glm::vec3(0.0f, 1.0f, 0.0));
574
575         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
576         Elices.Draw(lightingShader);
577
578         model = glm::translate(Auxiliar, glm::vec3(-0.066f, 0.015f, 0.052f));
579         model = glm::rotate(model, glm::radians(rotElices), glm::vec3(0.0f, 1.0f, 0.0));
580
581         glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
582         Elices.Draw(lightingShader);
583
```

Figure 13 Drone Hierarchy

```
875
876     /*=====================================================
877     * Drone animation / Animacion Dron
878     * =====================================================*/
879     if (AnguloDron <= 360.0f) {
880         AnguloDron += 0.04f;
881     }
882     else {
883         AnguloDron = 0.0f;
884         rotDron = 0.0f;
885     }
886
887
888     PosicionX = -2.9f + 1.1f * cos(AnguloDron);
889     PosicionZ =  1.5f + 1.1f * sin(AnguloDron);
890
```

Figure 14 Variables used for the animation.



$$(x-h)^2+(y-k)^2=r^2$$

Figure 15 Equation of the Circle.

Kite:

For the animation of the kite, trigonometric functions were used, specifically the Sine function. The movement path of this function is perfect for the animation, so additional values were added to create the desired trajectory for the kite.

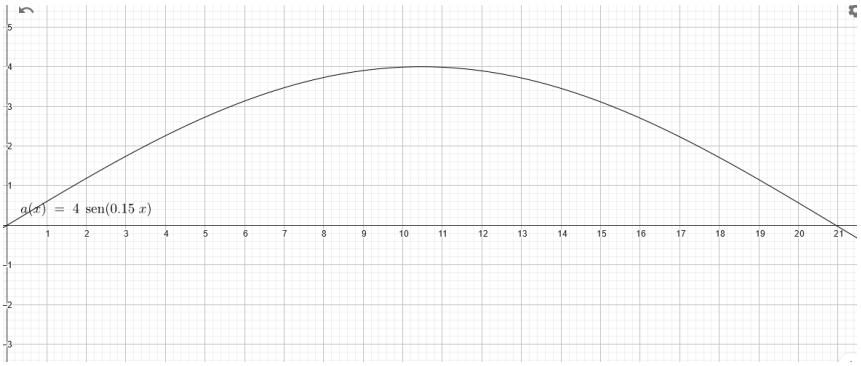


Figure 16 Equation and desired trajectory for our animation.