PROYECTO
CASA DE JIMMY NEUTRON

NO. CUENTA: 317296103

PROFESOR: ING. CARLOS ALDAIR ROMAN BALBUENA

COMPUTACIÓN GRÁFICA | GRUPO: 5

FECHA DE ENTREGA: 8/06/2023

DESCRIPCIÓN DE PROYECTO Y MANUALES TÉCNICOS

REQUERIMIENTOS DE PROYECTO

1.1 PROPOSITOS DEL PROYECTO

Desarrollar un proyecto funcional y entregable que reúna los componentes esenciales del aprendizaje referente a la materia de computación gráfica, al mismo tiempo que demuestre la aplicación efectiva de habilidades fundamentales, prácticas y teóricas.

1.2 OBJETIVOS DEL PROYECTO

Aplicar y demostrar los conocimientos adquiridos durante todo el curso.

- El alumno deberá seleccionar una fachada y dos cuartos que pueden ser reales o ficticios y presentar imágenes de referencia de dichos espacios para su recreación 3D en OpenGL. En la imagen de referencia se debe visualizar 5 objetos que el alumno va a recrear virtualmente y donde dichos objetos deben ser lo más parecido a su imagen de referencia, así como su ambientación.
- Debe contener 4 animaciones dichas animaciones deben tener contexto con el espacio presentado.
- Entregar un manual de usuario en formato PDF, que deberá contener capturas de pantalla que ejemplifiquen cada elemento del manual.
- Entregar un manual técnico que incluya al cronograma de actividades realizadas para la conclusión del proyecto.

1.3 REQUERIMIENTOS DE ALTO NIVEL

Requisitos Técnicos:

- Se deben proporcionar pruebas en plataformas de control de versiones de software, con detalles sobre el progreso en los repositorios (Github).
- Es necesario utilizar técnicas de modelado geométrico, modelado jerárquico, importación de modelos y texturizado de escenarios.

1.4 SUPOSICIONES

Se solicita que la propuesta sea presentada con suficiente antelación para obtener la aprobación previa antes de proceder con la construcción. Esto se realizará a través de un documento de propuesta de proyecto. En caso de ser aprobada, se procederá con el desarrollo del proyecto y se entregará el 8 de junio de 2023

1.4.1 RESTRICCIONES

El proyecto debe ser entregado de manera individual e incluir un manual de usuario que explique cada interacción dentro del entorno virtual recreado. Además, se requiere un manual técnico que contenga la documentación del proyecto, incluyendo objetivos, diagrama de Gantt, alcance del proyecto, limitaciones y documentación del código (que no se limite únicamente a copiar y pegar código, sino que incluya comentarios detallados). Asimismo, se debe compartir la liga de su proyecto en un repositorio en GitHub y se debe subir en la plataforma de classroom a más tardar a las 11:59 pm del jueves para que el profesor pueda descargar el proyecto para su evaluación.

1.5 DESCRIPCION DE ALTO NIVEL Y ALCANCES

- Los objetos recreados repetidamente se contarán como un objeto dentro de la evaluación.
- Puertas, ventanas y escaleras no se cuentan como objetos ya que se debe recrear obligatoriamente ya que son parte de la ambientación.
- Queda estrictamente prohibido hacer la entrega fuera del repositorio si se hace la entrega fuera de esta plataforma no se revisará el proyecto y quedará anulado.
- Queda prohibido recrear cualquier espacio perteneciente a la UNAM con temática de los Simpson, Rick and Morty, Bob esponja, la casa de Kamehouse de Dragon Ball, la casa de coraje el perro cobarde y la casa de las chicas super poderosas.
- Las animaciones realizadas deben de tener contexto es decir no puede estar rotando un objeto nada más para cumplir con la rúbrica, tienen que ir acorde con el contexto del espacio recreado.
- Las personas que no cumplan con la documentación del proyecto y no entreguen en tiempo y forma su proyecto serán acreedores automáticamente una calificación reprobatoria.
- El proyecto es de carácter individual, si se trabajó en equipo una fachada y un cuarto o más elementos el alumno está obligado a crear y a amueblar un cuarto nuevo y a recrear nuevos elementos ya que si no lo hace se tomará el proyecto como copia y se reprobará la materia automáticamente.

1.6 METODOLOGIA DE DESARROLLO DE SOFTWARE

Para este proyecto de modelado en 3D y animación con OpenGL 3, se eligió utilizar la metodología Scrum debido a una serie de razones significativas:

En primer lugar, Scrum ofrece flexibilidad y adaptabilidad, lo cual es esencial dada la naturaleza del proyecto, donde cada semana, al menos, se modela y texturiza un objeto nuevo. Esta metodología permite ajustar las prioridades y tareas en cada sprint, facilitando la gestión de los cambios constantes y los requisitos en evolución del entorno de modelado y animación.

Además, Scrum enfatiza en las entregas frecuentes y el valor temprano. Al trabajar con sprints de corta duración, al final de cada semana se obtiene un incremento funcional del proyecto. Esto permite entregar resultados tangibles al cliente de manera temprana, lo cual es especialmente relevante en un proyecto donde se busca validar y obtener retroalimentación sobre los objetos modelados y texturizados.

Scrum también fomenta la colaboración y la comunicación efectiva entre el equipo de desarrollo, los stakeholders y el cliente. Las reuniones diarias, las sesiones de planificación y revisión de sprints, y la transparencia en el progreso del proyecto, facilitan una comunicación fluida, una resolución conjunta de problemas y una alineación constante entre todas las partes involucradas.

Además, Scrum promueve la mejora continua y el aprendizaje del equipo. Al final de cada sprint, se realiza una retrospectiva para analizar lo que funcionó bien y lo que se puede mejorar en términos de modelado, texturización y proceso de desarrollo en general. Esto impulsa una cultura de mejora constante y eficiencia en las entregas.

1.7.1 RESUMEN DEL CRONOGRAMA DE HITOS

Calendario / Schedule

Casa Jimmy Neutron

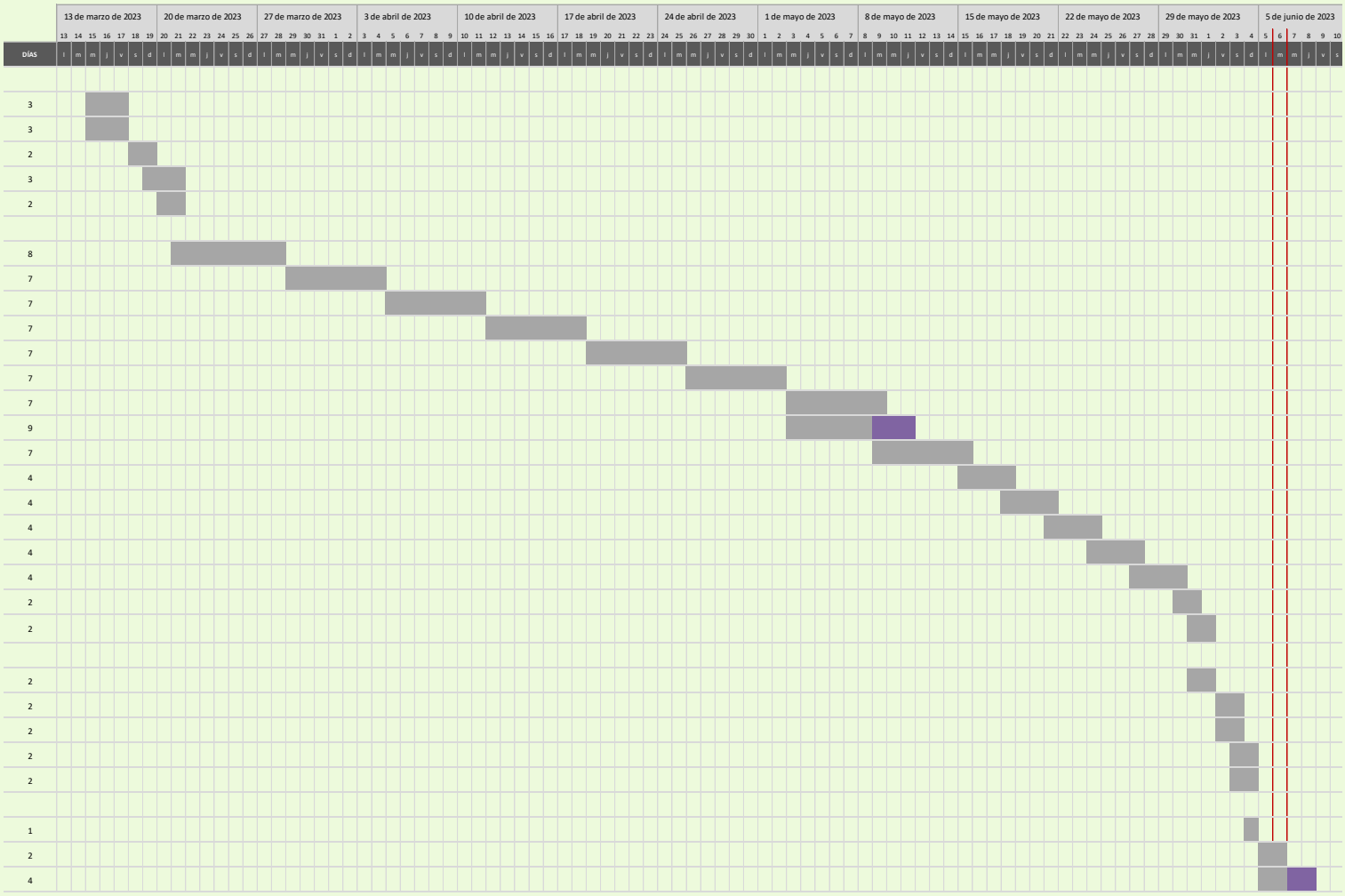
Proyecto Final Teoria

317296103

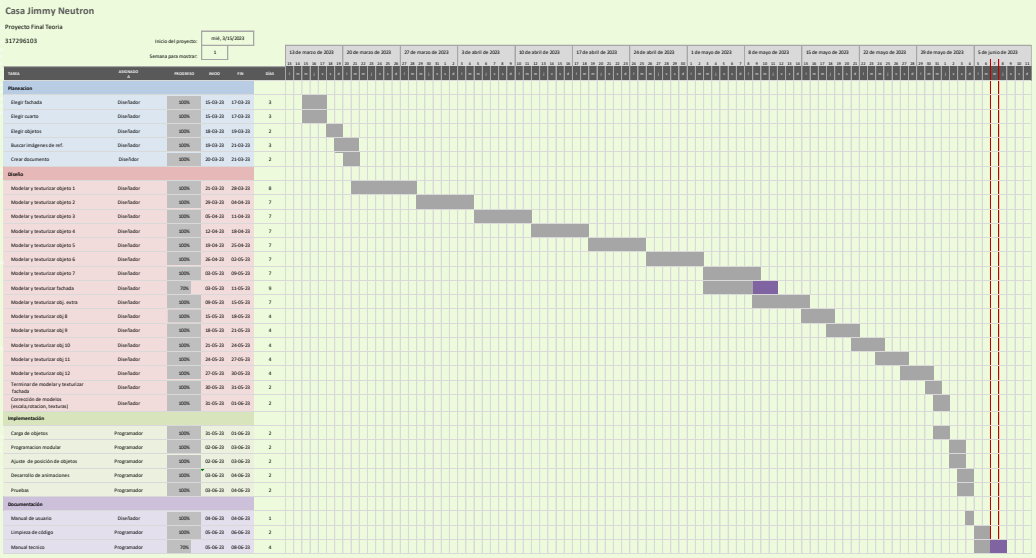
Inicio del proyecto: mié, 3/15/2023

Semana para mostrar: 1

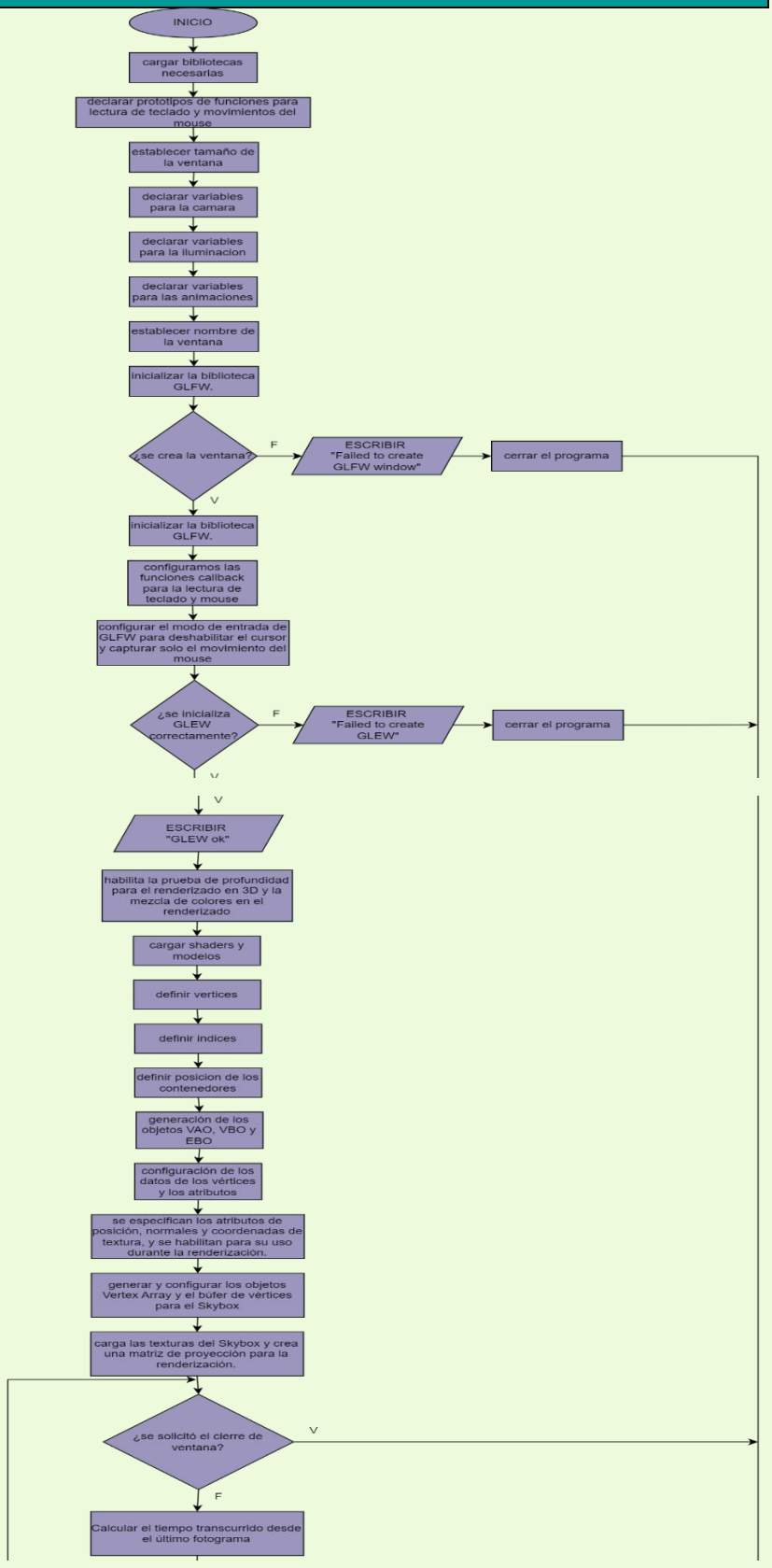
TAREA	ASIGNADO A	PROGRESO	INICIO	FIN	DÍAS
Planeacion					
Elegir fachada	Diseñador	100%	15-03-23	17-03-23	3
Elegir cuarto	Diseñador	100%	15-03-23	17-03-23	3
Elegir objetos	Diseñador	100%	18-03-23	19-03-23	2
Buscar imágenes de ref.	Diseñador	100%	19-03-23	21-03-23	3
Crear documento	Diseñdor	100%	20-03-23	21-03-23	2
Diseño					
Modelar y texturizar objeto 1	Diseñador	100%	21-03-23	28-03-23	8
Modelar y texturizar objeto 2	Diseñador	100%	29-03-23	04-04-23	7
Modelar y texturizar objeto 3	Diseñador	100%	05-04-23	11-04-23	7
Modelar y texturizar objeto 4	Diseñador	100%	12-04-23	18-04-23	7
Modelar y texturizar objeto 5	Diseñador	100%	19-04-23	25-04-23	7
Modelar y texturizar objeto 6	Diseñador	100%	26-04-23	02-05-23	7
Modelar y texturizar objeto 7	Diseñador	100%	03-05-23	09-05-23	7
Modelar y texturizar fachada	Diseñador	70%	03-05-23	11-05-23	9
Modelar y texturizar obj. extra	Diseñador	100%	09-05-23	15-05-23	7
Modelar y texturizar obj 8	Diseñador	100%	15-05-23	18-05-23	4
Modelar y texturizar obj 9	Diseñador	100%	18-05-23	21-05-23	4
Modelar y texturizar obj 10	Diseñador	100%	21-05-23	24-05-23	4
Modelar y texturizar obj 11	Diseñador	100%	24-05-23	27-05-23	4
Modelar y texturizar obj 12	Diseñador	100%	27-05-23	30-05-23	4
Terminar de modelar y texturizar fachada	Diseñador	100%	30-05-23	31-05-23	2
Corrección de modelos (escala,rotacion, texturas)	Diseñador	100%	31-05-23	01-06-23	2
Implementación					
Carga de objetos	Programador	100%	31-05-23	01-06-23	2
Programacion modular	Programador	100%	02-06-23	03-06-23	2
Ajuste de posición de objetos	Programador	100%	02-06-23	03-06-23	2
Desarrollo de animaciones	Programador	100%	03-06-23	04-06-23	2
Pruebas	Programador	100%	03-06-23	04-06-23	2
Documentación					
Manual de usuario	Diseñador	100%	04-06-23	04-06-23	1
Limpieza de código	Programador	100%	05-06-23	06-06-23	2
Manual tecnico	Programador	70%	05-06-23	08-06-23	4

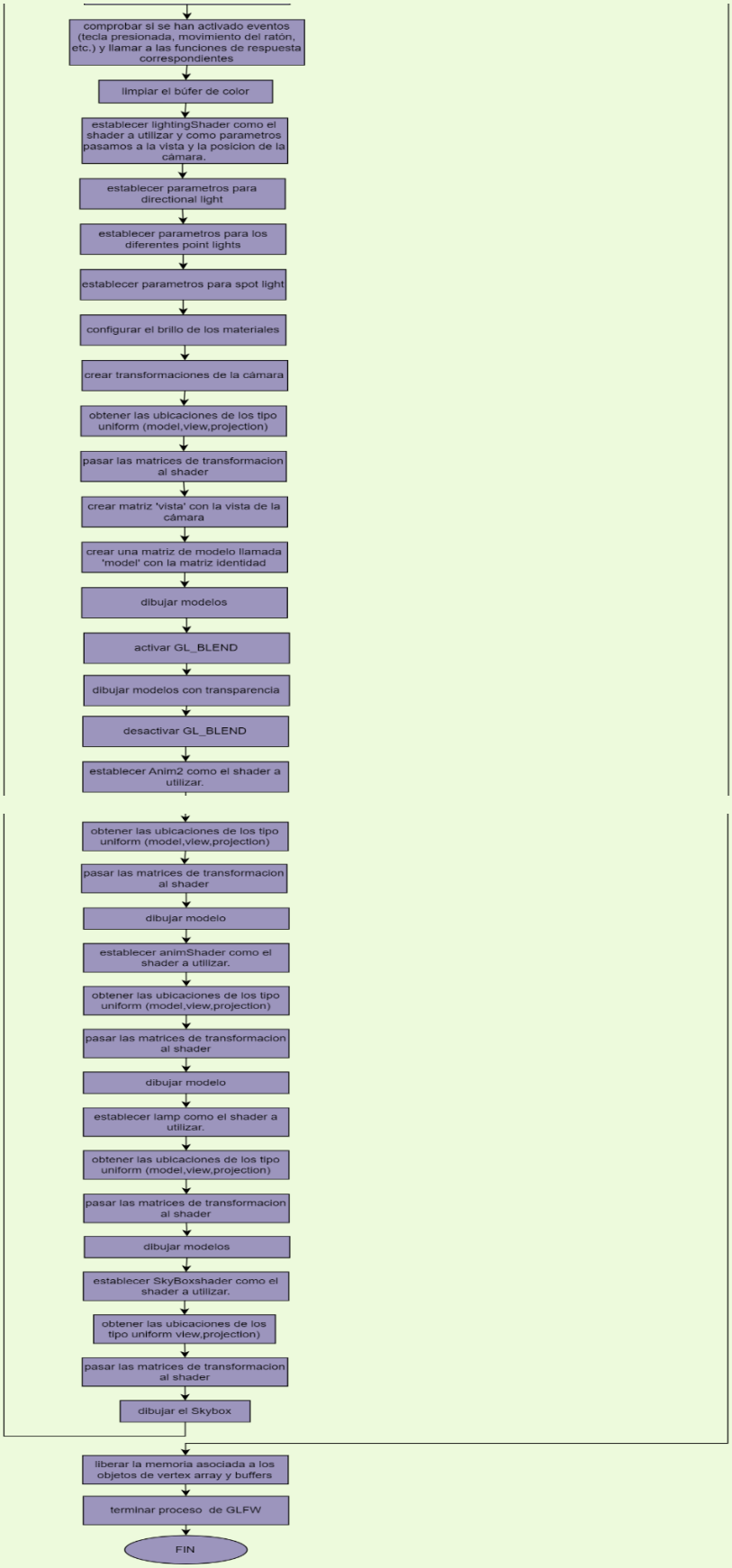


Se separó el calendario del diagrama para que estos fueran ampliados y vistos con mejor detalle para el lector, pero a continuación es el diagrama unido.



1.7.2 DIAGRAMA DE FLUJO / FLOWCHART




comprobar si se han activado eventos (tecla presionada, movimiento del ratón, etc.) y llamar a las funciones de respuesta correspondientes
limpiar el búfer de color
establecer lightingShader como el shader a utilizar y como parametros pasamos a la vista y la posicion de la cámara.
establecer parametros para directional light
establecer parametros para los diferentes point lights
establecer parametros para spot light
configurar el brillo de los materiales
crear transformaciones de la cámara
obtener las ubicaciones de los tipo uniform (model,view,projection)
pasar las matrices de transformacion al shader
crear matriz 'vista' con la vista de la cámara
crear una matriz de modelo llamada 'model' con la matriz identidad
dibujar modelos
activar GL_BLEND
dibujar modelos con transparencia
desactivar GL_BLEND
establecer Anim2 como el shader a utilizar.
obtener las ubicaciones de los tipo uniform (model,view,projection)
pasar las matrices de transformacion al shader
dibujar modelo
establecer animShader como el shader a utilizar.
obtener las ubicaciones de los tipo uniform (model,view,projection)
pasar las matrices de transformacion al shader
dibujar modelo
establecer lamp como el shader a utilizar.
obtener las ubicaciones de los tipo uniform (model,view,projection)
pasar las matrices de transformacion al shader
dibujar modelos
establecer SkyBoxshader como el shader a utilizar.
obtener las ubicaciones de los tipo uniform view,projection)
pasar las matrices de transformacion al shader
dibujar el Skybox
liberar la memoria asociada a los objetos de vertex array y buffers
terminar proceso de GLFW
FIN

1.8 RESUMEN DE COSTOS Y APROXIMACIONES

La siguiente tabla de costos presenta una desglose detallado de los gastos asociados al proyecto. En ella, se incluyen tanto los costos fijos, como la adquisición de equipos y servicios recurrentes, así como los costos variables, como los honorarios de los profesionales involucrados. El precio de venta se ha calculado considerando una ganancia del 25% sobre los costos totales, y se ha incluido el Impuesto al Valor Agregado (IVA) correspondiente. Esta tabla nos permite tener una visión clara de los gastos incurridos y nos ayuda en la planificación financiera del proyecto.

Elemento de costo	Periodo 15/03/23 - 08/06/23	
	Fijo	Variable
Amortizacion laptop	$ 3,208.25	---
Amortizacion mouse	$ 51.46	---
Licencia 1: Software Maya	$ 11,652.00	---
Licencia 2: Software 365 para empresas	$ 877.84	---
Internet	$ 4,468.00	---
Agua	$ 1,288.00	---
Luz	$ 554.00	---
Programador	---	$ 1,440.00
Diseñador	---	$ 20,625.00
Comidas	$ 18,200.00	---
Total, de costos		$ 62,364.55
Precio de venta con un 25% de margen de ganancia		$ 83,152.73
Precio de venta + IVA		$ 96,457.17
Utilidad neta		**$ 34,092.62**

1.9 PROPUESTA PARA EL PROYECTO DE COMPUTACIÓN GRAFICA

- **ESTRUCTURA PRINCIPAL (FACHADA)**

Se basará en la siguiente estructura (Fig.1), la cual albergara las áreas funcionales definidas a continuación, se aprecia además la ambientación exterior a esta estructura principal, los cuales también buscaran ser recreados para los propósitos de este proyecto.

- **CUARTOS POR RECREAR**

1. Se ubica en la parte superior derecha de la estructura, donde se encuentran objetos comunes en un dormitorio solo que, con temática del espacio, en especial cohetes, como una cama, escritorio, computadora personal, despertador, pizarrón, cómoda, lampara, mapa, silla, entre otros. (Fig.2)
2. Se ubica en la parte inferior derecha al fondo de la estructura, donde se encuentran objetos comunes en una cocina solo que propios de la temática de la serie, como un refrigerador, mesa, sofá, fregadero, estufa, telefono de pared, entre otros. (Fig.4)

- **OBJETOS**

En este caso se recrearán solamente algunos objetos, principalmente; la cama, la mesa, la lampara de noche, el escritorio, la silla, el avión colgante y el pizarrón ya que estos objetos son muy característicos de la habitación del personaje en que está inspirado este proyecto, se consideran importantes para la ambientación.

Para la cocina se recreará la mesa, el refrigerador, el teléfono de pared, el fregadero y la lampará colgante

- **ANIMACIONES**

1. Animación de movimiento en la silla y el avión colgante, estos dos objetos tienen la particularidad de rotar sobre su mismo eje ya que uno está colgando de un hilo/cuerda y la silla es una silla giratoria la cual lo que gira únicamente es la parte del asiento, estos pueden rotar 360 grados.
2. Animación de apertura de puerta, es un movimiento característico del objeto al abrirse.
3. Animación de encendido y apagado de lampara, es característico de este objeto.
4. Animación de movimiento del cajón inferior izquierdo del escritorio, esto consiste en abrirse el cajón.
5. Animación del movimiento de las copas del árbol exterior
6. Animación del alienígena que se encuentra entre la cama y el escritorio

Fig. 1 Fachada

Fig. 2 Cuarto a recrear

Fig. 3 Fachada trasera

Fig. 4 Cocina

MANUAL TÉCNICO:

Plataformas de Trabajo.

Para el desarrollo del ambiente fue necesario trabajar bajo el entorno de desarrollo integrado de Microsoft Visual Studio (IDE) el cual es utilizado para desarrollar programas informáticos, aplicaciones, y servicios, para mi caso, siguiendo de la mano las operaciones de OpenGL que se considera principalmente una API (una interfaz de programación de aplicaciones) el cual brinda un gran conjunto de funciones para manipular gráficos e imágenes. Sin embargo, OpenGL por sí mismo no es una API, sino simplemente una especificación, desarrollada y mantenida por el Grupo Khronos.

La especificación OpenGL define exactamente cuál debe ser el resultado/salida de cada función y cómo debe funcionar. Luego, depende de los desarrolladores implementar estas especificaciones, encontrar una solución de cómo debería operar una función. Dado que la especificación de OpenGL no brinda detalles de implementación, las versiones desarrolladas reales de OpenGL pueden tener diferentes implementaciones, siempre que sus resultados cumplan con la especificación (y, por lo tanto, sean los mismos para los usuarios).

Es importante denotar que para el desarrollo de este proyecto se trabajó con las bibliotecas OpenGL que están escritas en C y permiten muchas derivaciones en otros lenguajes, pero en esencia sigue siendo una biblioteca C.

Dado que muchas de las construcciones de lenguaje de C no se traducen tan bien 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 de Jimmy Neutron, siguiendo las especificaciones de requerimientos funcionales del apartado anterior. Un objeto en OpenGL es una colección de opciones que representa un subconjunto del estado de OpenGL. Por ejemplo, podríamos tener un objeto que represente la configuración de la ventana de dibujo; luego podríamos establecer su tamaño, cuántos colores admite, etc. Uno podría visualizar un objeto como una estructura tipo C.

Lo bueno de usar estos objetos es que se puede definir más de un objeto en la aplicación, configurar sus opciones y cada vez que comenzamos una operación que usa el estado de OpenGL, vinculamos el objeto con nuestra configuración preferida. Hay objetos, por ejemplo, que actúan como objetos contenedores para los datos del modelo 3D (una casa o un personaje) y cada vez que se quiere dibujar uno de ellos, se vincula el objeto que contiene los datos del modelo que se quiere dibujar (primero se crean y configuran las opciones para estos objetos). Tener varios objetos permite especificar muchos modelos y siempre que se quiere dibujar un modelo específico, simplemente se vincula el objeto correspondiente antes de dibujar sin volver a configurar todas sus opciones.

2. Parámetros Básicos del Entorno

2.1. Uso de la Ventana

Lo primero a configurar, antes de crear el entorno gráfico, es un contexto OpenGL y una ventana de aplicación para dibujar. Sin embargo, esas operaciones son especificas por sistema operativo y OpenGL intenta abstraerse de estas operaciones a propósito. Esto significa que hay que crear una ventana, definir un contexto y manejar la entrada del usuario manualmente. Afortunadamente, existen bastantes bibliotecas que brindan la funcionalidad que necesaria, algunas dirigidas específicamente a OpenGL. Se utilizó GLFW. GLFW es una biblioteca, escrita en C, dirigida específicamente a OpenGL. GLFW brinda las necesidades básicas requeridas para mostrar cosas en la pantalla. Permite crear un contexto OpenGL, definir parámetros de ventana y manejar la entrada del usuario, que es suficiente para este propósito.

2.2. Vinculación

Para que el proyecto use GLFW, se necesita vincular la biblioteca con el proyecto. Esto se pudo lograr especificando que se quiere usar glfw3.lib en la configuración del enlazador, pero el proyecto aún necesita saber dónde encontrar glfw3.lib ya que estas bibliotecas de terceros se encuentran en un directorio diferente. Por lo tanto, primero se debe agregar este directorio al proyecto.

Se puede especificar al IDE una ruta dinámica para que reconozca todos los archivos, las rutas son las siguientes.

```
$(SolutionDir)/External
Libraries/GLEW/lib/Release/Win32

$(SolutionDir)/External
Libraries/GLFW/lib-vc2015

$(SolutionDir)/External
Libraries/SOIL2/lib
```

```
$(SolutionDir)/External
Libraries/assimp/lib
```

2.2.1 Bibliotecas Externas Incluidas.

Para que el proyecto pudiera saber en que parte buscar las herramientas necesarias, se insertó manualmente la cadena de ubicación adecuada, el IDE también buscará en esos directorios cuando busque bibliotecas y archivos de encabezado. Tan pronto como se incluyeron las carpetas como la de GLFW, se pudo encontrar todos los archivos de encabezado para GLFW al incluir <GLFW/..>., etc. La lista de Biblioteca incluye:

```
$(SolutionDir)/External
Libraries/GLEW/include
```

```
$(SolutionDir)/External
Libraries/GLFW/include
```

```
$(SolutionDir)/External Libraries/glm
```

```
$(SolutionDir)/External
Libraries/assimp/include
```

2.2.2 Parámetros de Entrada del Vinculador.

Una vez que se establecieron las cabeceras externas, en el IDE, VisualStudio puede encontrar todos los archivos necesarios, y finalmente se pueden vincular las bibliotecas al proyecto yendo a la pestaña Vinculador > Entrada para terminar nuestra configuración.

```
soil2-debug.lib;assimp-vc140-
mt.lib;opengl32.lib;glew32.lib;glfw3.
lib;
```

3 Shaders

Se utilizaron shaders para ayudar a construir el entorno de acuario de este proyecto, Los shaders son pequeños programas que descansan en la GPU.

Estos programas se ejecutan para cada sección específica de la canalización de gráficos. En un sentido básico, los shaders no son más que programas que transforman entradas en salidas. Los shaders también son programas muy aislados en el sentido de que no se les permite comunicarse entre sí; la única comunicación que tienen es a través de sus entradas y salidas.

Los shaders utilizados están escritos en el lenguaje GLSL similar a C. GLSL está diseñado para su uso con gráficos y contiene características útiles específicamente dirigidas a la manipulación de vectores y matrices.

Los shaders siempre comienzan con una declaración de versión, seguida de una lista de variables de entrada y salida, uniformes y su función principal.

El punto de entrada de cada shader está en su función principal donde se procesa cualquier variable de entrada y se muestran los resultados en sus variables de salida.

Se utilizaron shaders para la carga de modelos, para la iluminación, para la ambientación (cubemaps), y para las animaciones en el cuarto.

Aparecen listados en una carpeta especifica llamada Shaders y su programación es autóctona de su función.

4. Carga de Modelos

4.1 ASSIMP

Para nuestro entorno no se pueden definir manualmente todos los vértices, las normales y las coordenadas de textura de formas complicadas.

En su lugar se hizo uso de herramientas de modelado 3D, como Maya, para crear formas complicadas y aplicarles texturas a través de mapeo UV. Luego, las herramientas generan automáticamente todas las coordenadas de vértice, las normales de vértice y las coordenadas de textura mientras las exportan a un formato de archivo de modelo que podemos usar (OBJ). De esta forma, se cuenta con un extenso conjunto de herramientas para crear modelos de alta calidad sin tener que preocuparse demasiado por los detalles técnicos. Todos los aspectos técnicos están ocultos en el archivo del modelo exportado. Sin embargo, como programadores de gráficos, tenemos que preocuparnos por estos detalles técnicos.

La biblioteca de importación de modelos utilizada es Assimp, que significa Biblioteca abierta de importación de activos. Assimp puede importar docenas de formatos de archivo de modelo diferentes (y exportar a algunos también) cargando todos los datos del modelo en las estructuras de datos generalizadas de Assimp. Tan pronto como Assimp haya cargado el modelo, podemos recuperar todos los datos que necesitamos de las estructuras de datos de Assimp. Debido a que la estructura de datos de Assimp permanece igual, independientemente del tipo de formato de archivo que se importa, este abstrae de todos los diferentes formatos de archivo que existen.

Al importar un modelo a través de Assimp, se carga todo el modelo en un objeto de escena que contiene todos los datos del modelo/escena importados. Assimp luego tiene una colección de nodos donde cada nodo contiene índices de datos almacenados en el objeto de escena donde cada nodo puede tener cualquier número de hijos.

4.2 Mesh

Con Assimp pudimos cargar muchos modelos diferentes en la aplicación, pero una vez cargados, todos se almacenan en las estructuras de datos de Assimp. Lo que eventualmente se necesita es transformar esos datos a un formato que

OpenGL entienda para que podamos representar los objetos. Una malla debería necesitar al menos un conjunto de vértices, donde cada vértice contiene un vector de posición, un vector normal y un vector de coordenadas de textura. Una malla también debe contener índices para el dibujo indexado y datos de materiales en forma de texturas (mapas difusos/especulares)

Gracias al constructor, se obtienen grandes listas de datos de malla que se pueden usar para renderizar. Se necesita configurar los búferes apropiados y especificar el diseño del shader de vértices a través de punteros de atributo de vértice para que finalmente con la última función que se necesita definir para que la clase Mesh esté completa es su función 'Draw'. Antes de renderizar la malla, primero se les deben enlazar las texturas apropiadas antes de llamar a glDrawElements. Sin embargo, esto es difícil ya que se desconoce la existencia y cantidad de texturas hay en la malla y el tipo.

4.3 Model

A partir de este punto en el proyecto, Se comienza a crear el código de traducción y carga real con Assimp. El objetivo es crear otra clase que represente un modelo en su totalidad, es decir, un modelo que contenga múltiples mallas, posiblemente con múltiples texturas. Una ventana que tiene marco, adornos y cristales, aún podría cargarse como un solo modelo. Se carga el modelo a través de Assimp y se traduce a varios objetos de malla que hemos creado.

Teniendo en cuenta que se asume que las rutas de los archivos de textura en los archivos del modelo son locales para el objeto del modelo real, Ejemplo. en el mismo directorio que la ubicación del propio modelo.

Entonces se puede simplemente concatenar la cadena de ubicación de textura y la cadena de directorio que se recuperó anteriormente (en la función loadModel) para obtener la ruta de textura completa (es por eso que la función GetTexture también necesita la cadena de directorio).

Algunos modelos que se obtuvieron en Internet para este proyecto usaron rutas absolutas para sus ubicaciones de textura, lo que no funcionó al momento de llegar a este punto. En ese caso, editamos manualmente el archivo para usar rutas locales para las texturas (si es posible). O se reemplazaron las texturas y se inició un proceso de adaptación al modelo que se describe a continuación.

2.1 MANUAL TÉCNICO:

PROCESO DE ELABORACIÓN Y ADAPTACIÓN

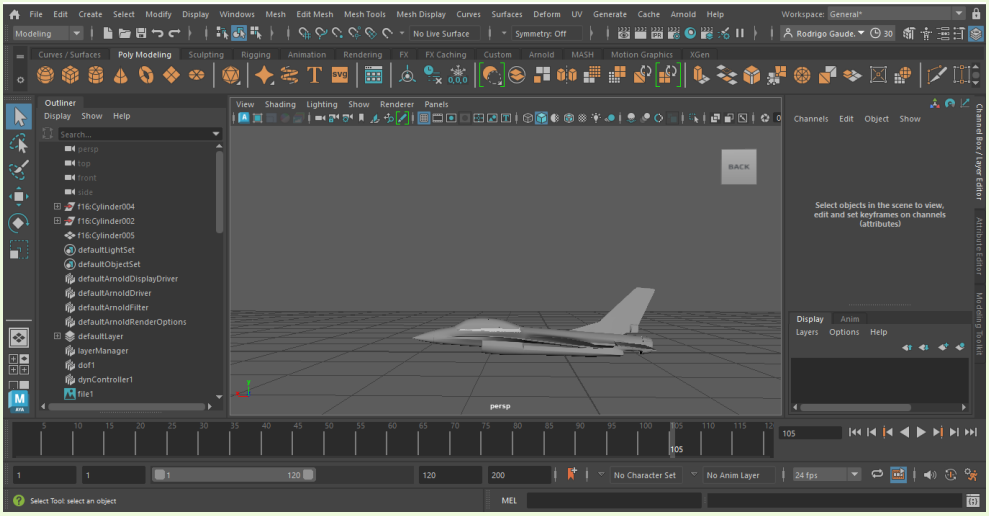
Con base en el cronograma de actividades y la culminación de la fase de Planeación y Diseño, se llegó a la fase contigua de Implementación donde se añadieron entradas, cambios, y eliminación de elementos para ajustarse a la entrega pactada en el análisis de requerimientos.

Entre estos rubros se definieron procesos para añadir objetos;

1. Importación o creación del modelo.

La mayoría de los modelos del proyecto son material intelectual propio, pero sí hay modelos, o partes dé, que se obtuvieron de plataformas que ofertan diversos modelos gráficos con distintas licencias de uso.

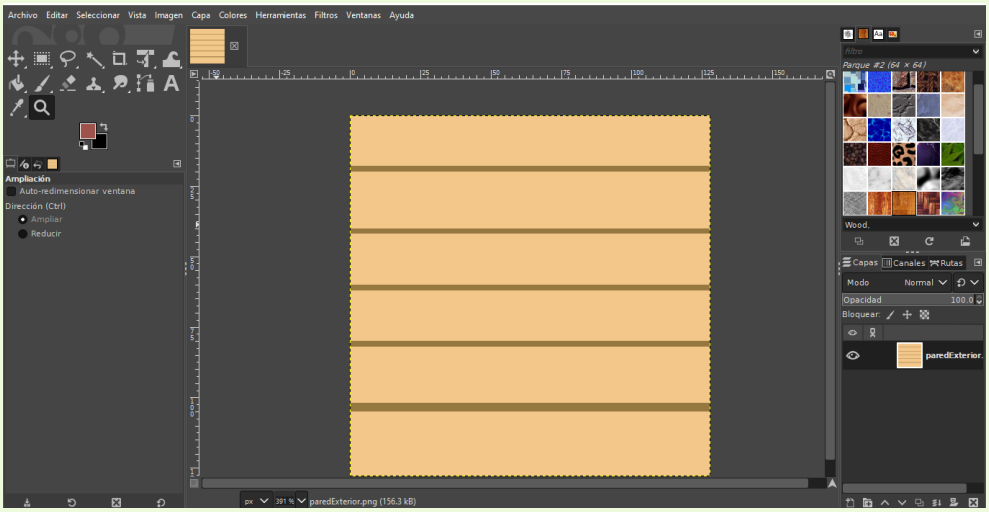
Los modelos obtenidos al ser de licencia gratuita no contenían texturas ni materiales. Para este punto se establecieron parámetros para su integración dentro del escenario.



2. Selección de Texturas y adaptación de materiales.

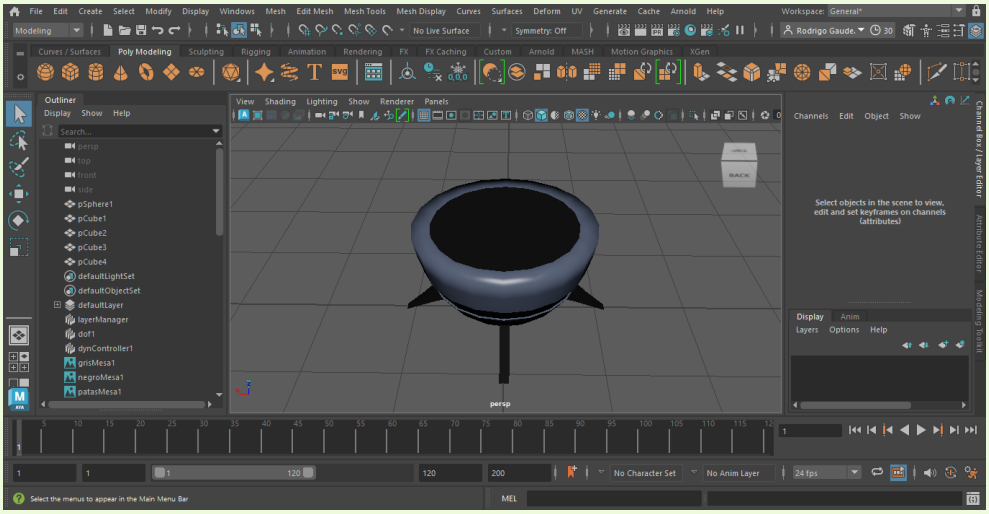
Los modelos seleccionados y que fueron filtrados para la fase final de introducción al escenario, fueron revisitados para la toma de texturas. Se trabajaron las imágenes con el

software GIMP, para poder utilizarse como textura y moldear el mapa de uv. Con la finalidad de crear una mejor ambientación al entorno.



3. Convergencia de materiales y finalización del modelado

Se acoplan los materiales al modelo, se vuelven a revisar transformaciones básicas del objeto para adecuar su posición dentro del escenario y finalmente, se importa para su posterior incorporación al entorno de desarrollo integrado que trabaja la integración completa del proyecto.



4. Integración del modelo al escenario final

Finalmente, se intercambia el espacio de trabajo para continuar con la integración y acoplamiento al entregable final. Se asegura que el nuevo modelo no afecte la estabilidad

del proyecto, que no existan posicionamientos no deseados, o que bien, la integración del componente sea la deseada.

5. Control de Versiones

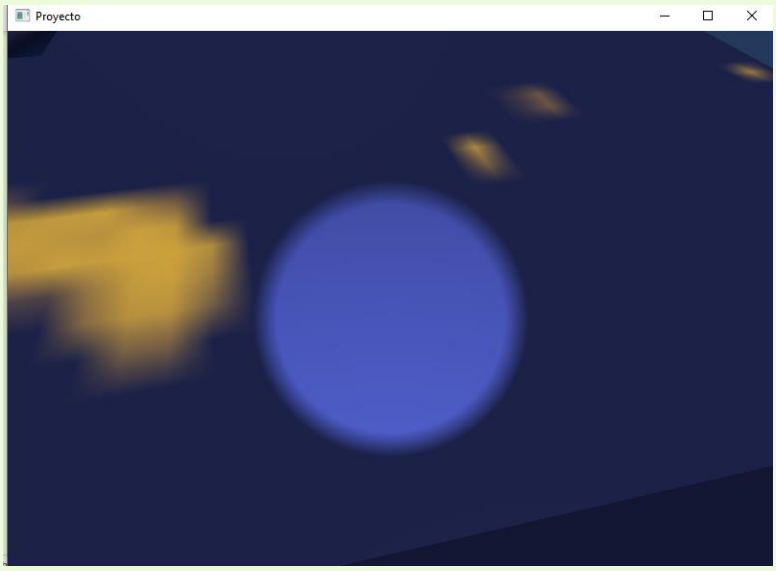
Se guardan los nuevos cambios y se actualizan en el repositorio central; se contemplan comentarios en la documentación como el cambio realizado y la funcionalidad.

Se utiliza la herramienta colaborativa de GitHub para establecer un parámetro de control de versiones, respaldo y trabajo contiguo.

2.2 MANUAL TÉCNICO: USO DE AMBIENTACIÓN: ILUMINACIÓN

Para este apartado se utilizó la spotlight en la posición de la cámara sintética a forma de que el usuario al navegar pareciera que trae una lampara de mano consigo.

```
// 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"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.diffuse"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.specular"), 1.0f, 1.0f, 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.linear"), 0.14f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.quadratic"), 0.07f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.cutOff"), glm::cos(glm::radians(12.5f)));
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.outerCutOff"), glm::cos(glm::radians(15.0f)));
```



Se ocupó únicamente dos pointlight, aunque no eran necesarias, se dejaron las otras dos ya que no afectan, sin embargo, pudieron haberse eliminado del shader y después del código fuente.

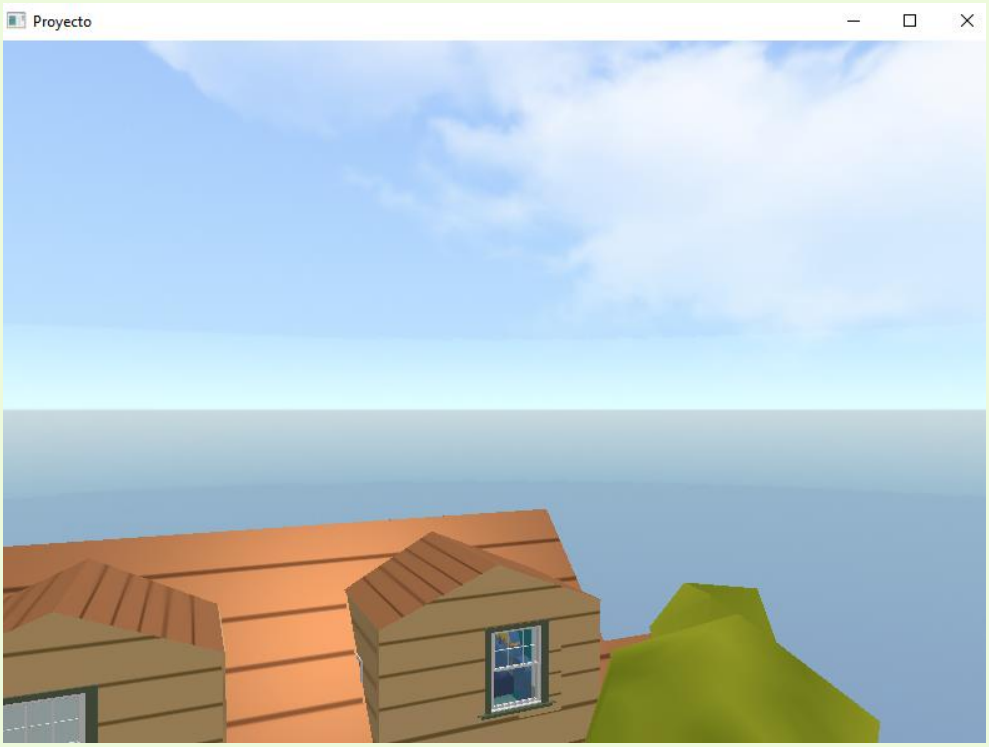
```
// Directional light
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.direction"), -0.2f, -1.0f, -0.3f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.5f, 0.5f, 0.5f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.diffuse"), 0.4f, 0.4f, 0.4f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.specular"), 0.5f, 0.5f, 0.5f);



// Point light 1
glm::vec3 lightColor;
//Para encender y apagar la lampara
lightColor.x = abs(sin(glfwGetTime() * Light1.x));
lightColor.y = abs(sin(glfwGetTime() * Light1.y));
lightColor.z = abs(sin(glfwGetTime() * Light1.z));

glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].position"), pointLightPositions[0].x, pointLightPositions[0].y, pointLightPositions[0]
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].ambient"), lightColor.x, lightColor.y, lightColor.z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].diffuse"), lightColor.x, lightColor.y, lightColor.z);
//Specula es el punto brilloso que se ve al rebotar la luz en una superficie
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].specular"), 1.0f, 1.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].constant"), 1.0f);
//Definen que tan brillante y grande
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].linear"), 0.7f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].quadratic"), 1.8f);
```

La iluminación se dejó para que quedara de acuerdo con el ambiente, siendo de día.



2.3 MANUAL TÉCNICO: USO DE ANIMACIONES

Para las animaciones se utilizaron dos shaders que brindan movimiento de acuerdo con una función elegida.

Para el movimiento de la copa del árbol se seleccionó una función y la aplicación del efecto en ciertos ejes específicos para que se genere un movimiento oscilatorio de forma que parezca que se mueve cuando el aire le pega y regresa, esto con una amplitud pequeña para que sea suave el movimiento.

```
const float amplitude = 0.5;
const float frequency = 0.05;
const float PI = 3.14159;
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform float time;

void main()
{
  float distance = length(aPos);
  float effect = amplitude*sin(-PI*distance*frequency+time);
  gl_Position = projection*view*model*vec4(aPos.x+effect,aPos.y, aPos.z+effect,1);
  TexCoords=vec2(aTexCoords.x,aTexCoords.y);

}
```

En el caso de la animación del Alíen se eligieron otros parámetros y aplicación del efecto de forma que sí se deforme un poco el objeto, ya que el alíen al no estar en su armadura no es capaz de mantener su forma y pareciera una forma gelatinosa.

```
const float amplitude = 0.15;
const float frequency = 4.0;
const float PI = 3.14159;
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform float time;

void main()
{
  float distance = length(aPos);
  float effect = amplitude*sin(-PI*distance*frequency+time);
  gl_Position = projection*view*model*vec4(aPos.x+effect,aPos.y*effect, aPos.z+effect,1);
  TexCoords=vec2(aTexCoords.x,aTexCoords.y);

}
```

Para la demás animación lo que se hizo fue crear variables booleanas que activaran la animación y que la detuvieran en cierto momento. Se agregaron estas líneas en las funciones main(), KeyCallback() y DoMovement(). A continuación, se encuentra el diccionario de funciones y variables para más detalles.

2.4 MANUAL TÉCNICO: DICCIONARIO DE FUNCIONES

Función	Descripción
Main()	Función principal en la que se ejecuta la parte de creación de la ventana, los objetos VAO, VBO y EBO, las matrices de transformación para el shader, manejo de la cámara, carga los shaders, carga y dibujo de modelos, manejo de la iluminación y creación del skybox

KeyCallBack(GLFWwindow *window, int key, int scancode, int action, int mode)	Es una función de devolución de llamada que se activa cuando se produce un evento de teclado. Toma como parámetros la ventana GLFW window, el código de tecla key, el código de escaneo scancode, la acción realizada action y el modo de entrada mode. Esta función se utiliza para manejar y responder a eventos de teclado, como presionar, soltar o mantener presionada una tecla específica, por ejemplo, al presionar la tecla ESC se cierra la ventana.
MouseCallBack(GLFWwindow *window, double xPos, double yPos)	Es una función de devolución de llamada que se activa cuando se produce un evento de mouse. Toma como parámetros la ventana GLFW window y las coordenadas del cursor del ratón xPos y yPos. Esta función se utiliza para manejar y responder a eventos de mouse, como movimientos del cursor.
DoMovement()	Se encarga de gestionar el movimiento dentro del entorno interactivo. Permite controlar la posición y orientación de un objeto o cámara en respuesta a eventos de entrada, como presionar teclas o mover el ratón.

2.5 MANUAL TÉCNICO: DICCIONARIO DE VARIABLES

Nombre	Tipo	Descripción
WIDTH	const Gluint	Define el ancho de la ventana de visualización
HEIGHT	const Gluint	Define la altura de la ventana de visualización
SCREEN_WIDTH y SCREEN_HEIGHT	int	Representan el ancho y alto de la pantalla en la que se está ejecutando la aplicación.
camera	Camera	Es un objeto de la clase Camera que representa la cámara en el espacio 3D. Se inicializa con una posición inicial
lastX y lastY	Glfloat	Almacenan las últimas coordenadas del cursor del raton.
keys[1024]	bool	Se utiliza para rastrear el estado de las teclas del teclado.

firstMouse	bool	Indica si es la primera vez que se mueve el raton. Se utiliza para gestionar los movimientos del mouse correctamente.
range y movCamera	float	Se utilizan para controlar el rango de movimiento de la cámara en el escenario.
tiempo	float	Almacena información de tiempo
lightPos	glm::vec3	Representa la posición de la luz en el espacio 3D
PosIni	glm::vec3	Representa la posición inicial en el escenario.
lightDirection	glm::vec3	Representa la dirección de la luz.
active	bool	Sirve para activar la animación de la lampara que se encuentra en el cuarto de Jimmy
animS1	bool	Controla la activación de la rotación negativa de la silla.
animS2	bool	Controla la activación de la rotación positiva de la silla.
rotS	float	Almacena el ángulo de rotación actual de la silla.
animS3	bool	Indica si se ha alcanzado una cierta posición en la rotación de la silla.
anim	bool	Controla la activación de la rotación negativa del avión.
animm	bool	Controla la activación de la rotación positiva del avión.
rot	float	Almacena el ángulo de rotación actual del avión.
animmm	bool	Indica si se ha alcanzado una cierta posición en la rotación del avión.
animC1	float	Controla la activación de la rotación negativa del cajón.
animC2	bool	Controla la activación de la rotación positiva del cajón.
transC	bool	Almacena la cantidad de desplazamiento actual del cajón.
animC3	float	Indica si se ha alcanzado una cierta posición en el desplazamiento del cajón.

animP1	bool	Controla la activación de la rotación negativa de la puerta.
animP2	bool	Controla la activación de la rotación positiva de la puerta.
rotP	float	Almacena el ángulo de rotación actual de la puerta.
animP3	bool	Indica si se ha alcanzado una cierta posición en la rotación de la puerta.
lastKeyPressTime	double	Almacena el momento del último evento de presionar una tecla.
doubleTapInterval	double	Representa el intervalo de tiempo permitido entre dos pulsaciones de tecla consecutivas para considerarlas como un "doble toque".
deltaTime	GLfloat	Almacena la diferencia de tiempo entre el fotograma actual y el fotograma anterior.
lastFrame	GLfloat	Almacena el tiempo del fotograma anterior.
pointLightPositions[]	glm::vec3	Almacena las posiciones de los distintos point lights, en total hay 4 vectores de posiciones guardados.
Light1	glm::vec3	Almacena el color de la lampara del cuarto de Jimmy
vertices[]	Glfloat	Establece las normales, las posiciones y las coordenadas de texturas que se asignarán al VBO.
skyboxVertices[]	Glfloat	Establece las posiciones que se asignarán al VBO del Skybox.
indices[]	Gluint	Contiene el indice de los vertices de 0 a 35
cubePositions[]	glm::vec3	Contiene la posicion de todos los containers
VAO	GLuint	Representa un objeto que encapsula los estados de configuración de los atributos de los vértices utilizados para renderizar geometría en OpenGL.
VBO	GLuint	Representa un objeto que almacena en memoria los datos de los vértices utilizados para renderizar geometría en OpenGL.

EBO	GLuint	Representa un objeto que almacena en memoria los índices de los vértices utilizados para renderizar geometría indexada en OpenGL.
lightVAO	GLuint	Representa un objeto que encapsula los estados de configuración de los atributos de los vértices utilizados para renderizar luces en OpenGL.
skyboxVBO	GLuint	Representa un objeto que almacena en memoria los datos de los vértices utilizados para renderizar el skybox en OpenGL.
skyboxVAO	GLuint	Representa un objeto que encapsula los estados de configuración de los atributos de los vértices utilizados para renderizar el skybox en OpenGL.
faces	vector<const GLchar*>	Vector que almacena punteros a cadenas de caracteres (const GLchar*) utilizados para representar las caras de un cubo en un cubemap.
cubemapTexture	GLuint	Representa una textura especial utilizada para mapear el entorno alrededor de una escena en un cubo tridimensional en OpenGL.
projection	glm::mat4	Matriz de proyección utilizada para transformar las coordenadas de los vértices desde el espacio de la cámara al espacio de la proyección en OpenGL. Es una matriz de 4x4.
currentFrame	GLfloat	Representa el tiempo transcurrido desde el inicio de la aplicación en OpenGL. Se utiliza para calcular la animación.
viewPosLoc	GLint	Usada para obtener la posición de la vista de la cámara.
modelLoc	GLint	Ubicación del uniform en el shader que representa la matriz de transformación del modelo
viewLoc	GLint	Ubicación del uniform en el shader que representa la matriz de transformación de la vista
projLoc	GLint	Ubicación del uniform en el shader que representa la matriz de proyeccion

model	glm::mat4	Matriz de transformación del modelo utilizada para transformar los vértices de un objeto desde el espacio del modelo al espacio de la vista en OpenGL. Es una matriz de 4x4.
view	glm::mat4	Matriz de transformación de la vista utilizada para transformar los vértices desde el espacio del mundo al espacio de la cámara en OpenGL. Es una matriz de 4x4 .
xOffset y yOffset	GLfloat	Usadas para obtener la diferencia entre la posición actual de la cámara y la anterior en el espacio x,y.

TECHNICAL GUIDE:

Development platforms

For the development of the environment, it was necessary to work under the Microsoft Visual Studio integrated development environment (IDE) which is used to develop computer programs, applications, and services, in my case, following the operations of OpenGL which is considered mainly an API (an application programming interface) which provides a large set of functions to manipulate graphics and images. However, OpenGL itself is not an API, but simply a specification, developed and maintained by the Khronos Group.

The OpenGL specification defines exactly what the result/output of each function should be and how it should work. Then, it is up to the developers to implement these specifications, to find a solution of how a function should operate. Since the OpenGL specification does not provide implementation details, the actual developed versions of OpenGL can have different implementations, as long as their results comply with the specification (and are therefore the same for users).

It is important to note that for the development of this project we worked with the OpenGL libraries which are written in C and allow many derivations in other languages, but in essence it is still a C library.

Since many of the C language constructs do not translate as well to other higher-level languages, OpenGL was developed with several abstractions in mind. One of those abstractions is objects in OpenGL.

Multiple objects were defined for this project to recreate Jimmy Neutron's house, following the functional requirements specifications in the previous section. An object in OpenGL is a collection of options that represents a subset of the OpenGL state. For example, we could have an object that represents the drawing window configuration; we could then set its size, how many colors it supports, etc. One could visualize an object as a C-like structure.

The good thing about using these objects is that you can define more than one object in the application, configure its options and every time you start an operation that uses the OpenGL state, you bind the object with your preferred configuration. There are objects, for example, that act as container objects for the 3D model data (a house or a character) and every time you want to

draw one of them, you link the object that contains the data of the model you want to draw (first you create and configure the options for these objects). Having several objects allows you to specify many models and whenever you want to draw a specific model, you simply bind the corresponding object before drawing without reconfiguring all its options.

2. Basic Environment Parameters

2.1. Using the Window

The first thing to configure, before creating the graphical environment, is an OpenGL context and an application window for drawing. However, these operations are operating system specific, and OpenGL tries to abstract from these operations on purpose. This means that you must create a window, define a context and handle user input manually. Fortunately, there are quite a few libraries that provide the necessary functionality, some specifically targeted at OpenGL. GLFW was used. GLFW is a library, written in C, specifically targeted at OpenGL. GLFW provides the basic needs required to display things on the screen. It allows creating an OpenGL context, defining window parameters and handling user input, which is sufficient for this purpose.

2.2. Linking

In order for the project to use GLFW, the library needs to be linked to the project. This could be achieved by specifying that you want to use glfw3.lib in the linker configuration, but the project still needs to know where to find glfw3.lib since these third-party libraries are located in a different directory. Therefore, this directory must first be added to the project.

A dynamic path can be specified to the IDE to recognize all the files, the paths are as follows.

```
$(SolutionDir)/External
Libraries/GLEW/lib/Release/Win32

$(SolutionDir)/External
Libraries/GLFW/lib-vc2015

$(SolutionDir)/External
Libraries/SOIL2/lib

$(SolutionDir)/External
Libraries/assimp/lib
```

2.2.1 External Libraries Included.

So that the project could know where to look for the necessary tools, the appropriate location string was manually inserted, the IDE will also look in those directories when searching for libraries and header files. As soon as folders such as GLFW were included, all header files for GLFW could be found by including <GLFW/..>., etc. The Library list includes:

```
$(SolutionDir)/External
Libraries/GLEW/include
```

```
$(SolutionDir)/External
Libraries/GLFW/include

$(SolutionDir)/External Libraries/glm

$(SolutionDir)/External
Libraries/assimp/include
```

2.2.2 Linker Input Parameters.

Once the external headers were set, in the IDE, VisualStudio can find all the necessary files, and finally you can link the libraries to the project by going to the Linker > Input tab to finish our configuration.

```
soil2-debug.lib;assimp-vc140-

mt.lib;opengl32.lib;glew32.lib;glfw3.
lib;
```

3 Shaders

Shaders were used to help build the aquarium environment of this project, Shaders are small programs that rest on the GPU.

These programs run for each specific section of the graphics pipeline. In a basic sense, shaders are nothing more than programs that transform inputs into outputs. Shaders are also very isolated programs in the sense that they are not allowed to communicate with each other; the only communication they have is through their inputs and outputs.

The shaders used are written in the C-like GLSL language. GLSL is designed for use with graphics and contains useful features specifically aimed at manipulating vectors and matrices.

Shaders always begin with a version statement, followed by a list of input and output variables, uniforms, and their main function.

The entry point of each shader is in its main function where any input variables are processed and the results are displayed in its output variables.

Shaders were used for model loading, lighting, cubemaps, and room animations.

They are listed in a specific folder called Shaders, and their programming is native to their function.

4. Loading Models

4.1 ASSIMP

For our environment it is not possible to manually define all vertices, normals and texture coordinates of complicated shapes.

Instead, we made use of 3D modeling tools, such as Maya, to create complicated shapes and apply textures to them through UV mapping. Then, the tools automatically generate all vertex coordinates, vertex normals and texture coordinates while exporting them to a model file format that we can use (OBJ). In this way, you have an extensive set of

tools to create high quality models without having to worry too much about the technical details. All technical aspects are hidden in the exported model file. However, as graphics programmers, we have to worry about these technical details.

The model import library used is Assimp, which stands for Open Asset Import Library. Assimp can import dozens of different model file formats (and export to some as well) by loading all the model data into Assimp's generalized data structures. As soon as Assimp has loaded the model, we can retrieve all the data we need from Assimp's data structures. Because the Assimp data structure remains the same regardless of the type of file format that is imported, Assimp abstracts from all the different file formats that exist.

When importing a model through Assimp, the entire model is loaded into a scene object that contains all of the imported model/scene data. Assimp then has a collection of nodes where each node contains indexes of data stored in the scene object where each node can have any number of children.

4.2 Mesh

With Assimp we were able to load many different models into the application, but once loaded, they are all stored in Assimp's data structures. What is eventually needed is to transform that data into a format that OpenGL understands so that we can render the objects. A mesh should need at least one set of vertices, where each vertex contains a position vector, a normal vector, and a texture coordinate vector. A mesh should also contain indexes for indexed drawing and material data in the form of textures (fuzzy/specular maps).

Thanks to the builder, you get large lists of mesh data that can be used for rendering. You need to set up the appropriate buffers and specify the vertex shader layout through vertex attribute pointers so that finally with the last function that needs to be defined for the Mesh class to be complete is its 'Draw' function. Before rendering the mesh, the appropriate textures must first be bound to them before calling glDrawElements. However, this is difficult since the existence and number of textures in the mesh and the type is unknown.

4.3 Model

From this point in the project, we start creating the actual translation and loading code with Assimp. The goal is to create another class that represents a model in its entirety, that is, a model containing multiple meshes, possibly with multiple

textures. A window that has frame, trim and glass could still be loaded as a single model. The model is loaded through Assimp and translated into the various mesh objects we have created.

Keeping in mind that the texture file paths in the model files are assumed to be local to the actual model object, Ex. in the same directory as the location of the model itself.

Then one can simply concatenate the texture location string and the directory string that was retrieved earlier (in the loadModel function) to get the full texture path (that is why the GetTexture function also needs the directory string).

Some models that were obtained from the Internet for this project used absolute paths for their texture locations, which didn't work by the time we got to this point. In that case, we manually edited the file to use local paths for the textures (if possible). Or the textures were replaced and a process of adaptation to the model described below was started

2.1 TECHNICAL GUIDE:

ELABORATION AND ADAPTATION PROCESS

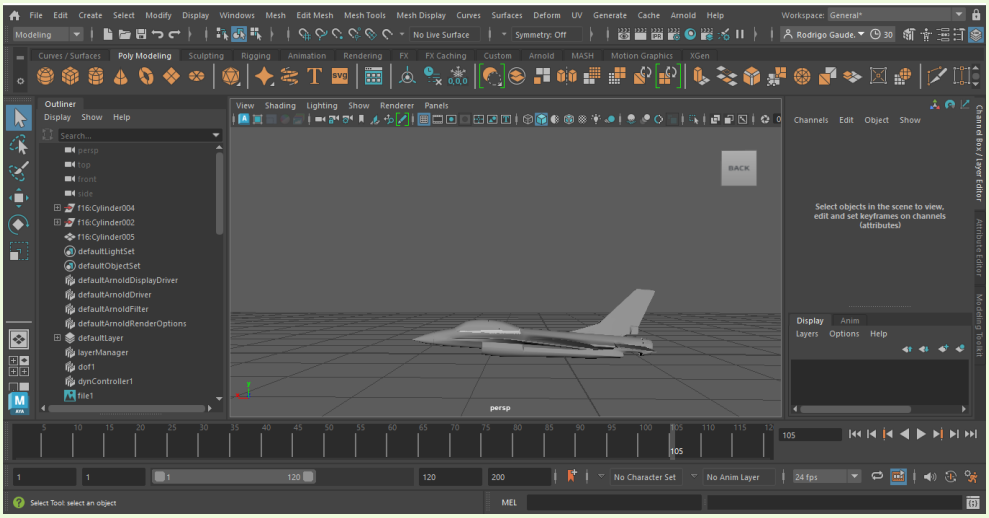
Based on the schedule of activities and the completion of the Planning and Design phase, the next phase was the Implementation phase where inputs, changes, and elimination of elements were added to adjust to the delivery agreed upon in the requirements analysis.

Among these items, processes for adding objects were defined.

1. Import or creation of the model.

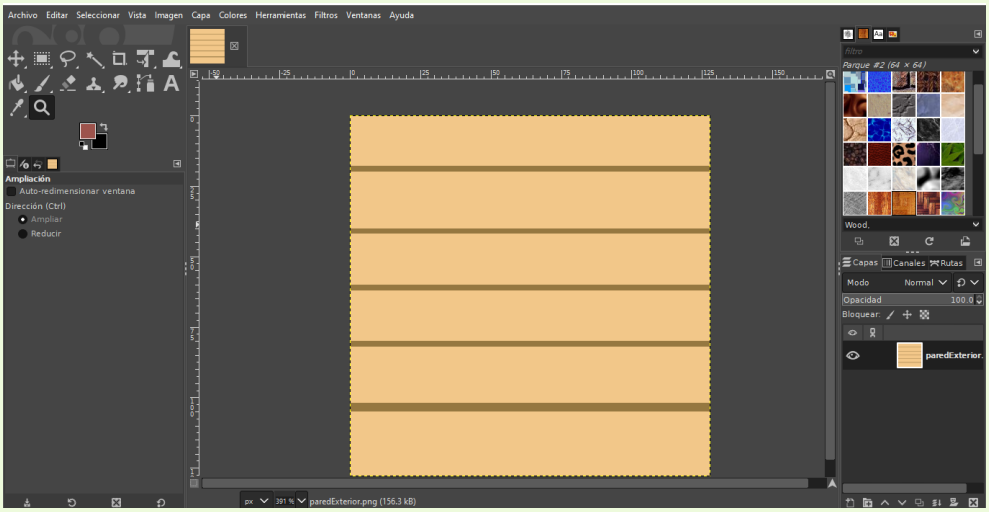
Most of the models of the project are our own intellectual material, but there are models, or parts thereof, that were obtained from platforms that offer various graphic models with different licenses for use.

The models obtained from free licenses did not contain textures or materials. For this point, parameters were established for their integration within the scenario.



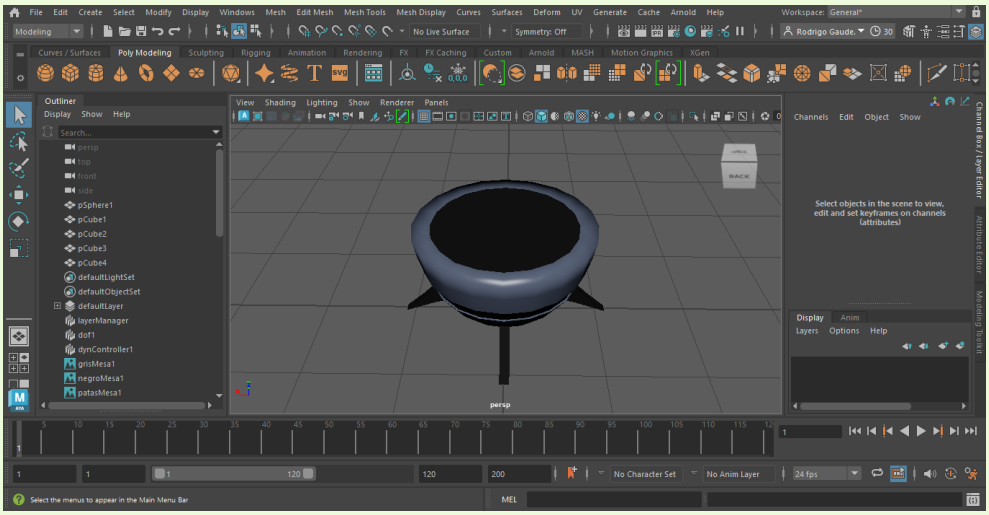
2. Texture selection and material adaptation.

The selected models, which were filtered for the final phase of introduction to the scenario, were reviewed for texture acquisition. The images were worked with the GIMP software, to be used as texture and to mold the uv map. To create a better ambience to the environment.



3. Convergence of materials and completion of modeling

The materials are coupled to the model, basic transformations of the object are reviewed again to adapt its position within the scenario and finally, it is imported for its later incorporation to the integrated development environment that works the complete integration of the project.



4. Integration of the model to the final scenario

Finally, the workspace is exchanged to continue with the integration and coupling to the final deliverable. It is ensured that the new model does not affect the stability of the project, that there is no undesired positioning, or that the integration of the component is as desired.

5. Version Control

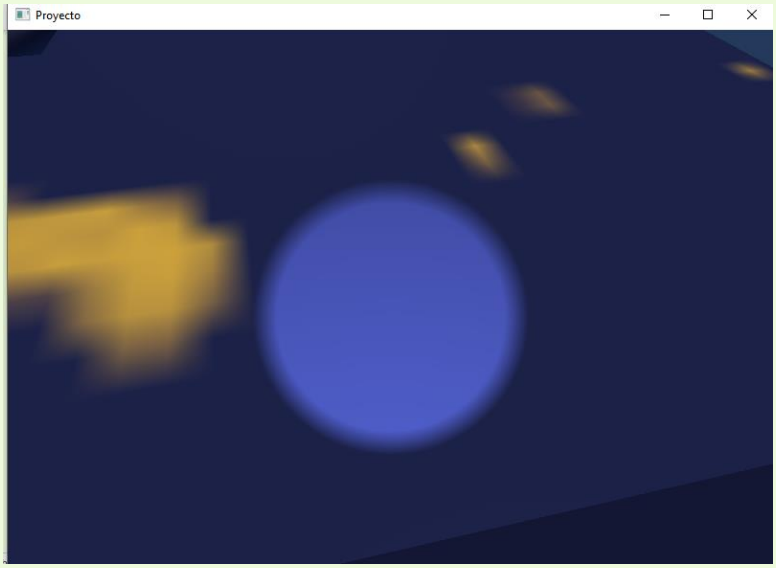
New changes are saved and updated in the central repository; comments are contemplated in the documentation as the change made and the functionality.

The GitHub collaborative tool is used to establish a version control parameter, backup and contiguous work.

2.2 TECHNICAL GUIDE: USE OF AMBIANCE: LIGHTING

For this section we used the spotlight in the position of the synthetic camera so that the user would look as if he was carrying a hand lamp.

```
// 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"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.diffuse"), 1.0f, 1.0f, 1.0f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "spotLight.specular"), 1.0f, 1.0f, 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.constant"), 1.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.linear"), 0.14f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.quadratic"), 0.07f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.cutOff"), glm::cos(glm::radians(12.5f)));
glUniform1f(glGetUniformLocation(lightingShader.Program, "spotLight.outerCutOff"), glm::cos(glm::radians(15.0f)));
```



Only one pointlight was used, although they were not necessary, the other three were left as they do not affect, however they could have been removed from the shader and then from the source code.

```
// Directional light
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.direction"), -0.2f, -1.0f, -0.3f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.ambient"), 0.5f, 0.5f, 0.5f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.diffuse"), 0.4f, 0.4f, 0.4f);
glUniform3f(glGetUniformLocation(lightingShader.Program, "dirLight.specular"), 0.5f, 0.5f, 0.5f);



// Point light 1
glm::vec3 lightColor;
//Para encender y apagar la lampara
lightColor.x = abs(sin(glfwGetTime() * Light1.x));
lightColor.y = abs(sin(glfwGetTime() * Light1.y));
lightColor.z = abs(sin(glfwGetTime() * Light1.z));

glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].position"), pointLightPositions[0].x, pointLightPositions[0].y, pointLightPositions[0
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].ambient"), lightColor.x, lightColor.y, lightColor.z);
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].diffuse"), lightColor.x, lightColor.y, lightColor.z);
//Specula es el punto brilloso que se ve al rebotar la luz en una superficie
glUniform3f(glGetUniformLocation(lightingShader.Program, "pointLights[0].specular"), 1.0f, 1.0f, 0.0f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].constant"), 1.0f);
//Definen que tan brillante y grande
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].linear"), 0.7f);
glUniform1f(glGetUniformLocation(lightingShader.Program, "pointLights[0].quadratic"), 1.8f);
```

The illumination was left so that it would be in accordance with the environment, being daytime.

2.3 TECHNICAL GUIDE: USE OF ANIMATIONS

For the animations we used two shaders that provide movement according to a chosen function.

For the movement of the top of the tree a function was selected and the application of the effect in certain specific axes to generate an oscillatory movement so that it seems to move when the air hits it and returns, this with a small amplitude so that the movement is smooth.

```
const float amplitude = 0.5;
const float frequency = 0.05;
const float PI = 3.14159;
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform float time;

void main()
{
  float distance = length(aPos);
  float effect = amplitude*sin(-PI*distance*frequency+time);
  gl_Position = projection*view*model*vec4(aPos.x+effect,aPos.y, aPos.z+effect,1);
  TexCoords=vec2(aTexCoords.x,aTexCoords.y);

}
```

In the case of the animation of the Alien, other parameters and application of the effect were chosen so that the object deforms a little, since the Alien, not being in its armor, is not able to maintain its shape and looks like a gelatinous form.

```
const float amplitude = 0.15;
const float frequency = 4.0;
const float PI = 3.14159;
out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform float time;

void main()
{
  float distance = length(aPos);
  float effect = amplitude*sin(-PI*distance*frequency+time);
  gl_Position = projection*view*model*vec4(aPos.x+effect,aPos.y*effect, aPos.z+effect,1);
  TexCoords=vec2(aTexCoords.x,aTexCoords.y);

}
```

For the rest of the animation, what was done was to create Boolean variables that activate the animation and stop it at a certain moment. These lines were added in the functions main(), KeyCallback() and DoMovement().Here are some images regarding the declaration of variables and their usage for animation in the previously mentioned functions. Here is the dictionary of functions and variables for more details.

2.4 TECHNICAL GUIDE: FUNCTION DICTIONARY

Function	Details

Main()	Main function where the creation of the window, VAO, VBO and EBO objects takes place, along with transformation matrices for the shader, camera management, loading shaders, loading and drawing models, handling lighting and creating the skybox.
KeyCallBack(GLFWwindow *window, int key, int scancode, int action, int mode)	This is a callback function that is triggered when a keyboard event occurs. It takes the GLFW window as parameter (window), the key code (key), the scancode, the action performed (action) and the input mode (mode). This function is used to process and react to keyboard events such as pressing, releasing or holding down a specific key. For example, when the ESC key is pressed, the window is closed.
MouseCallBack(GLFWwindow *window, double xPos, double yPos)	This is a callback function that is triggered when a mouse event occurs. It takes the GLFW window as parameter (window) and the coordinates of the mouse pointer (xPos and yPos). This function is used to process and react to mouse events, such as cursor movements.
DoMovement()	It manages movement within the interactive environment. It allows controlling the position and orientation of an object or camera in response to input events such as pressing buttons or moving the mouse.

2.5 TECHNICAL GUIDE: DICTIONARY OF VARIABLES

Name	Tipo	Descripción
WIDTH	const Gluint	Define the width of the display window.
HEIGHT	const Gluint	Define the height of the display window.
SCREEN_WIDTH y SCREEN_HEIGHT	int	Represents the width and height of the screen on which the application is running.

camera	Camera	It is an object of the Camera class that represents the camera in 3D space. It is initialized with an initial position.
lastX y lastY	Glfloat	Stores the last coordinates of the mouse pointer.
keys[1024]	bool	Used to track the state of the keyboard keys.
firstMouse	bool	Indicates if it is the first time the mouse is being moved. Used to process mouse movements correctly.
range y movCamera	float	Used to control the camera's range of movement in the scene.
tiempo	float	Stores time information.
lightPos	glm::vec3	Represents the position of the light in 3D space.
PosIni	glm::vec3	Represents the initial position in the scene.
lightDirection	glm::vec3	Represents the direction of the light.
active	bool	Used to activate the animation of the lamp in Jimmy's room.
animS1	bool	Controls the activation of the negative rotation of the chair.
animS2	bool	Controls the activation of the positive rotation of the chair.
rotS	float	Stores the current rotation angle of the chair.
animS3	bool	Indicates whether a certain position in the chair rotation has been reached.
anim	bool	Controls the activation of the negative rotation for the plane.
animm	bool	Controls the activation of the positive rotation for the plane.
rot	float	Saves the current rotation angle of the plane.
animmm	bool	Indicates whether a certain position in the plane rotation has been reached.
animC1	float	Controls the activation of the negative rotation for the drawer.
animC2	bool	Controls the activation of the positive rotation for the drawer.

transC	bool	Saves the current displacement amount of the drawer.
animC3	float	Indicates whether a certain position in the drawer displacement has been reached.
animP1	bool	Controls the activation of the negative rotation for the door.
animP2	bool	Controls the activation of the positive rotation for the door.
rotP	float	Saves the current rotation angle of the door.
animP3	bool	Indicates whether a certain position in the door rotation has been reached.
lastKeyPressTime	double	Stores the time of the last key press event.
doubleTapInterval	double	Represents the time interval allowed between two consecutive key presses to consider them as "double taps".
deltaTime	GLfloat	Stores the time difference between the current image and the previous image.
lastFrame	GLfloat	Stores the time of the previous image.
pointLightPositions[]	glm::vec3	Stores the positions of the various point lights. There are a total of 4 stored position vectors.
Light1	glm::vec3	Save the color of the lamp in Jimmy's room.
vertices[]	Glfloat	Set the normals, positions, and texture coordinates to be assigned to the VBO.
skyboxVertices[]	Glfloat	Set the positions to be assigned to the Skybox VBO.
indices[]	Gluint	Contains the vertex indices from 0 to 35.
cubePositions[]	glm::vec3	Contains the positions of all containers.
VAO	GLuint	Represents an object that encapsulates the configuration states of vertex attributes used to render geometry in OpenGL.
VBO	GLuint	Represents an object that stores the vertex data used for rendering geometry in OpenGL.

EBO	GLuint	Represents an object that stores the vertex indices used for rendering indexed geometry in OpenGL.
lightVAO	GLuint	Represent an object that encapsulates the configuration states of vertex attributes used for rendering highlights in OpenGL.
skyboxVBO	GLuint	Represent an object that stores the vertex data used for rendering skybox in OpenGL.
skyboxVAO	GLuint	Represents an object that encapsulates the configuration states of vertex attributes used to render the skybox in OpenGL.
faces	vector<const GLchar*>	Vector that stores pointers to strings (const GLchar*) used to represent the faces of a cube in a cubemap.
cubemapTexture	GLuint	Represents a special texture used to map the environment of a scene onto a 3D cube in OpenGL.
projection	glm::mat4	Projection matrix used to transform vertex coordinates from camera space to projection space in OpenGL. It is a 4x4 matrix.
currentFrame	GLfloat	Represents the elapsed time since the application was started in OpenGL. It is used for animation calculations.
viewPosLoc	GLint	Used to get the view position of the camera.
modelLoc	GLint	Position of the uniform in the shader that represents the model transformation matrix.
viewLoc	GLint	Position of the uniform in the shader that represents the view transformation matrix.
projLoc	GLint	Position of the uniform in the shader representing the projection matrix.
model	glm::mat4	Model transformation matrix used to transform object vertices from model space to view space in OpenGL. It is a 4x4 matrix.

view	glm::mat4	View transformation matrix used to transform vertices from space to camera space in OpenGL. It is a 4x4 matrix.
xOffset y yOffset	GLfloat	Used to find the difference between the current and the previous camera position in x and y space.

CONCLUSIONES

Inicialmente, consideré este proyecto como algo sencillo, y de hecho lo es, pero solo hasta luego de haberlo hecho lo considero así. Sin embargo, me encontré con la dificultad de tener que aprender a modelar y editar imágenes para recrear los objetos, así como mi propia fachada, y texturizarlos de la manera más adecuada posible para lograr el ambiente deseado, tal como se muestra en mis imágenes de referencia. Fue un proceso acumulativo que, de no haberlo abordado por etapas, no habría sido posible llevar a cabo con éxito.

Este proyecto resultó ser estresante, pero al mismo tiempo me brindaba una gran satisfacción cada vez que veía cómo mis objetos seleccionados interactuaban en conjunto en el ambiente, junto con las animaciones. Fue un desafío pensar en las animaciones complejas que iba a utilizar, ya que, a diferencia de las simples, tuve que crear modelos adicionales para poder añadirles animaciones más complejas. Creo que esto proporciona un efecto que se ajusta al ambiente y cumple con el aspecto de animación compleja de manera exitosa.

RESULTS

Originally I thought this project was simple, and it is, but it is only after its completion that I see it that way. The difficulty, however, was that I had to learn how to model and manipulate images to recreate objects as well as my own facade and texture them in the most appropriate way to achieve the desired environment as depicted in my reference images. It was a cumulative process that could not have been successfully accomplished without a step-by-step approach.

This project was exhausting, but at the same time, I was very pleased every time I saw how the objects I selected worked together with the animations in the environment. It was a challenge to think about the complex animations I wanted to use, because unlike the simple ones, I had to create additional models to add more complex animations to them. I think this gives an effect that fits the environment and successfully implements the complex animation aspect.

Documentación/Documentation

General Terms and Conditions | CGTrader. (s. f.). CGTrader. https://www.cgtrader.com/pages/terms-and-conditions

Ghogen. (2023, 18 abril). Tutorial: Compilación de una aplicación - Visual Studio (Windows). Microsoft Learn. https://learn.microsoft.com/es-es/visualstudio/ide/walkthrough-building-an-application?view=vs-2022

Using the TurboSquid Royalty Free License - TurboSquid Blog. (2023, 16 marzo). TurboSquid Blog. https://blog.turbosquid.com/royalty-free-license/

Términos y condiciones para los modelos de TurboSquid

Royalty Free License

This is a legally binding agreement between licensee (“you”), and TurboSquid regarding your rights to use Stock Media Products from the Site under this license. “You” refers to the purchasing entity, whether that is a natural person who must be at least 18 years of age, or a corporate entity. The rights granted in this agreement are granted to the purchasing entity, its parent company, and its majority owned affiliates on a “royalty free” basis, which means that after a Purchase, there are no future royalties or payments that are required. Collectively, these rights are considered “extended uses”, and are granted to you, subject to applicable Editorial Use Restrictions described below. The license granted is wholly transferable to other parties so long it is in force and not terminated, otherwise violated, or

extinguished, as set forth herein. This agreement incorporates by reference the Terms of Use as well as the Site's policies and procedures as such.

Creations of Imagery.

Permitted Uses of Creations of Imagery. Subject to the following restrictions, you may use Creations of Imagery within news, film, movies, television programs, video projects, multi-media projects, theatrical display, software user interfaces; architectural renderings, Computer Games, virtual worlds, simulation and training environments; corporate communications, marketing collateral, tradeshow promotional items, booth decorations and presentations; pre-visualizations, product prototyping and research; mobile, web, print, television, and billboard advertising; online and electronic publications of blogs, literature, social media, and email campaigns; website designs and layouts, desktop and mobile wallpapers, screensavers, toolbar skins; books, magazines, posters, greeting cards; apparel items, brochures, framed or printed artwork, household items, office items, lenticular prints, product packaging and manufactured products.

Restrictions on Permitted Uses of Creations of Imagery.

a. Stock Media Clearinghouse. You may NOT publish or distribute Creations of Imagery through another stock media clearinghouse, for example as part of an online marketplace for photography, clip art, video, or design templates.

b. Promotional Images. Images displayed for the promotion of Stock Media Products, such as preview images on the Stock Media Product's Product Page ("Promotional Images"), may be used in Creations of Imagery, provided that the Stock Media Product itself has been Purchased and subject to the following restrictions:

i. You may NOT use a Promotional Image that has any added element which is not included as part of the Stock Media Product. An example of this type of restricted use is if the Stock Media Product contains a 3D model of an airplane, and there is a Promotional Image of that airplane rendered over a blue sky; however, the blue sky image is not included as part of the Stock Media Product. Other prohibited examples include use of Promotional Images from movies or advertisements that may have used Stock Media Product.

ii. You may NOT use any Promotional Image that has a logo, mark, watermark, attribution, copyright or other notice superimposed on the image without prior approval from TurboSquid Support.

c. Business Logos. You may NOT use Imagery in any Creation that is a trademark, servicemark, or business logo. This restriction is included because the owners of these types of Creations typically seek exclusivity on the use of the imagery in their Creation, which is incompatible with the non-exclusive license granted to you under this agreement.

Creations of Computer Games and Software

Permitted Uses in Creations of Computer Games and Software. Subject to the following restrictions, you may include Stock Media Products in Creations of Computer Games, virtual worlds, simulation and training environments; mobile, desktop and web applications; and interactive electronic publications of literature such as e-books and electronic textbooks.

Restrictions on Permitted Uses of Stock Media Products in Creations of Games and Software.

a. Interactivity. Your inclusion of Stock Media Products within any such Creation is limited to uses where Stock Media Product is contained in an interactive experience for the user and not made available outside of the interactive experience. Such a permitted example of this use would be to include a 3D model of human anatomy in a medical training application, in a way that the 3D model or its environment may be manipulated or interacted with.

b. Access to Stock Media Products. You must take all reasonable and industry standard measures to prevent other parties from gaining access to Stock Media Products. Stock Media Products must be contained in proprietary formats so that they cannot be opened or imported in a publicly available software application or framework, or extracted without reverse engineering. WebGL exports from Unity, Unreal, Lumberyard, and Stingray are permitted. Any other open format or format encrypted with decryptable open standards (such as an encrypted compression archive or other WebGL programs not listed here) are prohibited from using Stock Media Products. If your Creation uses WebGL and you are not sure if it qualifies, please contact use@turbosquid.com and describe your Creation in detail.

Términos y condiciones para los modelos de CGTrader

Royalty Free LicenseEC

If the model is under Royalty Free license, you can use it as long as it is incorporated into the product and as long as the 3rd party cannot retrieve it on its own in both digital and physical form. You cannot resell the model you bought in its digital form and you cannot resell it in its printed form as a separate, single item.

Product may not be sold, given, or assigned to another person or entity in the form it is downloaded from the Site.

The Buyer's license to Product in this paragraph is strictly limited to Incorporated Product. Any use or republication, including sale or distribution of Product that is not Incorporated Product is strictly prohibited. For illustration, approved distribution or use of Product as Incorporated Product includes, but is not limited to:

- as rendered still images or moving images; resold as part of a feature film, broadcast, or stock photography;
- as purchased by a game's creators as part of a game if the Product is contained inside a proprietary format and displays inside the game during play, but not for users to re-package as goods distributed or sold inside a virtual world;
- as Product published within a book, poster, t-shirt or other item;
- as part of a physical object such as a toy, doll, or model.

If you use any Product in software products (such as video games, simulations, or VR-worlds) you must take all reasonable measures to prevent the end user from gaining access to the Product. Methods of safeguarding the Product include but are not limited to:

- using a proprietary disc format such as Xbox 360, Playstation 3, etc.;
- using a proprietary Product format;
- using a proprietary and/or password protected database or resource file that stores the Product data;

encrypting the Product data.

Without prejudice the information above, the Seller grants to the Buyer who purchases license rights to Product and uses it solely as Incorporated Product a non-exclusive, worldwide, license in any medium now known or hereinafter invented to:

reproduce, post, promote, license, sell, publicly perform, publicly display, digitally perform, or transmit for promotional and commercial purposes

use any trademarks, service marks or trade names incorporated in the Product in connection with Seller material;

use the name and likeness of any individuals represented in the Product only in connection with Your material.

The resale or redistribution by the Buyer of any Product, obtained from the Site is expressly prohibited unless it is an Incorporated Product as licensed above.

We also always suggest buyers discuss the usage with the seller and gain their approval, this would make you a hundred percent sure that you can proceed with your decision of using the purchase.

Editorial License

Editorial license gives a permission to use the product only in an editorial manner, relating to events that are newsworthy, or of public interest, and may not be used for any commercial, promotional, advertising or merchandising use.

In a few instances, you may otherwise have the rights to IP in content that is under Editorial license. For instance, you may be the advertising agency for a brand/IP owner or you may be the brand/IP owner itself purchasing user generated content. Given you have the rights clearance through other means, you may use the content under Editorial License commercially. Every user failing to comply with Editorial Use restrictions takes the responsibility to prove the ownership of IP rights.

For users, who are not brand/IP owners or official affiliates, the restrictions of Editorial-licensed content usage include, but are not limited to, the following cases:

Products may not be used on any item/product created for resale such as, commercials, for-profit animations, video games, VR/AR applications, or physical products such as a merchandise or t-shirt.

Products may not be used as part of own product promotional materials, billboard, trade show or exhibit display.

The product may not be incorporated into a logo, trademark or service mark. As an example, you cannot use Editorial content to create a logo design.

Products may not be used in any insulting, abusive or otherwise unlawful manner.

The product may not be used for any commercial related purpose.