UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

PROYECTO FINAL

Videojuego: Cute Land

COMPUTACIÓN GRÁFICA AVANZADA

GRUPO:1 **CLAVE:2948**

EQUIPO:

- **ARGUELLES MACOSAY MARIANA**
- **MARTÍNEZ SANTANA DIANA ANAYANSSI**

FECHA DE ENTREGA:

10 FEBRERO 2021

1. Introducción

La computación gráfica es considerada como uno de los campos de mayor crecimiento y de mayor valor agregado dentro del Cómputo, la informática y tecnologías de la información. Ofrece una amplia diversidad de opciones para el desarrollo tanto profesional como académico.
Para la realización del proyecto final es importante recalcar que se necesita el uso de otros conceptos adquiridos a lo largo de l a carrera, como lo son el álgebra Lineal y geometría Analítica, así como de habilidades desarrolladas y conocimientos de programación, principalmente en lenguaje C, C++ y C#, entre otros.

En el presente documento se presentará el proyecto del videojuego Cute Land, siendo un shooter con escenario y personajes tiernos, empleando modelos de internet así como creados desde cero.
El juego se centra en nuestra personaje principal MayCute, la cual debe eliminar las redBalls que amenazan con dañar el pueblo de Cute Land, trayendo paz y armonía de nuevo al lugar.

2. Objetivo

El equipo demostrará los conocimientos adquiridos a lo largo del curso implementando lo visto en las diferentes prácticas(como el modelado geométrico, así como en colores e iluminación, sombras, animaciones, colisiones, entre otras) para la creación de un videojuego que se desarrolle en el ambiente de OpenGL, de tal forma que los objetivos principales son:

- ★ Elaborar un proyecto haciendo uso del curso de computación gráfica avanzada.
- ★ Hacer uso de elementos aprendidos en las diferentes prácticas realizadas.
- ★ Hacer uso de herramientas como gimp, Blender y Mixamo.
- ★ Implementar diferentes luces y cámaras para el juego.

3. Hipótesis

Se demostrarán los conocimientos adquiridos a lo largo del curso así como también implementaciones investigadas a través de internet, como el uso de un mando de consola(PlayStation, Xbox, entre otros.).
El videojuego Cute Land contará con un escenario repleto de modelos relacionados con dulces o postres, donde el modelo del personaje principal contará con diferentes animaciones para las acciones como disparar, victoria, reposo y caminar. Las redball serán los enemigos principales que podrán dañar al personaje principal y serán eliminados creando una fuente de partículas como resultado.

4. Análisis

Para la creación de un proyecto interactivo y atractivo para el público en general se deben de tomar en cuenta muchas variables por lo que nos centramos en la facilidad de interacción dentro del juego para mejorar la experiencia del usuario y una curva de aprendizaje corta, la operación de los controles y teclas para el movimiento del personaje principal se programó de forma conjunta por lo que se puede hacer uso de un mando de consola de videojuego o el uso de teclas del teclado.

El tamaño y volumen de los archivos, texturas e imágenes se trató de mantener al mínimo para mejorar el rendimiento de la tarjeta gráfica, en cuanto a los modelos se crearon o modificaron para tener la menor cantidad de vertical posibles y que sus texturas sean lo más chicas posibles para un mejor renderizando.

El renderizado iterativo es crucial para los modelos del mismo tipo que se utilizaron en el mapa, por ejemplo los bastones de luz que se decidieron usar como luminarias.

Para el terreno se decidió que el personaje podría andar sin restricción alguna a excepción de haber colisionado, se contempló el uso y cambio a un segundo personaje el cual no tiene restricción alguna en el mapa, ni siquiera con objetos que tienen el colisionador implementado, esto para mejorar la jugabilidad en caso de encontrarte acorralado pon el enemigo y en ese caso podría escapar atravesando los objetos en un tiempo determinado.

Se necesitaba la implementación de un lugar seguro por lo que se decidió la designación de atrás específicas en donde los enemigos no tuvieran acceso e incluso mantenerse a distancia, por lo que la implementación de casos y estados es necesaria.

La lógica del juego fue planteada con detenimiento para crear un tipo de adictividad al juego con el fin de hacer que el usuario sienta felicidad al momento de obtener puntuaciones altas o de obtener cierta cantidad de puntuación o experiencia al eliminar un objetivo o pasar a la siguiente ronda, por lo que se necesita un sistema de puntuaciones en dónde se guarden los valores de la puntuación.

Mostrar la puntuación en pantalla, estos e planea lograr con una textura y una máscara de textura ubicada frente a la cámara, con su respectivo valor alfa para la transparencia.

Para mostrar que has eliminado al enemigo con éxito se planeó un sistema de partículas que se renderiza únicamente cuando se ha hecho una colisión al objetivo, en este caso una colisión rayo-esfera, en donde desde la posición de el personaje será dirigido un rayo hacia un objetivo y en caso de colisionar el objeto dejará de ser renderizado y un sistema de partículas será renderizado en la misma posición.

Para el ambiente se determinó que debía de ser del agrado al público por lo que se optó por una temática linda y adorable, de esta manera mantenemos unos gráficos y la temática agradables realizamos nuestro propio skybox y el cambio del skybox a uno nocturno pero lindo de todos modos por lo que se necesita la implementación de un contador con respecto al tiempo para el cambio entre skybox y la simulación de un ciclo de día y noche, además se debe realizar un cambio en el valor de la luz direccional y ambiental que da la ambientación de luminosidad al mapa o la falta de esta.

Se debe crear la sensación de altitud en el mapa, es decir, que el terreno posea desniveles y para ello los personajes y objetos deben de encontrarse arriba de dichos niveles, sin embargo los objetos grandes se renderizan desde su pivote y esto origina una diferencia de niveles ya que la orientación del objeto con respecto s los ejes X y Z no se cambian, por lo que se decidió que para simular un ambiente más creíble y visualmente atractivo el terreno debía ser "suave" permitiendo parecer

a los objetos situados en las colinas que están parcialmente enterrados en el terreno, por lo que dedico que texturizado el suelo como nubes , algodón de azúcar o nieve(helado) eran las mejores opciones.

Aleatoriedad en posiciones del renderizado de los enemigos es crucial para el juego y de esa forma las rondas nunca serán iguales, que la dificultad se mantiene por nivel de manera que la cantidad de enemigos a renderizar dependerá de la ronda o nivel en que se encuentre por lo que se traduce en el aumento de la dificultad.

La ambientación auditiva también es importante por lo que se decidió tenér música de ambientación a lo largo de todo el mapa por los que la música se escuchará siempre y uniforme en el mapa, los efectos de sonido de las diferentes acciones como los disparos o la explosión de las pelotas o el sonido de rebote serán implementados con respecto a las posiciones de los objetos que lleven dicha animación o sonido, por lo que el uso de una librería de sonido y un editor de sonido son necesarios para la realización del proyecto.

La implementación del buffer de profundidad y de la aplicación de los shaders a cada uno de los objetos por modelo a renderizar es importante para las sombras que se deberán ver en el mapa con el fin de crear una sensación de realidad y profundidad en el mapa.

5. PLAN DE TRABAJO

Bosquejos

Realizamos los bocetos principales del videojuego a través de dibujos digitales; entre los que creamos fueron los bosquejos del escenario, el de los personajes principales y de armas, como se muestra a continuación:

Escenario

Para el bosquejo del escenario(véase Figura 1.), decidimos utilizar un concepto de dulces, colores pasteles y cosas que se consideran “tiernas”, tanto en el terreno como en el Skybox y los modelos que se verán alrededor de estos.

Figura 1. Bosquejo de escenario

Personajes

Para el bosquejo de los personajes (véase Figura 2.), inicialmente pensamos en utilizar 3 diferentes modelos(un personaje femenino de estilo tierno, un conejo con un pequeño moño y un fantasma con transparencia), sin embargo al final solo optamos por el personaje femenino y el fantasma como segundo personaje el cual tendrá un efecto diferente en el mapa.

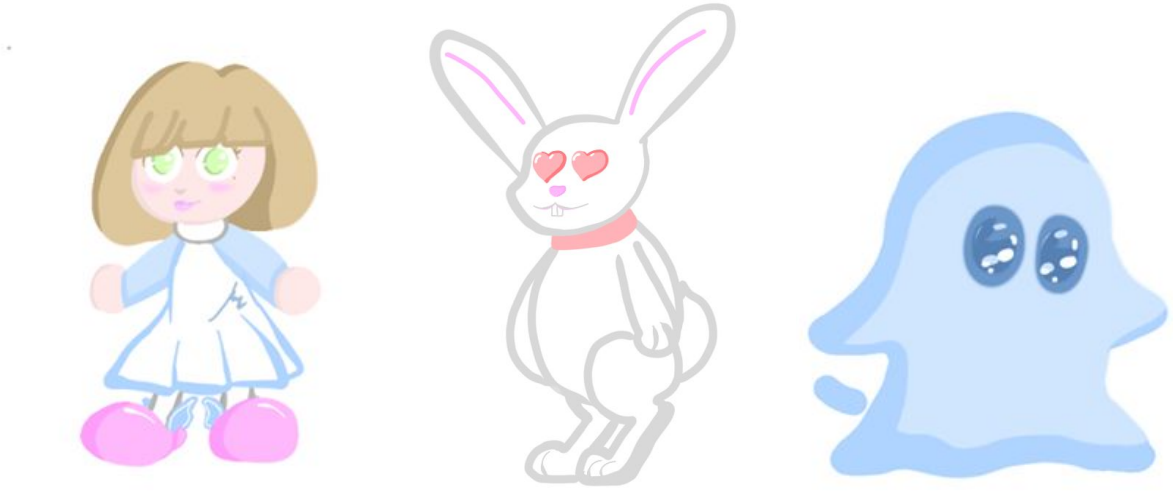

Figura 2. Bosquejo de personajes

Arma

Para el bosquejo del arma (véase Figura 3.), decidimos hacerlo con un diseño de colores pasteles con un diseño pequeño, como el de una pistola espacial. Como idea original teníamos pensado que fuera una pistola de agua, sin embargo después pensamos que sería menor que fuera de un rayo de colores, sin embargo por el tiempo el uso de esta pistola quedó detenido hasta arreglar a primera instancia el control de las pelotas y su movimiento hacia el personaje principal.

Figura 3. Bosquejo de arma

Cronogramas de actividades

Para llevar un control de las actividades que realizamos a lo largo de estos meses se creó un cronograma de actividades, de esta forma se presenta a continuación las actividades realizadas a través de los meses(Véase Tabla 1, Tabla 2 y Tabla 3.):

Noviembre			Diciembre		
Sem3	sem4	sem5	Sem1	Sem2	Sem3
Realizar el planteamiento del proyecto.	Realizar modificaciones y recomendaciones.	Búsqueda de modelos o realización. Búsqueda de texturas y búsqueda de skybox	Creación de escenario(Mapa de alturas y de texturas) y luces	Creación de animaciones de los diferentes modelos,	Importación de modelos, animaciones, mapas, texturas, etc. en el proyecto

Tabla 1. Cronograma noviembre a diciembre

Diciembre		Enero			
Sem4	sem5	sem1	Sem2	Sem3	Sem4
Realizar el código necesario para las colisiones	Acordar el número de pelotas que se deben disparar..	Realizar código para el número de pelotas y tiempo; primeras pruebas	Ajuste de vista en tercera y primera persona	Realizar seguimiento de pelotas al personaje principal.	Unión de partículas con pelotas al colisionar

Tabla 2. Cronograma diciembre a enero

Enero	Febrero	
sem5	sem1	Sem2
Cambios y correcciones hasta el momento..	Realización de documentación y multimedia necesaria	Presentación Corrección de errores

Tabla 3. Cronograma enero y febrero

Dinámicas del juego

Las dinámicas del juego inicialmente eran las siguientes:

- ★ El jugador debe disparar a las pelotas saltarinas
- ★ Tiene controles de disparo, movimiento, rotación y salto.
- ★ Se gana en caso de matar a todas las pelotas saltarinas en cierto tiempo
- ★ Se pierde en caso de perder tres vidas por las pelotas saltarinas o no conseguirlo en el tiempo planteado

Para el final del proyecto cambiaron algunas pero de eso se hablará más adelante.

6. DESARROLLO

Modelos y animaciones

Comenzamos buscando todos los modelos posibles para utilizarlos así como también modelos realizados desde cero para poder estar seguros que se renderizaron alrededor de nuestro proyecto:

Como personaje principal utilizamos un modelo de May en el juego de Pokémon Omega Ruby / Alpha Sapphire, y agregando animaciones a través de Mixamo, también se le agregó al modelo una pistola para cuando va a disparar(Véase Figura 4):

Figura 4. Modelo May con animaciones

Para las pelotas se utilizó como modelo una simple esfera que con una textura roja, haciendo sencillo su modelado, ya que este al contrario de May, fue realizado desde cero:

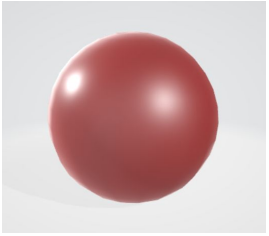

Figura 5. Modelo RedBall

Como se mencionó anteriormente hubo modelos que se realizaron desde cero, tratando de que tuvieran la menor cantidad de vértices posibles y haciendo que las las texturas fueran lo más chicas posibles haciendo que el proyecto no ocupara tanto volumen y el renderizado fuera mucho más rápido y fácil. Algunos de los modelos realizados desde cero son los que se pueden observar en la figura 7.

Figura 6. Modelos realizados desde 0

Skybox y Terreno

Para el terreno realizamos diferentes texturas con la ayuda de la herramienta Gimp, teniendo como base el uso de colores pasteles y dando un toque de nube, ya que la idea del terreno era asemejar el algodón de azúcar, o chicle(Véase la figura 7).

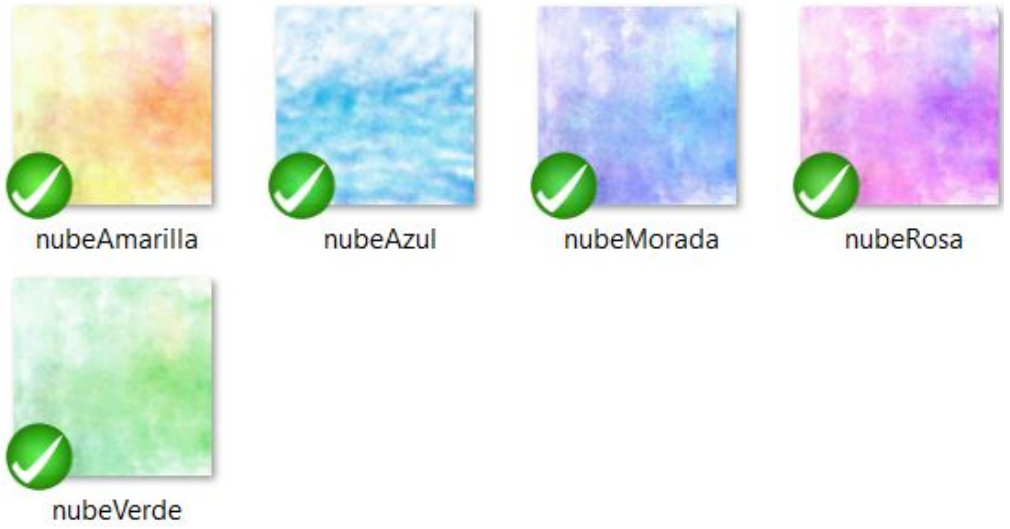

Figura 7. Texturas utilizadas.

realizamos el mapa de alturas correspondiente a nuestra idea de “piso de algodón/chicle” así como también el mapa de texturas tomando en cuenta las diferentes alturas consideradas en el heightmap(véase figura 8).

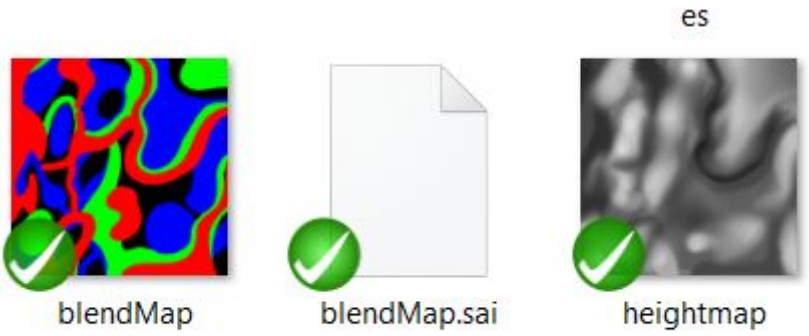

Figura 8. Mapas de texturas y de alturas.

En cuanto al skybox realizamos igualmente con ayuda de de la herramienta de Gimp una serie de imágenes con la idea de proyectar un ambiente dulce (dulce en el cielo), así como en el terreno; Al igual que todo el el proyecto, los colores de la paleta del skybox son en su mayoría colores pasteles.

También se creó un skybox con temática de noche, de tal manera que las luces creadas para este proyecto pudieran visualizarse con mucha menor cantidad de luz en el terreno(Véase Figura 9).

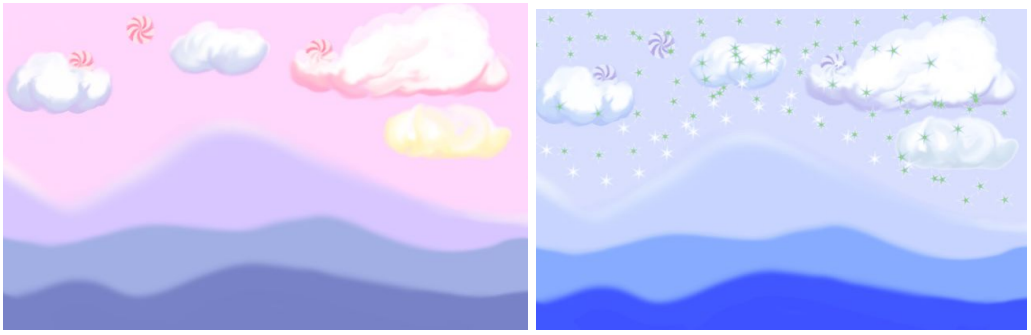

Figura 9. Skybox de día y de noche respectivamente

Análisis del código realizado

Hicimos uso de los diferentes shader vistos a lo largo del semestre, como fueron el shader multilighting, shader de partículas, shader Skybox, de terreno, entre otros(véase figura 10):

```
shader.initialize("../Shaders/colorShader.vs", "../Shaders/colorShader.fs");
shaderSkybox.initialize("../Shaders/skyBox.vs", "../Shaders/skyBox_fog.fs");
shaderMulLighting.initialize("../Shaders/iluminacion_textura_animation_shadow.vs", "../Shaders/multipleLights_shadow.fs");
shaderTerrain.initialize("../Shaders/terrain_shadow.vs", "../Shaders/terrain_shadow.fs");
shaderParticlesFountain.initialize("../Shaders/particlesFountain.vs", "../Shaders/particlesFountain.fs");
shaderParticlesFire.initialize("../Shaders/particlesFire.vs", "../Shaders/particlesFire.fs", { "Position", "Velocity", "Age" });
shaderViewDepth.initialize("../Shaders/texturizado.vs", "../Shaders/texturizado_depth_view.fs");
shaderDepth.initialize("../Shaders/shadow_mapping_depth.vs", "../Shaders/shadow_mapping_depth.fs");
```

Figura 9. Shaders ocupados

Se realizaron las declaraciones y renderizaciones en el mapa de los modelos utilizando el método para colocar luces(véase figura 11), como se vió en esa respectiva práctica, utilizando el mapa de heigmap para dar la orientación de dichos modelos(Véase figura 12):

```
// Render the lamps
for (int i = 0; i < lamp1Position.size(); i++) {
    lamp1Position[i].y = terrain.getHeightTerrain(lamp1Position[i].x, lamp1Position[i].z);
    BastonLampara.setPosition(lamp1Position[i]);
    BastonLampara.setScale(glm::vec3(9.0, 9.0, 9.0));
    BastonLampara.setOrientation(glm::vec3(0, lamp1Orientation[i], 0));
    BastonLampara.render();
}

// Render the CuteHouse1
for (int i = 0; i < CutH1Position.size(); i++) {
    CutH1Position[i].y = terrain.getHeightTerrain(CutH1Position[i].x, CutH1Position[i].z);
    CuteHome1.setPosition(CutH1Position[i]);
    CuteHome1.setScale(glm::vec3(4.0, 4.0, 4.0));
    CuteHome1.setOrientation(glm::vec3(0, CutH1Orientation[i], 0));
    CuteHome1.render();
}

// Render the CuteHouse2
for (int i = 0; i < CutH2Position.size(); i++) {
    CutH2Position[i].y = terrain.getHeightTerrain(CutH2Position[i].x, CutH2Position[i].z);
    CuteHome2.setPosition(CutH2Position[i]);
    CuteHome2.setScale(glm::vec3(2.0, 2.0, 2.0));
    CuteHome2.setOrientation(glm::vec3(0, CutH2Orientation[i], 0));
    CuteHome2.render();
}
```

Figura 11. Parte del código realizado para el render de modelos

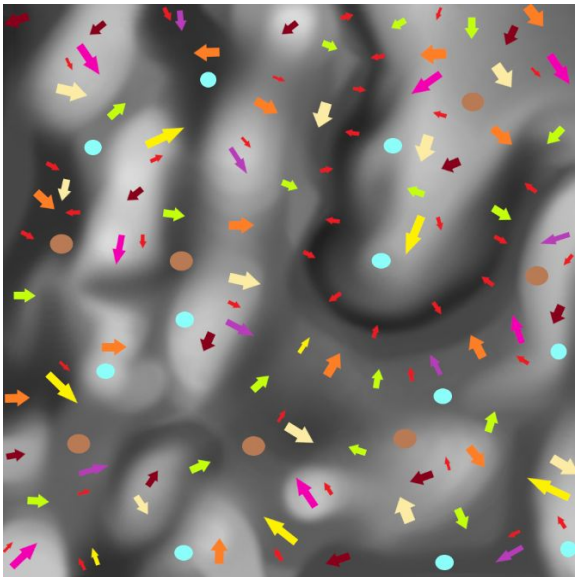

Figura 12. Imagen utilizada para la ubicación de los diferentes modelos empleados

Para la parte del código realizamos las colisiones de Caja vs Esfera y Caja vs Caja(OBB y SBB); De tal manera que se nos facilitara el realizar la colisión que queríamos observar, la cuál era May Pelotas., todos los modelos cuentan con su respectiva colisión, como se puede visualizar en la siguiente figura:

```
//Collider de las chocoPaleta
for (int i = 0; i < ChocoPPosition.size(); i++) {
    AbstractModel::OBB ChocoPCollider;
    glm::mat4 modelMatrixColliderChocoP = glm::mat4(1.0);
    modelMatrixColliderChocoP = glm::translate(modelMatrixColliderChocoP, ChocoPPosition[i]);
    modelMatrixColliderChocoP = glm::rotate(modelMatrixColliderChocoP, glm::radians(ChocoPOrientation[i]),
        glm::vec3(0, 1, 0));
    addOrUpdateColliders(collidersOBB, "ChocoPaleta-" + std::to_string(i), ChocoPCollider, modelMatrixColliderChocoP);
    // Set the orientation of collider before doing the scale
    ChocoPCollider.u = glm::quat_cast(modelMatrixColliderChocoP);
    modelMatrixColliderChocoP = glm::scale(modelMatrixColliderChocoP, glm::vec3(2.0, 1.5, 1.5));
    modelMatrixColliderChocoP = glm::translate(modelMatrixColliderChocoP, ChocoPaleta.getObb().c);
    ChocoPCollider.c = glm::vec3(modelMatrixColliderChocoP[3]);
    ChocoPCollider.e = ChocoPaleta.getObb().e * glm::vec3(2.0, 1.5, 1.5);
    std::get<0>(collidersOBB.find("ChocoPaleta-" + std::to_string(i))->second) = ChocoPCollider;
}
//Collider de las Cookies
for (int i = 0; i < CookPosition.size(); i++) {
    AbstractModel::OBB CookCollider;
    glm::mat4 modelMatrixColliderCook = glm::mat4(1.0);
    modelMatrixColliderCook = glm::translate(modelMatrixColliderCook, CookPosition[i]);
    modelMatrixColliderCook = glm::rotate(modelMatrixColliderCook, glm::radians(CookOrientation[i]),
        glm::vec3(0, 1, 0));
    addOrUpdateColliders(collidersOBB, "Cookie-" + std::to_string(i), CookCollider, modelMatrixColliderCook);
    // Set the orientation of collider before doing the scale
    CookCollider.u = glm::quat_cast(modelMatrixColliderCook);
    modelMatrixColliderCook = glm::scale(modelMatrixColliderCook, glm::vec3(150.0, 100.5, 150.5));
    modelMatrixColliderCook = glm::translate(modelMatrixColliderCook, Cookie.getObb().c);
    CookCollider.c = glm::vec3(modelMatrixColliderCook[3]);
    CookCollider.e = Cookie.getObb().e * glm::vec3(140.0, 100.5, 140.5);
    std::get<0>(collidersOBB.find("Cookie-" + std::to_string(i))->second) = CookCollider;
}

//Collider de las Donas
```

Figura 13. Parte de código de los colliders de los diferentes modelos

En cuanto al movimiento de la MayCute y sus animaciones, se utilizó el mando de un control de consola con la la función de GLFW_JOYSTICK, de tal forma que se pueda mover ya no solamente con el teclado de la computadora, si no con cualquier mando aceptable para la librería como se muestra en la figura 14.

```
if (present == 1) {
    int count;
    const float* axes = glfwGetJoystickAxes(GLFW_JOYSTICK_1, &count);
    if (axes[0] < -0.5) {
        matrixModelMayow = glm::rotate(matrixModelMayow, glm::radians(1.0f), glm::vec3(0, 1, 0));
        animationIndex = 1;
    }
    else if (axes[0] > 0.5) {
        matrixModelMayow = glm::rotate(matrixModelMayow, glm::radians(-1.0f), glm::vec3(0, 1, 0));
        animationIndex = 1;
    }
    else if (axes[1] > 0.5) {
        matrixModelMayow = glm::translate(matrixModelMayow, glm::vec3(0, 0, 0.3));
        animationIndex = 1;
    }
    else if(axes[1] < -0.5) {
        matrixModelMayow = glm::translate(matrixModelMayow, glm::vec3(0, 0, -0.3));
        animationIndex = 1;
    }
    else if (axes[2] < -0.5) {
        camera->mouseMoveCamera(offsetX, 0.0, deltaTime);
    }
    else if (axes[2] > 0.5) {
        camera->mouseMoveCamera(offsetX, offsetY, deltaTime);
    }
    else if (axes[4] > 0.5) {
        animationIndex = 0;
    }
    else if (axes[5] > 0.5) {
        animationIndex = 2;
    }
    else {
        animationIndex = 7;
    }
    const char *Control_name = glfwGetJoystickName(GLFW_JOYSTICK_1);
```

Figura 14. Declaraciones para el uso de un mando de consola

En cuanto a la neblina que utilizamos, realizamos los cambios necesarios para tener una niebla de color rosa, utilizando el color de pantone(Véase la figura 15). Tomando en cuenta que debe dividirse todo ente 255(véase la figura 16).

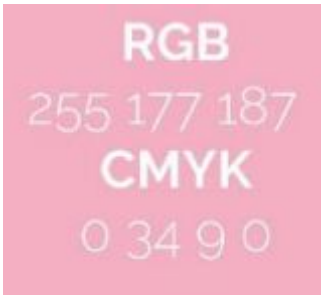

Figura 15. Color pantone utilizado para la neblina

```
/*****************************************
 * Agregar: Propiedades de la neblina
 *****************************************/
shaderMulLighting.setVectorFloat3("fogColor", glm::value_ptr(glm::vec3(1.0, 0.69412, 0.73334)));
shaderTerrain.setVectorFloat3("fogColor", glm::value_ptr(glm::vec3(1.0, 0.69412, 0.73334)));
shaderSkybox.setVectorFloat3("fogColor", glm::value_ptr(glm::vec3(1.0, 0.69412, 0.73334)));
```

Figura 16. Color de la neblina en el código

Para las luces que se fueron implementando a lo largo del proyecto, utilizamos únicamente luz de tipo pointLights, realizando esto en el modelo del bastón de dulce, la declaración de las luces se muestra en la siguiente figura 17

```
/*****************************************
 * Propiedades PointLights
 *****************************************/

shaderMulLighting.setInt("pointLightCount", lamp1Position.size());
shaderTerrain.setInt("pointLightCount", lamp1Position.size());
for (int i = 0; i < lamp1Position.size(); i++) {
    glm::mat4 matrixAdjustLamp = glm::mat4(1.0f);
    matrixAdjustLamp = glm::translate(matrixAdjustLamp, lamp1Position[i]);
    matrixAdjustLamp = glm::rotate(matrixAdjustLamp, glm::radians(lamp1Orientation[i]), glm::vec3(0, 1, 0));
    matrixAdjustLamp = glm::scale(matrixAdjustLamp, glm::vec3(1.0, 1.0, 1.0));
    matrixAdjustLamp = glm::translate(matrixAdjustLamp, glm::vec3(0, 15.3585, 0));
    glm::vec3 lampPosition = glm::vec3(matrixAdjustLamp[3]);
    shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.ambient", glm::value_ptr(glm::vec3(0.2, 0.16, 0.01
    shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.diffuse", glm::value_ptr(glm::vec3(0.4, 0.32, 0.02
    shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.specular", glm::value_ptr(glm::vec3(0.6, 0.58, 0.0
    shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].position", glm::value_ptr(lampPosition));
    shaderMulLighting.setFloat("pointLights[" + std::to_string(i) + "].constant", 1.0);
    shaderMulLighting.setFloat("pointLights[" + std::to_string(i) + "].linear", 0.09);
    shaderMulLighting.setFloat("pointLights[" + std::to_string(i) + "].quadratic", 0.01);
    shaderTerrain.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.ambient", glm::value_ptr(glm::vec3(0.2, 0.16, 0.01)));
    shaderTerrain.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.diffuse", glm::value_ptr(glm::vec3(0.4, 0.32, 0.02)));
    shaderTerrain.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.specular", glm::value_ptr(glm::vec3(0.6, 0.58, 0.03)))
    shaderTerrain.setVectorFloat3("pointLights[" + std::to_string(i) + "].position", glm::value_ptr(lampPosition));
    shaderTerrain.setFloat("pointLights[" + std::to_string(i) + "].constant", 1.0);
    shaderTerrain.setFloat("pointLights[" + std::to_string(i) + "].linear", 0.09);
    shaderTerrain.setFloat("pointLights[" + std::to_string(i) + "].quadratic", 0.01);
}
```

Figura 17. Declaración de las luces

Para la colocación de las pelotas de forma aleatoria hicimos uso de la función rand, dándonos así lugares random donde las pelotas pueden hacer su aparición(Véase figura 18)

```
if (ronda == 1) {
    cantidad_de_pelotas = (rand() % (ronda * 5 + 1))+5;
    printf("cantidad de pelotas :%d \n", cantidad_de_pelotas);
    for (int j = 0; j < cantidad_de_pelotas; j++) {
        pos_chica = rand() % (101);
        pos_grande = rand() % (101);
        pos_grande = -700 + pos_grande;
        decicion = rand() % (4);
        if (decicion == 0) {
            posx = pos_chica;
            posz = pos_grande;
        }
        if (decicion == 1) {
            posx = pos_grande;
            posz = pos_chica;
        }
        if (decicion == 2) {
            posx = pos_grande;
            posz = pos_grande;
        }
        if (decicion == 3) {
            posx = pos_chica;
            posz = pos_chica;
        }
        pelotasPosition.push_back(glm::vec3(posx, 0, posz));
    }
    ronda += 1;
}
```

Figura 18.Código de lugar random de las pelotas

También dependiendo de si la pelota colisiona con May o no, se renderizan las partículas, las cuales representan una explosión de dichas pelotas. como se muestra en la figura 19.

```
if (ColisionMP == true) {
BallKirby.render();
}
else {//Aqui va la renderización en este mismo punto de la particula
    glm::mat4 modelMatrixParticlesFountain = glm::mat4(1.0);
    modelMatrixParticlesFountain = glm::translate(modelMatrixParticlesFountain, pelotasPosition[i]);
    modelMatrixParticlesFountain[3][1] = terrain.getHeightTerrain(modelMatrixParticlesFountain[3][0], m
    modelMatrixParticlesFountain = glm::scale(modelMatrixParticlesFountain, glm::vec3(3.0, 3.0, 3.0));
    currTimeParticlesAnimation = TimeManager::Instance().GetTime();
}
```

Figura 19.Función que renderiza a la partícula o a la pelota.

Para la parte de la música, realizamos dos pequeñas pistas de audio: una para la música de fondo general del juego y otra para las pelotas, cuando saltan(Véase figura 20).

```
// Config source 0
// Generate buffers, or else no sound will happen!
alGenBuffers(NUM_BUFFERS, buffer);
buffer[0] = alutCreateBufferFromFile("../sounds/cuteMusicFondo.wav");
buffer[1] = alutCreateBufferFromFile("../sounds/Saltito.wav");
```

Figura 20.Declaración de las pistas de audio a utilizar.

En el caso de la música de fondo, se optó por que proviniera del modelo de MayCute, ya que haciendo esto, la música siempre se escucharía igual, esto es debido a que la cámara apunta directamente a dicho personaje. Mientras que la música de los saltos debe provenir de los diferentes modelos de las pelotas, como se muestra en l a figura 21.

```
//SonidoMusicaFondo
source0Pos[0] = matrixModelMayow[3].x;
source0Pos[1] = matrixModelMayow[3].y;
source0Pos[2] = matrixModelMayow[3].z;
alSourcefv(source[0], AL_POSITION, source0Pos);
//SonidoPelota
source1Pos[0] = matrixModelBallKirby[3].x;
source1Pos[1] = matrixModelBallKirby[3].y;
source1Pos[2] = matrixModelBallKirby[3].z;
alSourcefv(source[1], AL_POSITION, source1Pos);

// Listener for the Thris person camera
listenerPos[0] = matrixModelMayow[3].x;
listenerPos[1] = matrixModelMayow[3].y;
listenerPos[2] = matrixModelMayow[3].z;
alListenerfv(AL_POSITION, listenerPos);
```

Figura 21.Origen de las diferentes pistas.

7. Costo desglosado de su proyecto

El costo del proyecto final se puede dividir por las siguientes cosas como indica la tabla 4, la mayoría de los datos fueron ingresados pensando en la realidad y obteniendo precios de diferentes sitios web. Tomando en cuenta que nuestro proyecto es independiente y no cuenta con el mayor apoyo requerido para una gran franquicia.

Costos desglosados del proyecto	
Descripción	**Costo**
Personajes jugables	$2,500.00
Licencia de Visual Studio	$1,119.00
Mando personalizado de consola	$1,200.00
Soporte técnico y actualizaciones de errores	$14,700.00
Modelos y desarrollo de terreno	$1,700.00
Desarrollo general de juego	$12,000.00
Desarrollado por dos personas(al rededor de 6 meses, contando todo el semestre)	$15,000.0
Análisis de rendimiento	$7,000.0
Análisis del uso de la GPU	$5,000.0
Total	$60,219

Por lo tanto, creemos que al menos debemos contar con $60,000.00 pesos mexicanos para realizar este proyecto en la vida real.

8. LICENCIAMIENTO

Para poder sacar nuestro juego a una venta real, necesitamos diferentes licencias como la de Visual Studio, así como también realizar el registro del video juego: Trámite que deberá realizar ante la Subsecretaría de Programas Delegacionales y Reordenamiento de la Vía Pública dependiente de la Secretaría de Gobierno las personas físicas o morales para la obtención del Registro de Videojuegos que pretendan poner en operación los Establecimientos Mercantiles.
Desde esta perspectiva, otro aspecto a tomar en consideración es que la adquisición de licencias de uso depende no solo de la voluntad del Licenciatario de adquirir dicha licencia, sino que también depende de que el Licenciante quiera otorgar. Es allí en donde entra en juego el contexto situacional en donde serán presentadas las marcas o los productos como tal dentro del videojuego.
En nuestro caso al utilizar openGl que es software libre no debemos pagar una licencia a menos que queramos utilizar otras librerías con un costo real.
Otra cosa a tomar en cuenta es el uso de modelos con derechos de autor, ya sea que debamos cambiarlos(como el caso de nuestro personaje principal) o realizar otros modelos libres de derechos de autor, que sean hechos por nosotros con registro.

9. PRINCIPALES RETOS Y DIFICULTADES

La comunicación entre los integrantes del equipo cuyas reuniones fueron a distancia (no presencial), debido a ello se integraron ideas distintas del concepto principal de cada uno de los integrantes sin embargo llegando a consensos logramos integrar las ideas con éxito.

El equipo necesario para el proyecto(las computadoras), se debe tener disponibilidad de horario en una computadora personal o portátil por el tiempo de realización del proyecto, tener visual studio instalado y por lo tanto memoria en disco duro, además de la memoria de la tarjeta gráfica que en nuestro caso fallo en el equipo de uno de los integrantes, causa desconocida, pero se cree que fue la capacidad de la tarjeta gráfica o de la ram del equipo de computo.

Retroalimentación y tips, al tener la comunicación limitada y no tener forma de obtener nuevas ideas sobre la implementación de ciertos módulos al momento de

estancarse o bloquearse en un punto de la implementación era necesario replantearse el problema y seguir a prueba y error hasta hallar la solución al problema por lo que requerimos más tiempo del necesario al implementar ciertos módulos.

OpenGL no es amigable con el desarrollador, aunque éste ha mejorado mucho a través de los años sigue siendo una herramienta con una curva de aprendizaje alta.

El uso de colisiones de dos objetos en movimiento fue difícil ya que los colisionadores deben ser relativos al objeto designado.

10. MEJORAS, TRABAJO A FUTURO

Para las mejoras de trabajo, sería la implementación de otros modelos para personajes principales, como se tenía previsto desde el inicio, así como pantallas de carga o pequeños videos de multimedia agregando historia al juego y haciéndolo más atractivo.
Otra mejora que se debe tener en cuenta a corto plazo es solucionar los errores que no se pudieron llevar a cabo por la falta de tiempo o la falta de experiencia; como lo fué el seguimiento de las pelotas a lo largo del mapa hacia el modelo MayCute. De esta forma podríamos haber implementado el uso de collider para realizar la explosión de las pelotas y convertirlas en partículas.

El tiempo openGl requiere tiempo ya que mucha de las implementaciones son "talacha", es decir, tareas repetitivas como el renderizado de los modelos tomando en cuenta el proceso que debe tener este hasta el renderizado.

11. CONCLUSIONES

Para realizar un proyecto, siempre es necesario tener claros los objetivos y contar con un planteamiento firme para no regresar a esta parte a la hora de comenzar a diseñar.
Para nuestro proyecto final fue muy importante haber resuelto las dudas que se pudieron tener a lo largo de las diferentes prácticas realizadas en las diferentes semanas. De tal manera que podamos consultar las prácticas como apoyo o guías en el proyecto.
Debemos tener en cuenta realizar búsquedas en internet para poder utilizar

diferentes elementos, como los fue el uso de un mando de consola, pero siempre con conocimientos previos para poder entender los diferentes artículos buscados.

REPOSITORIO DE GITHUB

HTTPS://GITHUB.COM/MARIMAPLE/COMPUTACIONGRAFICAAVANZADAPROYECTO

BIBLIOGRAFÍA

https://www.models-resource.com/3d
https://www.xataka.com.mx/videojuegos/precio-base-algunos-videojuegos-mexico-se-dispara-1-699-pesos-eso-no-buenas-noticias
https://neox.atresmedia.com/games/noticias/videos/cuanto-dinero-cuesta-realmente-desarrollar-un-videojuego_201803125aa636600cf23f53ffd27533.html
https://videojuegos.mercadolibre.com.mx/para-playstation-ps4-4-gamepads-joysticks/control-ps4
https://visualstudio.microsoft.com/es/vs/support/purchasing-visual-studio-professional/
https://visualstudio.microsoft.com/es/vs/pricing/
https://visualstudio.microsoft.com/es/vs/features/game-development/
https://pdelegacionales.cdmx.gob.mx/servicios/servicio/registro-videojuejos