Universidad Nacional Autónoma de México

Facultad de Ingeniería

División de Ingeniería Eléctrica

Computación Gráfica Avanzada

Asesor: Martell Ávila Reynaldo

Alumnos:

Cuevas Salgado Carlos

Galindo Chavez Lisset América

Reporte Proyecto Final:

Semestre 2020-2

Introducción

Aplicando los conocimientos obtenidos en la materia de computación gráfica avanzada se desarrolló “Run or Die” un videojuego implementado en C++ usando la API OpenGL para la manipulación de gráficos y OpenAL para manejar audio espacial, es decir, que respete el efecto doppler. Adicionalmente se utilizó la librería Freetype para construir texto. Por otro lado, el software Blender fungió como principal herramienta para la creación de modelos así como para animarlos y editar modelos ya existentes. Además, el sitio web mixamo brindó la oportunidad de descargar animaciones complejas y reducir los tiempos de edición.

Las animaciones del videojuego son de tres tipos: mediante máquinas de estado, keyframes y animaciones mediante esqueleto. Las primeras son las más sencillas de emplear, sin embargo, no ofrecen animaciones fluidas. Mediante keyframes la animación es más fluida, se usó este tipo de animación para grabar los recorridos. Las animaciones por esqueleto son las más naturales, estas se pueden crear en blender y posteriormente exportar el modelo como fbx, o bien descargarlas de la página mixamo.

Se usaron tres tipos de terrenos: alturas, mezclas de mapas, y lógico. En el primer tipo de terreno solo se utiliza la escala de grises, el negro indica que no hay altura, mientras el blanco representa la altura máxima, a partir de la tonalidad del gris será la altura. El terreno de mapeo de mezclas ayuda a combinar hasta cuatro diferentes texturas en un solo mapa de forma que el terreno se vea más natural. Finalmente el terreno lógico está construido solo con escala de grises e indica a partir de la tonalidad o zona las acciones que se pueden realizar, utilizado especialmente para delimitar la pista.

En el videojuego se usan los tres tipos de luces: direccional, puntual y de linterna. La primera es como un sol, a partir de los valores dados permite representar la hora del día, la tarde y noche, la luz de tipo puntual es como un foco esta se implementa en las lámparas del videojuego, finalmente la luz de linterna se usa en los faros del carro. Otro auxiliar para representar el estado del día es la neblina. También se implementó lo que es el shadow mapping el cual permite generar sombras aumentando el realismo del videojuego.

La cámara usada en este videojuego es de tercer persona, es decir, se siguen las acciones que realiza el personaje principal. Se emplea lo que es el blending el cual ayuda a la creación de las transparencias. Estas se pueden notar en las animaciones con partículas. Se tiene dos sistemas de partículas, el primero es sin retroalimentación por lo que empieza la animación, esta termina y vuelve a

empezar, el segundo es con retroalimentación por lo que se generan partículas antes de que termine la animación dando la ilusión de que estas no termina.

Repositorios:

- https://github.com/cuevas097/DocumentacionRunOrDie
- https://github.com/America1096/RunOrDieCodigo

Objetivo del juego

El juego consiste en manejar un lamborghini aventador, dar tres vueltas a la pista antes de que se agote el tiempo, para reunir más tiempo se debe llegar a los arcos rojos (checkpoints). Debe esquivar los obstáculos, no debe salirse de la pista ya que de lo contrario se le restaron puntos de vida. Se tienen monedas y diamantes las cuales proporcionan nitro, una vez que tenga la barra de nitro llena puede activarse para aumentar su velocidad.

Desarrollo

Animaciones: Se manejaron tres formas de animación mediante máquinas de estado, keyframes y mediante esqueleto.

En el primer tipo de animación que se utilizó para la animación del topo y para los créditos. La animación del topo utiliza partículas para simular la tierra por lo que en el código de las partículas también se ve influenciada por la maquina de estados.

Máquina del topo:

```
else if (renderParticles && it->second.first.compare("topito") == 0 && ActivaTopito == 0) {
glm::mat4 modelMatrixParticlesTopito = glm::mat4(1.0);
modelMatrixParticlesTopito = glm::translate(modelMatrixParticlesTopito, it->second.second);
modelMatrixParticlesTopito[3][1] = terrain.getHeightTerrain(modelMatrixParticlesTopito[3][0], modelMatrixParticlesTopito[3][2]) - 0.5;
modelMatrixParticlesTopito = glm::scale(modelMatrixParticlesTopito, glm::vec3(10.0, 10.0, 10.0));
currTimeParticlesAnimationTopito = TimeManager::Instance().GetTime();
if (currTimeParticlesAnimationTopito - lastTimeParticlesAnimationTopito > 10.0) {
    lastTimeParticlesAnimationTopito = currTimeParticlesAnimationTopito;
    ActivaTopito = 1;
}
```

```
/*****************************
    Topito
*****************************/
if (ActivaTopito == 1) {
    if (subetopito == true) {
        MovimientoenYTopito += 0.05;
        if (MovimientoenYTopito >= 3.5f) {
            subetopito = false;
        }
    }
    else if (subetopito == false) {
        MovimientoenYTopito -= 0.05;
        if (MovimientoenYTopito <= 0.0f) {
            subetopito = true;
            ActivaTopito = 2;
        }
    }
}
else if (ActivaTopito == 2) {
    ActivaTopito = 0;
}
```

La animación del topo inicia con las partículas por lo que tiene el estado 0, cuando han pasado 10 segundos pasa al estado 1, en este estado el topo sube y baja, terminando esto pasa al estado 2 para regresar al estado 0. Se hizo de esta manera porque al dejarlo solo con 2 estados la animación tendia

a fallar ya que no entraba se mostraban las partículas.

Máquina de los créditos:

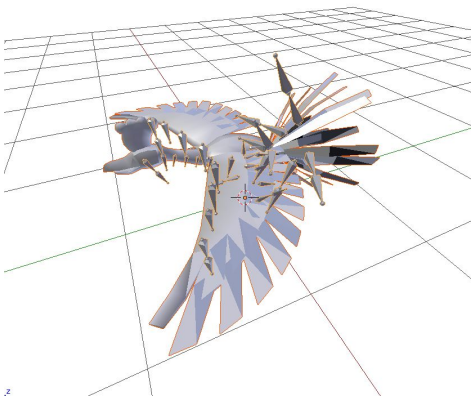
```
// Maquinas de estado para los creditos
if (creditos)
{
    switch (stateAku) {
    case 0:
        moverAku += 0.01;
        if (moverAku > 1)
            stateAku = 1;
        break;
    case 1:
        moverAku -= 0.01;
        if (moverAku < -1) {
            //              moverAku = 0.0;
            stateAku = 0;
        }
        break;
    }
}
```

Esta máquina es más sencilla, cuando inician los créditos el modelo se empieza a mover de derecha a izquierda.

Los keyframes se emplearon para manejar los recorridos, estos se grabaron en un archivo txt que posteriormente se lee. En el recorrido del ave se emplearon keyframes para su recorrido por la pista y animaciones por esqueleto para que el ave tuviera movimiento realista.

Codigo de animación por keyframes:

```
if (keyFramesHooh.size() > 0 && ActivaRecorrido) {
    // Para reproducir el frame
    interpolationHooh = numPasosHooh / (float)maxNumPasosHooh;
    numPasosHooh++;
    if (interpolationHooh > 1.0) {
        numPasosHooh = 0;
        interpolationHooh = 0;
        indexFrameHooh = indexFrameHoohNext;
        indexFrameHoohNext++;
    }
    if (indexFrameHoohNext > keyFramesHooh.size() - 1)
        indexFrameHoohNext = 0;
    modelMatrixHooh = interpolate(keyFramesHooh, indexFrameHooh, indexFrameHoohNext, 0, interpolationHooh);
}
```



La animación por esqueleto se empleó para animar 4 modelos en total, de las cuales 3 animaciones fueron descargadas de la página mixamo, solo se hizo la animación del ave.

Se muestra el modelo en blender con los huesos para su animación.

A continuación se muestra el código con el que se renderiza:

```
//Hooh
modelMatrixHooh[3][1] = terrain.getHeightTerrain(modelMatrixHooh[3][0], modelMatrixHooh[3][2]) + 10.0f;
glm::mat4 modelMatrixHoohBody = glm::mat4(modelMatrixHooh);
modelMatrixHoohBody = glm::scale(modelMatrixHoohBody, glm::vec3(0.021, 0.021, 0.021));
HoohModel.setAnimationIndex(0);
HoohModel.render(modelMatrixHoohBody);
```

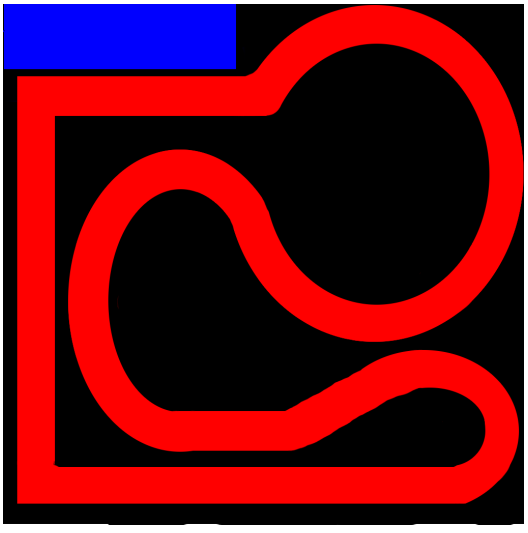
En el caso de las animaciones que se descargan de la página mixamo para generar su caja de colisión esta debe modificarse, primero se debe escalar, para se debe multiplicar por el escalamiento que se tiene en blender, en este caso es 100, por el tamaño al que se desea escalar.

```
//Collider de Harley
AbstractModel::OBB harleyCollider;
if (contadorArco < 1 && vueltaCont <= 1 && harley) {
    glm::mat4 modelmatrixColliderHarley = glm::mat4(modelMatrixHarley);
    // Set the orientation of collider before doing the scale
    harleyCollider.u = glm::quat_cast(modelMatrixHarley);
    modelmatrixColliderHarley = glm::scale(modelmatrixColliderHarley, glm::vec3(100.0f, 100.0f, 100.0f) *
        glm::vec3(0.015, 0.015, 0.015));
    modelmatrixColliderHarley = glm::translate(modelmatrixColliderHarley, harleyModel.getObb().c);
    harleyCollider.e = harleyModel.getObb().e * glm::vec3(100.0f, 100.0f, 100.0f) *
        glm::vec3(0.015, 0.015, 0.015);
    harleyCollider.c = glm::vec3(modelmatrixColliderHarley[3]);
    addOrUpdateColliders(collidersOBB, "harley", harleyCollider, modelMatrixHarley);
}
else
{
    harleyCollider.u = glm::quat_cast(glm::mat4(1.0));
    harleyCollider.c = glm::vec3(-50.0f, -50.0f, -50);
    harleyCollider.e = glm::vec3(0.0f, 0.0f, 0.0f);
    addOrUpdateColliders(collidersOBB, "harley", harleyCollider, modelMatrixHarley);
}
```

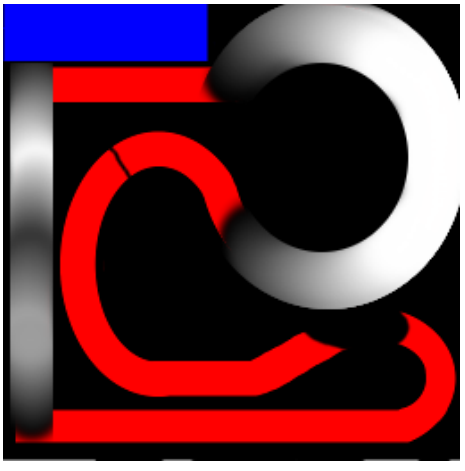
Terrenos

Se manejaron tres diferentes terrenos, de alturas, de mezclas de mapas y lógico. Primero se desarrolló el mapa del terreno para construir la pista, una vez con ese mapa se hizo el mapa de alturas, el último mapa que se construyó fue el lógico, este último para ayuda a que el carro no se salga de la pista, puede tocar ciertas zonas sin embargo esto le resta puntos de vida.

Mapa de mezclas:



Mapas de alturas: Las partes en color rojo o azul, no se consideran, se dejan para referencia de la pista .



Mapa lógico:

Código del mapa lógico:

Se declara el terreno y la textura con sus respectivas dimensiones.

```
// Terrain model instance
Terrain terrain(-1, -1, 200, 12, "../Textures/testHeightProyecto2.png");
// Terreno para limitar pista
Terrain terrainLogic(-1, -1, 200, 12, "../Textures/logicTextureMap2.png");
```

Se inicializa y se le da una posición.

```
terrainLogic.init();
terrainLogic.setPosition(glm::vec3(100, 0, 100));
```

Se solicita la altura del carro en el mapa lógico.

```
modelMatrixLambo[3][1] = terrain.getHeightTerrain(modelMatrixLambo[3][0], modelMatrixLambo[3][2]);
float region = terrainLogic.getHeightTerrain(modelMatrixLambo[3][0], modelMatrixLambo[3][2]);
float terrenoInicio = terrainLogicInicio.getHeightTerrain(modelMatrixLambo[3][0], modelMatrixLambo[3][2]);
```

Una vez que se tiene la región se aplican las condiciones para que el carro no salga de la pista.

Luces

Se manejan tres tipos de luces: direccional, puntual y de linterna. Del primer tipo de luz solo se tiene una, dependiendo del número de vueltas esta se cambia para poder manejar la ambientación de la pista en tres estados de día, atardecer y noche. La luz puntual se maneja para las lámparas y el semáforo. La luz de linterna se utiliza en los faros del carro.

Código del semáforo:

```
if (Verde == false) {
    for (int i = 0; i < contRoja; i++) {
        glm::mat4 matrixAdjustLamp = glm::mat4(1.0f);
        matrixAdjustLamp = glm::translate(matrixAdjustLamp, SemaforoRojo[i]);
        matrixAdjustLamp = glm::scale(matrixAdjustLamp, glm::vec3(1.1, 1.1, 1.1));
        glm::vec3 lampPosition = glm::vec3(matrixAdjustLamp[3]);
        shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.ambient", glm::value_ptr(glm::vec3(0.5, 0.01, 0.01)));
        shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.diffuse", glm::value_ptr(glm::vec3(0.7, 0.01, 0.01)));
        shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.specular", glm::value_ptr(glm::vec3(0.9, 0.01, 0.01)));
        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(0.5, 0.01, 0.01)));
        shaderTerrain.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.diffuse", glm::value_ptr(glm::vec3(0.7, 0.01, 0.01)));
        shaderTerrain.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.specular", glm::value_ptr(glm::vec3(0.6, 0.01, 0.01)));
        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", 0.02);
    }
}
if (Verde == true) {
    for (int i = 0; i < SemaforoVerde.size(); i++) {
        glm::mat4 matrixAdjustLamp = glm::mat4(1.0f);
        matrixAdjustLamp = glm::translate(matrixAdjustLamp, SemaforoVerde[i]);
        matrixAdjustLamp = glm::scale(matrixAdjustLamp, glm::vec3(1.1, 1.1, 1.1));
        glm::vec3 lampPosition = glm::vec3(matrixAdjustLamp[3]);
        shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.ambient", glm::value_ptr(glm::vec3(0.0, 1.0, 0.0)));
        shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.diffuse", glm::value_ptr(glm::vec3(0.0, 1.0, 0.0)));
        shaderMulLighting.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.specular", glm::value_ptr(glm::vec3(0.0, 1.0, 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(1.0, 1.0, 1.0)));
        shaderTerrain.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.diffuse", glm::value_ptr(glm::vec3(1.0, 1.0, 1.0)));
        shaderTerrain.setVectorFloat3("pointLights[" + std::to_string(i) + "].light.specular", glm::value_ptr(glm::vec3(1.0, 1.0, 1.0)));
        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", 0.02);
    }
}
```

Primero se encienden las luces rojas y finalmente la luz verde. Ya que el rojo opacaba a luz verde se optó por apagarlas para que la luz verde destaca.

Código luz de linterna:

```
if (SelectorDeDia == 2) {
    glm::vec3 spotPositionIzquierda = glm::vec3(modelMatrixLambo * glm::vec4(1.0, 0.0, 4.0, 1.0));
    glm::vec3 spotPositionDerecha = glm::vec3(modelMatrixLambo * glm::vec4(-1.0, 0.0, 4.0, 1.0));
    glm::vec3 LuzRotacion = glm::vec3(modelMatrixLambo[2][0], modelMatrixLambo[2][1], modelMatrixLambo[2][2]);

    shaderMulLighting.setInt("spotLightCount", 2);
    shaderTerrain.setInt("spotLightCount", 2);

    shaderMulLighting.setVectorFloat3("spotLights[0].light.ambient", glm::value_ptr(glm::vec3(0.8, 0.8, 0.8)));
    shaderMulLighting.setVectorFloat3("spotLights[0].light.diffuse", glm::value_ptr(glm::vec3(1.0, 1.0, 1.0)));
    shaderMulLighting.setVectorFloat3("spotLights[0].light.specular", glm::value_ptr(glm::vec3(0.8, 0.8, 0.8)));
    shaderMulLighting.setVectorFloat3("spotLights[0].position", glm::value_ptr(spotPositionIzquierda));
    //shaderMulLighting.setVectorFloat3("spotLights[0].direction", glm::value_ptr(glm::vec3(0, -0.75, 1)));
    shaderMulLighting.setVectorFloat3("spotLights[0].direction", glm::value_ptr(normalize(LuzRotacion)));
    shaderMulLighting.setFloat("spotLights[0].constant", 1.0);
    shaderMulLighting.setFloat("spotLights[0].linear", 1.0);
    shaderMulLighting.setFloat("spotLights[0].quadratic", 0.1);
    shaderMulLighting.setFloat("spotLights[0].cutOff", cos(glm::radians(30.0f)));
    shaderMulLighting.setFloat("spotLights[0].outerCutOff", cos(glm::radians(35.0f)));
    shaderTerrain.setVectorFloat3("spotLights[0].light.ambient", glm::value_ptr(glm::vec3(0.8, 0.8, 0.8)));
    shaderTerrain.setVectorFloat3("spotLights[0].light.diffuse", glm::value_ptr(glm::vec3(1.0, 1.0, 1.0)));
    shaderTerrain.setVectorFloat3("spotLights[0].light.specular", glm::value_ptr(glm::vec3(0.8, 0.8, 0.8)));
    shaderTerrain.setVectorFloat3("spotLights[0].position", glm::value_ptr(spotPositionIzquierda));
    shaderTerrain.setVectorFloat3("spotLights[0].direction", glm::value_ptr(normalize(LuzRotacion)));
    //shaderMulLighting.setVectorFloat3("spotLights[0].direction", glm::value_ptr(glm::vec3(0, -0.75, 1)));
    shaderTerrain.setFloat("spotLights[0].constant", 1.0);
    shaderTerrain.setFloat("spotLights[0].linear", 1.0);
    shaderTerrain.setFloat("spotLights[0].quadratic", 0.1);
    shaderTerrain.setFloat("spotLights[0].cutOff", cos(glm::radians(30.0f)));
    shaderTerrain.setFloat("spotLights[0].outerCutOff", cos(glm::radians(35.0f)));
```

Lo más importante en este tipo de luz es que se debe obtener la rotación del coche para que las luces se muevan en conjunto del carro de forma realista.

Cámara

Se utiliza una cámara en tercera persona la cual seguirá al personaje según sea el caso. Para mostrar el recorrido a la pista este es mediante el ave que en el código corresponde al caso 0, en el caso 1 seguirá al automóvil. Cuando se muestran los créditos se hace otro cambio a la cámara. De manera específica, para el coche es posible modificar la distancia de la cámara y también es posible girarla.

Código del cambio de la cámara según el personaje,

```
glm::mat4 projection = glm::perspective(glm::radians(45.0f),
    (float)screenWidth / (float)screenHeight, 0.1f, 100.0f);

if (modelSelected == 0) {
    axis = glm::axis(glm::quat_cast(modelMatrixHooh));
    angleTarget = glm::angle(glm::quat_cast(modelMatrixHooh));
    target = modelMatrixHooh[3];
}
else if (modelSelected == 1) {
    axis = glm::axis(glm::quat_cast(modelMatrixLambo));
    angleTarget = glm::angle(glm::quat_cast(modelMatrixLambo));
    target = modelMatrixLambo[3];
}
if (creditos) {
    //camera->setDistanceFromTarget(-30);
    camera->setDistanceFromTarget(8);
    axis = glm::axis(glm::quat_cast(modelMatrixCamaraFinal));
    angleTarget = glm::angle(glm::quat_cast(modelMatrixCamaraFinal));
    target = modelMatrixCamaraFinal[3];
}

if (std::isnan(angleTarget))
    angleTarget = 0.0;
if (axis.y < 0)
    angleTarget = -angleTarget;
if (girarCamara == 0)
    angleTarget -= glm::radians(180.0f); //Cambia la posicion, me sirve xD*/
if(girarCamara == 1)
{
    angleTarget -= glm::radians(180.0f); //Cambia la posicion, me sirve xD*/
    girarCamara = 2;
}
camera->setCameraTarget(target);
camera->setAngleTarget(angleTarget);
camera->updateCamera();
view = camera->getViewMatrix();
```

Transparencias

Estas solo se aplicación al carro y a los sistemas de partículas. Para usarlo se debe crear el arreglo con la información de los modelos. Seguido de ello se debe indicar la posición. Para renderizar las transparencias de deben usar las banderas de BlendFunc, dependiendo del valor de estas será el efecto de la transparencia.

Código para las transparencias.

```
// Blending model unsorted
std::map<std::string, glm::vec3> blendingUnsorted = {
        {"lambo", glm::vec3(-28.12, 0.0, -62.0)},
        {"fountain", glm::vec3(64.84, 0, 63.28)},
        {"fire", glm::vec3(-5.0, 0.0, -40.0)},
        {"topito", posicionTopito[0]},
        {"nitro", glm::vec3(0.0, 0.0, 0.0)}
};
```

```
blendingUnsorted.find("lambo")->second = glm::vec3(modelMatrixLambo[3]);

std::map<float, std::pair<std::string, glm::vec3>> blendingSorted;
std::map<std::string, glm::vec3>::iterator itblend;
for (itblend = blendingUnsorted.begin(); itblend != blendingUnsorted.end(); itblend++) {
    float distanceFromView = glm::length(camera->getPosition() - itblend->second);
    blendingSorted[distanceFromView] = std::make_pair(itblend->first, itblend->second);
}

/**********
 * Render de las transparencias
 */
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glDisable(GL_CULL_FACE);
```

Niebla

La niebla es un elemento que ayuda a la ambientación de la pista para poder armar los escenarios. Lo primero es la creación de un arreglo con los diferentes colores de la niebla. Posteriormente según sea el número de vueltas se le aplicará la niebla al escenario.

```
std::vector<glm::vec3> ColoresFog = { glm::vec3(0.6, 0.6, 0.6),glm::vec3(0.7, 0.35, 0.175),glm::vec3(0.2,0.2,0.2) };
```

```
shaderMulLighting.setVectorFloat3("fogColor", glm::value_ptr(ColoresFog[SelectorDeDia]));
shaderTerrain.setVectorFloat3("fogColor", glm::value_ptr(ColoresFog[SelectorDeDia]));
shaderSkyBox.setVectorFloat3("fogColor", glm::value_ptr(ColoresFog[SelectorDeDia]));
```

Partículas

Para las partículas se tiene dos sistemas uno de ellos tiene retroalimentación lo que ocasiona que antes que se termine de ejecutar todo el sistema se creen nuevas partículas por lo que permite simular el fuego. El otro tipo de sistema no tiene retroalimentación por lo que no se generan nuevas partículas, por lo que empieza, termina totalmente y vuelve a empezar.

Código sistemas de partículas, se anexan las dos partes más importantes del sistema la inicialización del buffer de partículas donde además se define el comportamiento del sistema mediante ecuaciones de un movimiento físico, y el código de la parte donde se realiza la ejecución, en esta parte se le da la matrix para posicionarlo, además de controlar su tiempo de ejecución.

```
glm::vec3 v(0.0f);
float velocity, theta, phi;
GLfloat *data = new GLfloat[nParticlesTopito * 3];
for (unsigned int i = 0; i < nParticlesTopito; i++) {

    theta = glm::mix(0.0f, glm::pi<float>() / 6.0f, ((float)rand() / RAND_MAX));
    phi = glm::mix(0.0f, glm::two_pi<float>(), ((float)rand() / RAND_MAX));

    v.x = sinf(theta) * cosf(phi);
    v.y = cosf(theta);
    v.z = sinf(theta) * sinf(phi);

    velocity = glm::mix(0.6f, 0.8f, ((float)rand() / RAND_MAX));
    v = glm::normalize(v) * velocity;

    data[3 * i] = v.x;
    data[3 * i + 1] = v.y;
    data[3 * i + 2] = v.z;
}
glBindBuffer(GL_ARRAY_BUFFER, initVelTopito);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, data);

// Fill the start time buffer
delete[] data;
data = new GLfloat[nParticlesTopito];
float time = 0.0f;
float rate = 0.0009f;
for (unsigned int i = 0; i < nParticlesTopito; i++) {
    data[i] = time;
    time += rate;
}
glBindBuffer(GL_ARRAY_BUFFER, startTimeTopito);
glBufferSubData(GL_ARRAY_BUFFER, 0, nParticlesTopito * sizeof(float), data);
```

```
else if (renderParticles && it->second.first.compare("topito") == 0 && ActivaTopito == 0) {
glm::mat4 modelMatrixParticlesTopito = glm::mat4(1.0);
modelMatrixParticlesTopito = glm::translate(modelMatrixParticlesTopito, it->second.second);
modelMatrixParticlesTopito[3][1] = terrain.getHeightTerrain(modelMatrixParticlesTopito[3][0], modelMatrixParticlesTopito[3][2]) - 0.5;
modelMatrixParticlesTopito = glm::scale(modelMatrixParticlesTopito, glm::vec3(10.0, 10.0, 10.0));
currTimeParticlesAnimationTopito = TimeManager::Instance().GetTime();
if (currTimeParticlesAnimationTopito - lastTimeParticlesAnimationTopito > 10.0) {
    lastTimeParticlesAnimationTopito = currTimeParticlesAnimationTopito;
    ActivaTopito = 1;
}
//glDisable(GL_DEPTH_TEST);
glDepthMask(GL_FALSE);
// Set the point size
glPointSize(12.0f);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textureParticleTopitoID);
shaderParticlesTopito.turnOn();
shaderParticlesTopito.setFloat("Time", float(currTimeParticlesAnimationTopito - lastTimeParticlesAnimationTopito));
shaderParticlesTopito.setFloat("ParticleLifetime", 3.5f);
shaderParticlesTopito.setInt("ParticleTex", 0);
shaderParticlesTopito.setVectorFloat3("Gravity", glm::value_ptr(glm::vec3(0.0f, -0.3f, 0.0f)));
shaderParticlesTopito.setMatrix4("model", 1, false, glm::value_ptr(modelMatrixParticlesTopito));
glBindVertexArray(VAOParticlesTopito);
glDrawArrays(GL_POINTS, 0, nParticlesTopito);
glDepthMask(GL_TRUE);
//glEnable(GL_DEPTH_TEST);
shaderParticlesTopito.turnOff();

}
```

Código sistema con retroalimentación.

```
std::vector<GLfloat> initialAge(nParticlesFire);
float rate = particleLifetime / nParticlesFire;
for (unsigned int i = 0; i < nParticlesFire; i++) {
    int index = i - nParticlesFire;
    float result = rate * index;
    initialAge[i] = result;
}
// Shuffle them for better looking results
//Random::shuffle(initialAge);
auto rng = std::default_random_engine{};
std::shuffle(initialAge.begin(), initialAge.end(), rng);
glBindBuffer(GL_ARRAY_BUFFER, age[0]);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, initialAge.data());

glBindBuffer(GL_ARRAY_BUFFER, 0);

// Create vertex arrays for each set of buffers
glGenVertexArrays(2, particleArray);

// Set up particle array 0
glBindVertexArray(particleArray[0]);
glBindBuffer(GL_ARRAY_BUFFER, posBuf[0]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, velBuf[0]);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);

glBindBuffer(GL_ARRAY_BUFFER, age[0]);
glVertexAttribPointer(2, 1, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(2);

// Set up particle array 1
glBindVertexArray(particleArray[1]);
glBindBuffer(GL_ARRAY_BUFFER, posBuf[1]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, velBuf[1]);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(1);
```

```
else if (renderParticles && it->second.first.compare("fire") == 0 && vida <= 0) {
    /**********
     * Init Render particles systems
     */
    lastTimeParticlesAnimationFire = currTimeParticlesAnimationFire;
    currTimeParticlesAnimationFire = TimeManager::Instance().GetTime();

    shaderParticlesFire.setInt("Pass", 1);
    shaderParticlesFire.setFloat("Time", currTimeParticlesAnimationFire);
    shaderParticlesFire.setFloat("DeltaT", currTimeParticlesAnimationFire - lastTimeParticlesAnimationFire);

    glActiveTexture(GL_TEXTURE1);
    glBindTexture(GL_TEXTURE_1D, texId);
    glEnable(GL_RASTERIZER_DISCARD);
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, feedback[drawBuf]);
    glBeginTransformFeedback(GL_POINTS);
    glBindVertexArray(particleArray[1 - drawBuf]);
    glVertexAttribDivisor(0, 0);
    glVertexAttribDivisor(1, 0);
    glVertexAttribDivisor(2, 0);
    glDrawArrays(GL_POINTS, 0, nParticlesFire);
    glEndTransformFeedback();
    glDisable(GL_RASTERIZER_DISCARD);

    shaderParticlesFire.setInt("Pass", 2);
    glm::mat4 modelFireParticles = modelMatrixLambo;
    modelFireParticles = glm::translate(modelFireParticles, glm::vec3(-0.5, 0, 0.0));
    //          modelFireParticles = glm::translate(modelFireParticles, it->second.second);
    shaderParticlesFire.setMatrix4("model", 1, false, glm::value_ptr(modelFireParticles));

    shaderParticlesFire.turnOn();
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, textureParticleFireID);
    glDepthMask(GL_FALSE);
    glBindVertexArray(particleArray[drawBuf]);
    glVertexAttribDivisor(0, 1);
    glVertexAttribDivisor(1, 1);
    glVertexAttribDivisor(2, 1);
    glDrawArraysInstanced(GL_TRIANGLES, 0, 6, nParticlesFire);
    glBindVertexArray(0);
    glDepthMask(GL_TRUE);
    drawBuf = 1 - drawBuf;
    shaderParticlesFire.turnOff();
```

Sombras

El utilizar sombras causa un efecto que ayuda al videojuego a verse más realista para ello se hace uso del buffer de profundidad. Se debe utilizar el shadowDepth para crear los efectos, para lo cual se cargan en la función prepareDepthScene los modelos a los cuales se les quiera dar sombra, indicando el shader.

Código para generar sombras:

```
/*******************************************
 * 1.- We render the depth buffer
 *******************************************/
glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// render scene from light's point of view
glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glClear(GL_DEPTH_BUFFER_BIT);
//glCullFace(GL_FRONT);
prepareDepthScene();
renderScene(false);
//glCullFace(GL_BACK);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

```
void prepareDepthScene() {

    skyboxSphere.setShader(&shaderDepth);

    terrain.setShader(&shaderDepth);

    // Lamborginhi
    modelLambo.setShader(&shaderDepth);
    modelLamboLeftDor.setShader(&shaderDepth);
    modelLamboRightDor.setShader(&shaderDepth);
    modelLamboFrontLeftWheel.setShader(&shaderDepth);
    modelLamboFrontRightWheel.setShader(&shaderDepth);
    modelLamboRearLeftWheel.setShader(&shaderDepth);
    modelLamboRearRightWheel.setShader(&shaderDepth);

    modelEclipse.setShader(&shaderDepth);

    //Coins
    for (int i = 0; i < TAMCOIN; i++)
    {
        modelCoin[i].setShader(&shaderDepth);
    }

    //Gema
    for (int i = 0; i < TAMGEMA; i++)
    {
        modelGema[i].setShader(&shaderDepth);
    }

    //Lamp models
    modelLamp1.setShader(&shaderDepth);
    modelLamp2.setShader(&shaderDepth);
    modelLampPost2.setShader(&shaderDepth);
```

OpenAL

Esta API ayuda a generar sonido espacial por lo que entre más cerca estemos del objeto más fuerte será el sonido adicionalmente si el sonido se encuentra en el lado derecho solo se escuchara en la bocina de ese lado.

Para usar Open AL se ocupó el siguiente código:

Se declaran el número máximo de sonidos a usar. Se crean tres arreglos para quien lo va a escuchar los cuales corresponden con listener, por cada sonido se declara dos arreglos uno de posicion y otro de velocidad.

```
// OpenAL Defines
#define NUM_BUFFERS 31
#define NUM_SOURCES 31
#define NUM_ENVIRONMENTS 1
// Listener
ALfloat listenerPos[] = { 0.0, 0.0, 4.0 };
ALfloat listenerVel[] = { 0.0, 0.0, 0.0 };
ALfloat listenerOri[] = { 0.0, 0.0, 1.0, 0.0, 1.0, 0.0 };
// Source 0
ALfloat source0Pos[] = { 0.0, 0.0, 0.0 };
ALfloat source0Vel[] = { 0.0, 0.0, 0.0 };
```

Se inicializan los buffers son el número máximo de sonidos a usar. Además de unas variables para su uso, posteriormente se declaran vectores binarios para controlar los sonidos.

```
// Buffers
ALuint buffer[NUM_BUFFERS];
ALuint source[NUM_SOURCES];
ALuint environment[NUM_ENVIRONMENTS];
// Configs
ALsizei size, freq;
ALenum format;
ALvoid *data;
int ch;
ALboolean loop;
std::vector<bool> sourcesPlay = { true, true, true, true, true, true, true };
std::vector<bool> sourcesPlayCoin = { true, true, true, true, true, true, true, true, true, true, true, true, true, true, true, true };
std::vector<bool> sourcesPlayGema = { true, true, true, true, true, true, true };
```

Posteriormente, con los datos antes declarados se inicializan, los datos para el oyente, así como cargar en los buffer los sonidos a utilizar.

```
/*****************************************
 * OpenAL init
 *****************************************/
alutInit(0, nullptr);
alListenerfv(AL_POSITION, listenerPos);
alListenerfv(AL_VELOCITY, listenerVel);
alListenerfv(AL_ORIENTATION, listenerOri);
alGetError(); // clear any error messages
if (alGetError() != AL_NO_ERROR) {
    printf("- Error creating buffers !!\n");
    exit(1);
}
else {
    printf("init() - No errors yet.");
}
// Config source 0
// Generate buffers, or else no sound will happen!
alGenBuffers(NUM_BUFFERS, buffer);
buffer[0] = alutCreateBufferFromFile("../sounds/fountain.wav"); //Fuente
buffer[1] = alutCreateBufferFromFile("../sounds/fire.wav"); // Fuego
buffer[2] = alutCreateBufferFromFile("../sounds/carAcel.wav"); //Acelerar
buffer[3] = alutCreateBufferFromFile("../sounds/nelson-aha.wav"); //Perder
buffer[4] = alutCreateBufferFromFile("../sounds/youWin.wav"); //Ganar
buffer[5] = alutCreateBufferFromFile("../sounds/gracias.wav"); //Creditos
buffer[6] = alutCreateBufferFromFile("../sounds/chocar.wav"); //Chocar
buffer[7] = alutCreateBufferFromFile("../sounds/coin.wav"); //Ganar
buffer[8] = alutCreateBufferFromFile("../sounds/gema.wav"); //Ganar
```

Después se inicializan los sonidos, con su posición, velocidad y el sonido que el corresponde. De manera específica, en la parte de AL_LOOPING, al colocar AL_TRUE el sonido se va a estar repitiendo uno y otra vez, de lo contrario, si se coloca AL_FALSE, se reproduce una vez y se para.

```
alSourcef(source[0], AL_PITCH, 1.0f);
alSourcef(source[0], AL_GAIN, 2.0f);
alSourcefv(source[0], AL_POSITION, source0Pos);
alSourcefv(source[0], AL_VELOCITY, source0Vel);
alSourcei(source[0], AL_BUFFER, buffer[0]);
alSourcei(source[0], AL_LOOPING, AL_TRUE);
alSourcef(source[0], AL_MAX_DISTANCE, 2000);

alSourcef(source[1], AL_PITCH, 1.0f);
alSourcef(source[1], AL_GAIN, 2.0f);
alSourcefv(source[1], AL_POSITION, source1Pos);
alSourcefv(source[1], AL_VELOCITY, source1Vel);
alSourcei(source[1], AL_BUFFER, buffer[1]);
alSourcei(source[1], AL_LOOPING, AL_TRUE);
alSourcef(source[1], AL_MAX_DISTANCE, 2000);
```

Se le asigna el sonido al modelo correspondiente a través de su matriz

```
if (presionaTecla)
{
    //std::cout << "Creditos: " << creditos << std::endl;
    source0Pos[0] = modelMatrixFountain[3].x;
    source0Pos[1] = modelMatrixFountain[3].y;
    source0Pos[2] = modelMatrixFountain[3].z;
    alSourcefv(source[0], AL_POSITION, source0Pos);

    source2Pos[0] = modelMatrixLambo[3].x;
    source2Pos[1] = modelMatrixLambo[3].y;
    source2Pos[2] = modelMatrixLambo[3].z;
    alSourcefv(source[2], AL_POSITION, source2Pos);
```

Finalmente se activan los sonidos, si se desean para en algún momento se utiliza alSourceStop y se indica el sonido que se desea parar.

```
//Indica como reproducir el sonido
for (unsigned int i = 0; i < 2; i++) {
    if (sourcesPlay[i]) {
        sourcesPlay[i] = false;
        alSourcePlay(source[i]);
```

```
if ((avanza || retrocede) && sourcesPlay[2])
{
    sourcesPlay[2] = false;
    alSourcePlay(source[2]);
}
else if (avanza == false && retrocede == false) {
    sourcesPlay[2] = true;
    alSourceStop(source[2]);
```

GLFW para Joystick

A través del uso de glfwJoystickPresent es posible detectar si un control está conectado, en este caso es de un PlayStation 4, con la función glfwGetJoystickAxes es posible obtener los valores de los joysticks (derecho e izquierdo) y los gatillos, mientras que con glfwGetJoystickButtons se puede acceder a los botones.

```
//Joystick
if (glfwJoystickPresent(GLFW_JOYSTICK_1)) {
    //std::cout << "Joystick conected" << std::endl;

    int axesCount;
    const float *axes = glfwGetJoystickAxes(GLFW_JOYSTICK_1, &axesCount);

    int buttonCount;
    const unsigned char *buttons = glfwGetJoystickButtons(GLFW_JOYSTICK_1, &buttonCount);
```

```
if (modelSelected == 1 && (axes[0] < 0) && (avanza || retrocede)) {
    modelMatrixLambo = glm::translate(modelMatrixLambo, glm::vec3(-0.83827, 0.17224, 1.47627));
    modelMatrixLambo = glm::rotate(modelMatrixLambo, glm::radians(2.0f), glm::vec3(0, 1, 0));
    modelMatrixLambo = glm::translate(modelMatrixLambo, glm::vec3(0.83827, -0.17224, -1.47627))
    //std::cout << "Right Stick X Axis Gira izquierda: " << axes[0] << std::endl; // tested wit
}
else if (modelSelected == 1 && (axes[0] > 0) && (avanza || retrocede)) {
    modelMatrixLambo = glm::translate(modelMatrixLambo, glm::vec3(0.83827, 0.17224, 1.47627));
    modelMatrixLambo = glm::rotate(modelMatrixLambo, -glm::radians(2.0f), glm::vec3(0, 1, 0));
    modelMatrixLambo = glm::translate(modelMatrixLambo, glm::vec3(-0.83827, -0.17224, -1.47627)
```

Con axes[0] se obtiene el valor del joystick izquierdo en el eje x, si su valor es positivo gira a la derecha, si el valor es negativo gira a la izquierda.

Con axes[5] se obtiene el valor del gatillo derecho, si su valor es mayor a 0 permite acelerar, si se deja de apretar se frena. Con el gatillo izquierdo (axes[4]) es posible ir de reversa.

```
if (modelSelected == 1 && (axes[5] > 0) && vida > 0
    avanza = true;
    if (velocidad <= 0.5) {
        frenado1 = FALSE;
        velocidad += 0.05;
    }
    //std::cout << "Right Trigger/R2: " << axes[5] <
}
else if (modelSelected == 1 && (axes[4] > 0) && vida
    retrocede = true;
    if (velocidad >= -0.5) {
        frenado2 = FALSE;
        velocidad -= 0.05;
```

Por último, con buttons se accede a los diferentes botones:

- buttons[0]: X
- buttons[1]: O
- buttons[2]: □
- buttons[3]: △

Solo se detecta si se presiona.

```
if (GLFW_PRESS == buttons[0] && presionaTecla == false) {
    presionaTecla = true; //Para desactivar la pantalla de inicio
    ActivaRecorrido = true;
    modelSelected = 0;
}
if (GLFW_PRESS == buttons[3] && modelSelected == 0) {
    ActivaRecorrido = false;
    modelSelected = 1;
    record = true;
    cambiarCamara++;
}
```

Resultados

Obtuvimos un videojuego totalmente funcional, cumpliendo las funciones establecidas en la presentacion del dia 14 de abril de 2020.

Además se logró poner en práctica todo lo realizado en la materia, lo cual, si en un futuro se desea trabajar en el ámbito de los videojuegos, nos será de gran ayuda, ya que, si bien existen motores que facilitan la realización de estos, el conocer las bases de ellos permite que su uso sea más sencillo.

De igual manera, el aprender a usar Blender es de gran utilidad, debido a que es uno de los programas más utilizados para la animación debido a sus diferentes herramientas que posee.

Trabajo a futuro

Como metas a futuros tenemos:

- Mejorar la animación del coche, aumentar la física empleada de forma que se tenga una animación más realista cuando éste gire, choque, salte.
- Hacer que el segundo coche tenga un recorrido más óptimo y que este pueda ser una competencia para el carro principal.
- Aumentar el número de pistas para que no solamente se tenga una.
- Poner un mapa de referencia para que el jugador observe su posición actual de la pista.
- Incluir plataformas de nitro y que no solo dependa de las monedas, en este proyecto ya no se incluyó debido a que a nuestro parecer, se saturaba la pista.

Conclusiones

Aplicamos todos los temas vistos en clases, cada tema logró mejorar diferentes aspectos del videojuego, resaltando la importancia de conocerlo y aplicarlo adecuadamente. Este proyecto fue un desafío que nos colocamos donde logramos alcanzar las metas impuestas, tuvimos diversas dificultades en la realización de este proyecto algunas de ellas fueron la construcción y delimitación de la pista, que el carro se adecuara correctamente a las alturas del terreno, manejar texto.

Para la construcción de la pista se ocupó GIMP en el cual se trabajó con cuidado para poder dibujar la pista de forma que se lograra ser estética, una vez con el mapa realizado, la construcción de los dos mapas restantes fue sencillo. Para delimitar la pista se empleó un mapa lógico el cual fue sugerido por nuestro asesor. Dicho para permite delimitar las zonas en las que el automóvil puede tocar, ciertamente es una solución sencilla y práctica.

La mecánica del coche fue otro aspecto complicado del videojuego, en especial para que el vehículo tuviera las inclinaciones adecuadas según su posición en la pista. En esta parte se hicieron diferentes implementaciones, sin embargo, no eran las más adecuadas, la implementación final se logró gracias a nuestro asesor, quien nos dio una mejor explicación detallada del problema.

Para la manipulación del texto se utilizó la librería freetype, la cual ayudó a la construcción de texto de forma sencilla adicionalmente se pudo crear la barra de vida y el nitro con esta librería.

Por otro lado, gracias a la librería glfw fue posible utilizar un control de PlayStation 4, lo cual hace más fácil la conducción del coche. Esta librería nos permitió manejar el joystick, los gatillos y los botones.

El proyecto requiere tiempo y esfuerzo, ha sido uno de los proyectos más interesantes realizados en la carrera, donde aprendimos a utilizar opengl, openal, y freetype, además de otros software como lo son blender y gimp los cuales fueron fundamentales para la construcción del proyecto. En este trabajo desarrollamos nuestra creatividad para poder plantear el objetivo del videojuego, así como para solucionar diferentes problemas del proyecto. También aumentó nuestra determinación, ya que nos enfocamos en cumplir con los aspectos del proyecto. Hemos quedado satisfechos con el resultado final de nuestro trabajo al poder cumplir con los objetivos planteados y tener nuestro propio videojuego.