SESIÓN 9

Uso del programa

En este espacio detallaré cada tecla que se puede usar en el programa: ### Movimientos de la Cámara Para esto primero debemos pulsar una tecla para activar el movimiento y después arrastrar el raton por la ventana manteniendo pulsado el botón izquierdo del ratón. - P -> Pan - T -> Tilt - D -> Dolly - C -> Crane - O -> Orbit - Z -> Zoom ### Control de Modelos - 1 -> Añande Dado - 2 -> Añade Ajax - 3 -> Añade Mandalorian - 4 -> Añade Pantera - 5 -> Añade Vaca - Q -> Elimina el ultimo modelo añadido - M -> Cambia el modelo seleccionado (se mueve por el vector y si llega al final vuelve al inicio) - Flecha Direccional Izq/Drch -> Cambia el eje en el que se aplica la traslación - Flecha Direccional Arriba/Abajo -> Traslada el objeto en el eje activo - G -> Cambia de modo Alambre a modo Relleno o viceversa

En esta nueva práctica se pide la implementación de texturas en la aplicación y para ello seguiremos los siguientes pasos. ## Clase Textura Esta nueva clase contendrá un id para la textura y un método de carga de archivos .png tendrá la siguiente estructura:

```
class Textura {
private:
    GLuint idTextura;
public:
    Textura(int id);

    ~Textura();

    void cargarTextura(std::string rutaDeFichero);

    void setKsTextura(glm::vec3 *KsTextura);

    glm::vec3 *getKsTextura();

    GLuint getIdTextura();
};
```

La siguiente función se encargará de cargar la imagen mediante la libreria lodepng, además se usa la función proporcionada para dar la vuelta a la imagen ya que como se menciona en el guión esta se carga del revés, finalmente se configuran los parámetros de la textura mediante funciones de OpenGL tal y como se muestra en el módulo de aprendizaje

```
void Textura::cargarTextura(std::string rutaDeFichero) {
    // Generamos el identificador de la textura
  glGenTextures(1, &this->idTextura);
    glBindTexture(GL_TEXTURE_2D, this->idTextura);
```

```
    // Cargamos la imagen
  std::vector<unsigned char> imagen; // Los píxeles de la imagen
  unsigned ancho, alto;
    unsigned error = lodepng::decode(imagen, ancho, alto, rutaDeFichero);
    if (error) {
        std::string mensaje = rutaDeFichero + " no se pudo cargar";
        throw std::runtime_error(mensaje);
    }
    // La textura se carga del revés, así que vamos a darle la vuelta
  unsigned char *imgPtr = &imagen[0];
    int numeroDeComponentesDeColor = 4;
    int incrementoAncho = ancho * numeroDeComponentesDeColor; // Ancho en bytes
  unsigned char *top = nullptr;
    unsigned char *bot = nullptr;
    unsigned char temp = 0;
    for (int i = 0; i < alto / 2; i++) {
        top = imgPtr + i * incrementoAncho;
        bot = imgPtr + (alto - i - 1) * incrementoAncho;
        for (int j = 0; j < incrementoAncho; j++) {
            temp = *top;
            *top = *bot;
            *bot = temp;
            ++top;
            ++bot;
        }
    }

    //AJUSTES DE LA TEXTURA

 // Cómo resolver la minificación. En este caso, le decimos que utilice mipmaps, y que aplic
    // Cómo resolver la magnificación. En este caso, le decimos que utilice mipmaps, y que a
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    // Cómo pasar de coordenadas de textura a coordenadas en el espacio de la textura en hor
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    // Cómo pasar de coordenadas de textura a coordenadas en el espacio de la textura en ver
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // Transferimos la información de la imagen. En este caso, la imagen está guardada en st

 // Finalmente, pasamos la textura a OpenGL  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, ancho,
                 GL_UNSIGNED_BYTE, imagen.data());

    // Generamos los mipmaps
  glGenerateTextureMipmap(this->idTextura);
}
```

Clase Material

Las texturas cargadas se agregarán mediante los materiales, ya que ahora esta clase tiene un puntero a una textura, puede quedarse como nullptr y entonces el progrma entenderá que hay que activar las subrutinas para coger el color del material o tener una dirección de memoria a una textura y entonces entenderá que se deben coger los colores de esta. mas adelante se explica como se hace esta decisión.

```
class Material {
private:
    //resto de los atributos
    Textura *textura;

public:
  //resto de las funciones
    Textura *getTextura();
};
```

Clase Modelo

Primero en la clase shaderProgram modificaremos el estruct del vertice para permitir guardar la coordenada de textura.

```
struct Vertice {
    glm::vec3 posicion, normal;
    glm::vec2 coordTextura;
};
```

Una vez hecho esto ahora si en la clase modelo debemos cambiar de nuevo la funcion de procesarMalla() para agregar las coordenadas de textura si es que tiene.

```
Malla *Modelo::procesarMalla(aiMesh *malla, const aiScene *escena) {
//resto del codigo

        //textura
  if (malla->mTextureCoords[0]) {
            glm::vec2 vector2;
            vector2.x = malla->mTextureCoords[0][i].x;
            vector2.y = malla->mTextureCoords[0][i].y;
            vertice.coordTextura = vector2;
        } else {
            vertice.coordTextura = glm::vec2(0.0f, 0.0f);
        }

//resto del codigo
}
```

Clase VAO

Debemos modificar la función crearModelo() para permitir agregar las coordenadas de textura a los shader

```
void VAO::crearModelo() {
    //ver si falla algun identificador
  if (idVAO > UINT_MAX || idVBO > UINT_MAX || idVBOn > UINT_MAX || idIBO > UINT_MAX) {
        throw std::runtime_error("Error: crearModelo fallo en el identificador");
    }

    //entrelazado
  glGenVertexArrays(1, &this->idVAO);
    glBindVertexArray(this->idVAO);
    glGenBuffers(1, &this->idVBOn);
    glBindBuffer(GL_ARRAY_BUFFER, this->idVBOn);
    glGenBuffers(1, &this->idIBO);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->idIBO);


    //entrelazado
  glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertice), (void *) offsetof(Verti
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertice), (void *) offsetof(Verti
    glEnableVertexAttribArray(1);
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(Vertice), (void *) offsetof(Verti
    glEnableVertexAttribArray(2);

    //entrelazado
  glBufferData(GL_ARRAY_BUFFER, this->vertices.size() * sizeof(Vertice), &this->vertices[0],
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, this->indices.size() * sizeof(unsigned int), &this


}
```

Clase Malla

Agregaremos el codigo necesario para poder decir entre las subrutinas de modo alambre, color material o color textura, los dos ultimos se decidirán de forma que si el material de ese modelo tiene una textura asociada, entonces, se eligirá el color de la textura y si no el del materias. Para el modo alambre tenemos el botón asociado.

```
void Malla::dibujar(ShaderProgram *shaderProgram, Textura *textura) {
    if (this->alambre) {
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

```
        //usar subrutina
  GLuint activar[2] = {0, 0};

        GLint posUniform = shaderProgram->getUniformLocation("ColorUniform");
        GLuint aux = shaderProgram->getSubroutineIndex("ColorAlambre");
        activar[posUniform] = aux;

        glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 2, activar);
    } else {
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    }

    glBindVertexArray(this->vao->getIdVAO());
    if (textura != nullptr) {
        if (textura->getIdTextura() == 0) {
            GLint posicion = glGetUniformLocation(shaderProgram->getIdSP(), "muestreador");
            glUniform1i(posicion, 0);
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D, textura->getIdTextura());
        } else {
            GLint posicion = glGetUniformLocation(shaderProgram->getIdSP(), "muestreador");
            glUniform1i(posicion, 1);
            glActiveTexture(GL_TEXTURE1);
            glBindTexture(GL_TEXTURE_2D, textura->getIdTextura());
        }

    }
    glDrawElements(GL_TRIANGLES, this->indices.size(), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);

}
```

Clase luz

Debido a la utilización de varias subrutinas ahora deberemos de modificar la forma de escoger uniforme y subrutina al igual que en la función anterior de la clase malla. Para esto me he basado en el guión de prácticas.

```
void Luz::aplicarLuz(ShaderProgram *shaderProgram) {
    GLuint activar[2] = {0, 0};
    if (!textura) {
        //buscar posicion de la subrutina
  GLint posUniform = shaderProgram->getUniformLocation("uniformEleccionColor");
        GLuint aux = shaderProgram->getSubroutineIndex("colorMaterial");

        //activar subrutina color material
```

```
activar[posUniform] = aux;
  } else {
      //buscar posicion de la subrutina
GLint posUniform = shaderProgram->getUniformLocation("uniformEleccionColor");
      GLuint aux = shaderProgram->getSubroutineIndex("colorTextura");

      //activar subrutina color textura
activar[posUniform] = aux;
  }

  if (tipo == 0) {
      //setear uniformes
shaderProgram->setUniform("Ia", *this->propiedades->getIa());

      //buscar posicion de la subrutina
GLint posUniform = shaderProgram->getUniformLocation("ColorUniform");
      GLuint aux = shaderProgram->getSubroutineIndex("ColorLuzAmbiente");

      //activar subrutina color luz ambiente
activar[posUniform] = aux;
  }
  if (tipo == 1) {
      //setear uniformes
shaderProgram->setUniform("Id", *this->propiedades->getId());
      shaderProgram->setUniform("Is", *this->propiedades->getIs());
      shaderProgram->setUniform("posicionLuz", *this->propiedades->getP());
      shaderProgram->setUniform("Shininess", this->propiedades->getS());

      //buscar posicion de la subrutina
GLint posUniform = shaderProgram->getUniformLocation("ColorUniform");
      GLuint aux = shaderProgram->getSubroutineIndex("ColorLuzPuntual");

      //activar subrutina color luz puntual
activar[posUniform] = aux;
  }
  if (tipo == 2) {
      //setear uniformes
shaderProgram->setUniform("Id", *this->propiedades->getId());
      shaderProgram->setUniform("Is", *this->propiedades->getIs());
      shaderProgram->setUniform("posicionLuz", *this->propiedades->getP());
      //calcular direccion desde posicion del foco al origen
glm::vec3 direccionLuz = glm::normalize(-(*this->propiedades->getP()));
      shaderProgram->setUniform("direccionLuz", direccionLuz);
      shaderProgram->setUniform("Shininess", this->propiedades->getS());

      //buscar posicion de la subrutina
```

```
  GLint posUniform = shaderProgram->getUniformLocation("ColorUniform");
        GLuint aux = shaderProgram->getSubroutineIndex("ColorLuzDireccional");

        //activar subrutina color luz direccional
  activar[posUniform] = aux;
    }
    if (tipo == 3) {
        //setear uniformes
  shaderProgram->setUniform("Id", *this->propiedades->getId());
        shaderProgram->setUniform("Is", *this->propiedades->getIs());
        shaderProgram->setUniform("posicionLuz", *this->propiedades->getP());
        //calcular direccion desde posicion del foco al origen
  glm::vec3 direccionLuz = glm::normalize(-(*this->propiedades->getP()));
        shaderProgram->setUniform("direccionLuz", direccionLuz);
        shaderProgram->setUniform("anguloApertura", this->propiedades->getGamma());
        shaderProgram->setUniform("Shininess", this->propiedades->getS());

        //buscar posicion de la subrutina
  GLint posUniform = shaderProgram->getUniformLocation("ColorUniform");
        GLuint aux = shaderProgram->getSubroutineIndex("ColorLuzFocal");

        //activar subrutina color luz focal
  activar[posUniform] = aux;
    }

    //activar subrutinas
  glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 2, activar);
}
```

Clase Renderer

Para poder cargar la textura ahroa debemos asignar una textura al material que después asignaremos al modelo. He creado una nueva función para crear los materiales y estructurar mejor el código.

```
void Renderer::crearMateriales() {
    Material *material = new Material();
    Material *material2 = new Material();
    material2->cargarTextura("../dado.png", 0);
    Material *material3 = new Material();
    material3->cargarTextura("../spot_texture.png", 1);
    this->materiales.push_back(material);
    this->materiales.push_back(material2);
    this->materiales.push_back(material3);
}
```

Además ahora como veremos más adelante en los shader debemos pasar nuevos

uniformo como la matriz para las texturas tal y como se muestra en el modulo de aprendizaje

```
void Renderer::refrescar() {
//resto del código

  glm::mat4 matrizMod = this->modelos[j]->getMallas()[k]->getMatrizModelado();
                glm::mat4 matrizVis = this->camara->mVision;
                glm::mat4 matrizProj = this->camara->mProy;

                glm::mat4 mat = matrizProj * matrizVis * matrizMod;
                glm::mat4 mat2 = matrizVis * matrizMod;
                glm::mat4 mat3 = glm::transpose(glm::inverse(mat2));

                //set uniform
  this->modelos[j]->getShaderProgram()->setUniform(mat, "mvpMatrix");
                this->modelos[j]->getShaderProgram()->setUniform(mat2, "mModelView");
                this->modelos[j]->getShaderProgram()->setUniform(mat3, "matrizMVit");
                this->modelos[j]->getShaderProgram()->setUniform("Ka",
                                                                 *this->modelos[j]->getMater
                this->modelos[j]->getShaderProgram()->setUniform("Kd",
                                                                 *this->modelos[j]->getMater
                this->modelos[j]->getShaderProgram()->setUniform("Ks",
                                                                 *this->modelos[j]->getMater
                //aplicamos la luz
  luces[i]->aplicarLuz(this->modelos[j]->getShaderProgram());
            }

            //resto del código
}
```

Vertex Shader

Ahora tenemos un nuevo Layout para la textura tal y como hemos implementado en la clase VAO, además tendremos la nueva uniforme para la matriz y el nuevo out que será la coordenada de textura que necesita el fragment para seleccionar el color tal y como se muestra en el modulo de aprendizaje

```
#version 410
layout (location = 0) in vec3 vPosicion;
layout (location = 1) in vec3 vNormal;
layout (location = 2) in vec2 vCoordTextura;

uniform mat4 mvpMatrix;
uniform mat4 mModelView;
uniform mat4 matrizMVit;
```

```
out vec3 posicion;
out vec3 normal;
out vec2 coordTextura;

void main() {
    posicion = vec3(mModelView * vec4(vPosicion, 1.0));
    normal = vec3(matrizMVit * vec4(vNormal, 0.0));
    coordTextura = vCoordTextura;
    gl_Position = mvpMatrix * vec4(vPosicion, 1.0);
}
```

Fragment Shader

El fragment cambia significativamente ya que ahora tendremos dos uniform con sus diferentes subrutinas, esto se hace para con el primer uniform seleccionar entre el color del alambre, el color del material o el color de la textura mediante el muestreador. Una vez que tenemos seleccionado el color base seleccionaremos con el otro uniform el color que debemos calcular según el tipo de luz.

```
#version 410

in vec3 posicion;
in vec3 normal;
in vec2 coordTextura;

uniform vec3 Ka;
uniform vec3 Kd;
uniform vec3 Ks;
uniform vec3 Ia;
uniform vec3 Id;
uniform vec3 Is;
uniform float Shininess;
uniform vec3 posicionLuz;
uniform vec3 direccionLuz;
uniform float anguloApertura;

uniform sampler2D muestreador;

layout (location = 0) out vec4 colorFragmento;

subroutine vec3 elegirColor();
subroutine uniform elegirColor uniformEleccionColor;

subroutine(elegirColor)
vec3 colorFijo()
{
```

```
    return vec3(1.0, 0.0, 0.0);
}

subroutine(elegirColor)
vec3 colorTextura()
{
    return vec3(texture(muestreador, coordTextura));
}

subroutine(elegirColor)
vec3 colorMaterial()
{
    return vec3(Kd);
}

subroutine vec4 CalcularColor(vec3 colorDePartida);
subroutine uniform CalcularColor ColorUniform;

//color rojo para alambre
subroutine(CalcularColor)
vec4 ColorAlambre(vec3 colorDePartida)
{
return vec4(colorDePartida, 1.0);
}

subroutine(CalcularColor)
vec4 ColorLuzAmbiente(vec3 colorDePartida)
{
    vec3 colorAmbiente = colorDePartida * Ia;
    return vec4(colorAmbiente, 1.0);
}

subroutine(CalcularColor)
vec4 ColorLuzPuntual(vec3 colorDePartida)
{
    vec3 N = normalize(normal);
    vec3 L = normalize(posicionLuz - posicion);
    vec3 V = normalize(-posicion);
    vec3 R = reflect(-L, N);

    vec3 colorDifuso = colorDePartida * Id * max(dot(N, L), 0.0);
    vec3 colorEspecular = Ks * Is * pow(max(dot(R, V), 0.0), Shininess);

    return vec4(colorDifuso + colorEspecular, 1.0);
}
```

```
subroutine(CalcularColor)
vec4 ColorLuzDireccional(vec3 colorDePartida)
{
    vec3 N = normalize(normal);
    vec3 L = -direccionLuz;
    vec3 V = normalize(-posicion);
    vec3 R = reflect(-L, N);

    vec3 colorDifuso = colorDePartida * Id * max(dot(N, L), 0.0);
    vec3 colorEspecular = Ks * Is * pow(max(dot(R, V), 0.0), Shininess);
    return vec4(colorDifuso + colorEspecular, 1.0);
}

subroutine(CalcularColor)
vec4 ColorLuzFocal(vec3 colorDePartida)
{
    vec3 L = normalize(posicionLuz - posicion);
        vec3 D = direccionLuz;
        float cosGamma = cos(anguloApertura * 180.0 / 3.14159265358979323846264338327795);
        float factorSpot = 1.0;

        if (dot(-L, D) < cosGamma){
            factorSpot = 0.0;
        }

        vec3 N = normalize(normal);
        vec3 V = normalize(-posicion);
        vec3 R = reflect(-L, N);

        vec3 colorDifuso = colorDePartida * Id * max(dot(N, L), 0.0);
        vec3 colorEspecular = Ks * Is * pow(max(dot(R, V), 0.0), Shininess);

        return vec4(colorDifuso + colorEspecular, 1.0) * factorSpot;
}

void main()
{
    vec3 colorDePartida = uniformEleccionColor();
    colorFragmento = ColorUniform(colorDePartida);
}
```

Clase ShaderProgram

Mencionar que en esta clase se han agragado dos funciones para buscar el identificador del uniform y de la Subrutina