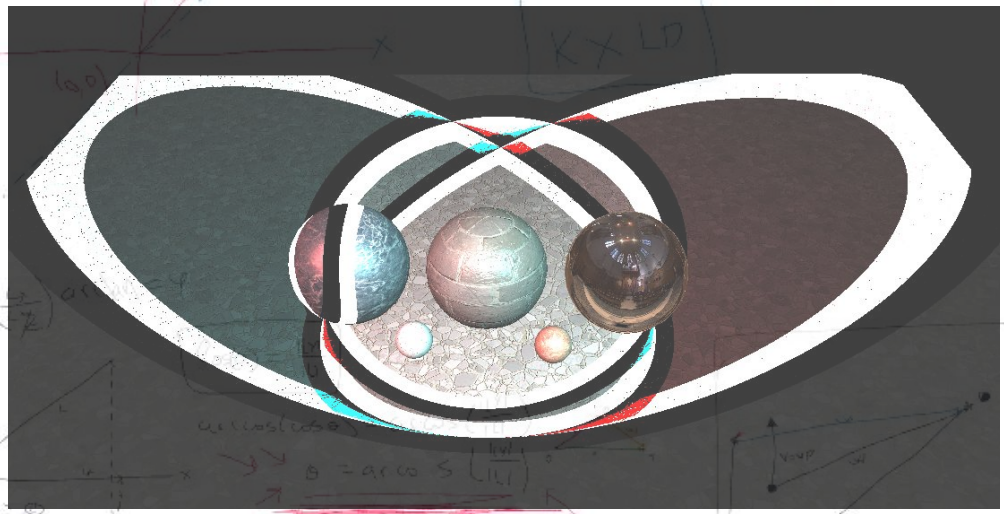
Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-4321 Computación Gráfica I
Prof. Alexandra La Cruz

Real-Time Shader Programming



Autores:
Carlos Rivero 13-11216
Carlos Sivira 15-11377
José Barrera 15-10123

Part 1:
Coordinate transform

- Para construir la matriz de rotación es necesario calcular los vectores W (dirección de la cámara al objeto), U (dirección right de la cámara, perpendicular a W y el eje Y) y V (la dirección hacia arriba de la cámara).
- Se construye la matriz de rotación como se indica en las clases de Stanford. En este caso construimos su inversa directamente.
- Se construye la matriz de transformación utilizando la función “translation” dada por Matrix4x4.
- Finalmente, se retorna el inverso de la transformación completa multiplicando la matriz de rotación por el inverso de la matriz de traslación.

src/dynamic_scene/scene.cpp

```
Matrix4x4 createWorldToCameraMatrix(const Vector3D& eye, const Vector3D& at, const Vector3D& up) {

    // TODO CS248: Camera Matrix
    // Compute the matrix that transforms a point in world space to a point in camera space.

    // Se calculan solo las direcciones puesto que no necesitamos las magnitudes.
    Vector3D W = (at - eye).unit(); // Dirección de la camara al objeto.
    Vector3D yAxis = Vector3D(0.0, 1.0, 0.0);
    Vector3D U = cross(W, yAxis).unit(); // Vector perpendicular a W y cualquier vector en la dirección de up, en este caso, Y.
    Vector3D V = up.unit(); // Y local a la camara.

    Matrix4x4 rot;
    rot[0][0] = U.x;    rot[0][1] = V.x;    rot[0][2] = -W.x;   rot[0][3] = 0.0;
    rot[1][0] = U.y;    rot[1][1] = V.y;    rot[1][2] = -W.y;   rot[1][3] = 0.0;
    rot[2][0] = U.z;    rot[2][1] = V.z;    rot[2][2] = -W.z;   rot[2][3] = 0.0;
    rot[3][0] = 0.0;    rot[3][1] = 0.0;    rot[3][2] = 0.0;    rot[3][3] = 1.0;

    //Se define la matriz de traslación
    Matrix4x4 tras = Matrix4x4::translation(eye);

    return (rot * tras.inv());
}
```

Propiedades utilizadas

- *La inversa de una matriz de traslación es la matriz de traslación con los signos opuestos en cada uno de los componentes de traslación.*
- *La inversa de una matriz de rotación es la transposición de la matriz de rotación.*
- *La inversa de un producto matricial es el producto de las matrices inversas ordenadas al revés.*

Part 2:
Implementing Phong Reflectance

- Se define la fuerza o intensidad de cada tipo de iluminación que recibe el objeto (ambiente, difusa y especular). Se ignora la ambiental porque el objeto no recibe esta iluminación.
- Se calcula el vector de reflexión especular como lo indica la clase de Stanford.
- Calculamos los colores tomando en cuenta todos los tipos de iluminación como lo indica LearnOpenGL.

media/shader.frag

```
vec3 Phong_BRDF(vec3 L, vec3 V, vec3 N, vec3 diffuse_color, vec3 specular_color, float specular_exponent)
{
    // TODO CS248: Phong Reflectance
    // Implement diffuse and specular terms of the Phong
    // reflectance model here.

    float diffuseStrength = 1.0;
    float specularStrength = 1.0;

    // difuse
    float diff = max( dot( N, L ), 0.0 );
    vec3 diffuse = diff * diffuse_color * diffuseStrength;

    // specular
    vec3 reflectDir = -L + 2.0 * dot (L, N) * N; // Lamina 50 de material, lighting and shading
    float spec = pow( max( dot( V, reflectDir ), 0.0 ), specular_exponent );
    vec3 specular = specularStrength * spec * specular_color;

    return ( diffuse + specular );
}
```

Part 3:
Normal mapping

- Para computar un punto en el espacio tangencial al espacio del mundo, como lo indica este video, es necesario calcular la matriz de rotación BTN (Binormal, Tangente y Normal).
- El vector binormal resulta del producto cruz entre la tangente y el vector normal. Esto es cierto porque por definición B es perpendicular a T y N.
- Luego, se obtiene el espacio del mundo al multiplicar esta matriz de rotación con la matriz de normales de la transformación del mundo.
- Luego, en shader.frag se asigna el color según la coordenada indicada en la textura y se convierte del espacio tangencial al espacio del mundo. Esto se consigue aplicando la transformación de la matriz de rotación BTN definida en shader.vert.
- En el mesh.cpp se realiza el binding de la variable "normalTextureSampler" con el Id de la textura normal.

media/shader.vert

```
void main(void)
{
    position = vec3(obj2world * vec4(vtx_position, 1));

    vec3 vertex_binormal = cross( vtx_tangent, vtx_normal );

    mat3 rotation_matrix = mat3(normalize(vertex_binormal), normalize(vtx_tangent), normalize(vtx_normal));
    tan2world = obj2worldNorm * rotation_matrix;

    normal = obj2worldNorm * vtx_normal;

    vertex_diffuse_color = vtx_diffuse_color;
    texcoord = vtx_texcoord;
    dir2camera = camera_position - position;
    gl_Position = mvp * vec4(vtx_position, 1);
}
```

media/shader.frag

```
uniform sampler2D diffuseTextureSampler;
uniform sampler2D normalTextureSampler;
```

```
// perform normal map lookup if required
vec3 N = vec3(0);
if (useNormalMapping) {

    vec3 normalColor = texture(normalTextureSampler, texcoord).rgb;
    vec3 tangent_space_normal = normalColor * 2.0 - 1.0;
    vec3 world_space_normal = tan2world * tangent_space_normal;
    N = normalize(world_space_normal);

} else {
    N = normalize(normal);
}
```

src/dynamic_scene/mesh.cpp

```
if (doNormalMapping_) {
    shader_->setTextureSampler("normalTextureSampler", normalTextureId_);
}
```

Part 4:
Adding environment lightning

- Para calcular la iluminación ambiental es necesario calcular los ángulos phi y theta para convertir el vector D a coordenadas esféricas.
- El ángulo Phi es calculado como la arcotangente de la proyección del Vector D en el eje x con la proyección de D en el eje z. Dado que phi existe entre [-PI, PI] entonces es necesario convertirlo de forma que se encuentre en el rango [0, 2PI].
- El ángulo theta se calcula con el arcocoseno de la proyección de D en el eje vertical Y.
- Ambos ángulos son normalizados, esto es, su valor queda entre 0 y 1.
- Con los ángulos normalizados es posible retornar el color resultante de aplicar la luz ambiental en la coordenada D.
- En el mesh.cpp se realiza el binding de la variable “environmentTextureSampler” con el Id de la textura de ambiente.

media/shader.frag

```
uniform sampler2D diffuseTextureSampler;
uniform sampler2D normalTextureSampler;
uniform sampler2D environmentTextureSampler;
```

```
vec3 SampleEnvironmentMap(vec3 D)
{
    float phi = atan(D.x, D.z); // Tan(phi) = sen(phi) / cos(phi) = ( |D.x| / |D'| ) / ( |D.z| / |D'| ).
    // D' es la proyeccion de D en el plano XZ
    float theta = acos(D.y); // D.y es la proyeccion de D en Y. El cos(theta) = | D.y |

    if (phi < 0) {
        phi = 2.0 * PI + phi;
    }

    // Normalization
    phi = phi / (2.0 * PI);
    theta = theta / PI;

    vec2 tex_coord = vec2(phi, theta);

    return texture(environmentTextureSampler, tex_coord).rgb;
}
```

src/dynamic_scene/mesh.cpp

```
// TODO CS248: Environment Mapping:
// You want to pass the environment texture into the shader program.
// See diffuseTextureSampler for an example of passing textures.

if (doEnvironmentMapping_) {
    shader_->setTextureSampler("environmentTextureSampler", environmentTextureId_);
}
```

Part 5.1:
Adding Spotlights

- Para computar la iluminación del spotlight, se calcula si un punto se encuentra dentro del cono, afuera del cono o en el área intermedia (determinada por la variable “SMOOTHING”) del cono de luz.
- Si la coordenada se encuentra en el área intermedia la intensidad se calcula mediante una interpolación lineal del ángulo como lo indica LearnOpenGL. En caso de encontrarse fuera o dentro, se anula o se aplica directamente el valor respectivamente.
- Finalmente, se aplica la atenuación calculada al color resultante.

media/shader_shadow.frag

```
// for all spot lights
for (int i = 0; i < num_spot_lights; ++i) {

    vec3 intensity = spot_light_intensities[i];   // intensity of light: this is intensity in RGB
    vec3 light_pos = spot_light_positions[i];     // location of spotlight
    float cone_angle = spot_light_angles[i];      // spotlight falls off to zero in directions whose
                                                  // angle from the light direction is grester than
                                                  // cone angle. Caution: this value is in units of degrees!

    vec3 dir_to_surface = position - light_pos;
    float angle = acos(dot(normalize(dir_to_surface), spot_light_directions[i])) * 180.0 / PI;

    float falloff = 1/(1 + pow(length(dir_to_surface), 2));

    float SMOOTHING = 0.2;

    float inside;
    if (angle > ((1.0 + SMOOTHING) * cone_angle)) {
        inside = 0.0;
    }
    else if (angle < ((1.0 - SMOOTHING) * cone_angle)) {
        inside = 1.0;
    }
    else {
        float outer = (1.0 + SMOOTHING) * cone_angle;
        float inner = (1.0 - SMOOTHING) * cone_angle;
        inside = mix(0.0, 1.0, ((outer - angle) / (outer - inner)));
    }

    vec3 L = normalize(-spot_light_directions[i]);
    vec3 brdf_color = Phong_BRDF(L, V, N, diffuseColor, specularColor, specularExponent);

    Lo += intensity * brdf_color * falloff * inside;
}
```