DOCUMENTACIÓN DE LA IMPLEMENTACIÓN:

Cube mapping

+

Environment mapping

informatika fakultatea
facultad de informática

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

Iker Pintado García

Índice

1. Introducción

La técnica de *cube mapping* consiste en dotar de un fondo o envoltura, el entorno en el que nos encontramos para dar una mayor sensación de que nos encontramos en un espacio real y no virtual. Esta técnica se puede apreciar en muchos la gran mayoría de videojuegos por las razones antes mencionadas. Para hacer esto, debemos obtener la textura del cubo que nos envolverá. Una vez obtenida, haremos que el centro del cubo se encuentre en nuestra misma posición para dar una sensación de horizonte inalcanzable.

Esta técnica se va a combinar con la de *environment mapping*. Esta consiste en hacer que algunos objetos reflejen su entorno como si de un espejo se tratase. Al tener textura única los objetos (ya que estas técnicas no se han mezclado con la de multi-textura), todos aquellos que obtengan esta propiedad, se verán como un espejo. Esto se consigue rebotando sobre la superficie del objeto un "rayo" que va desde nuestro punto de vista a un punto del objeto. El texel del lugar en donde choque el "rayo" rebotado será el texel del objeto en el punto antes mencionado.

2. Implementación

Ambas técnicas combinan muy bien ya que la mayoría de lo que refleja un objeto es el cubo de ambiente. Pero estas se implementan por separado.

2.1. Cube mapping

Para la implementación de esta técnica se han seguido los pasos especificados por el documento de egela correspondiente. Lo primero que se ha debido hacer, ha sido completar las funciones **CreateSkyBox** (crea el nodo del objeto skybox) y **DisplaySky** (dibuja dicho objeto como cielo) de **skybox.cc**:

```
void CreateSkybox(GObject *gobj,
                        ShaderProgram *skyshader,
                        const std::string &ctexname) {
      .
      .
      .
      Material *mat
=MaterialManager::instance()->create("SKY_MAT");
      mat->setTexture(ctex);
      gobj->setMaterial(mat);

      Node *newNode
=NodeManager::instance()->create("SKYBOX_NODE");
      newNode->attachShader(skyshader);
      newNode->attachGobject(gobj);
      RenderState::instance()->setSkybox(newNode);
}
```

Como se puede ver, en la primera función se crea el material del objeto que será el skybox, se le asigna una textura, y este objeto se adhiere, con el shader correcto, al nodo correspondiente.

```
void DisplaySky(Camera *cam) {
      .
      .
      .
      ShaderProgram *previous = rs->getShader();
      Vector3 pos = cam->getPosition();
      Trfm3D *matrix = new Trfm3D;
      matrix->setTrans(pos);
      skynode->setTrfm(matrix);
      glDisable(GL_DEPTH_TEST);
      rs->setShader(skynode->getShader());
      skynode->draw();
      glEnable(GL_DEPTH_TEST);
```

```
        rs->setShader(previous);
}
```

En esta función se puede apreciar como primero se guarda el shader anterior para restaurarlo después del proceso. Después, se obtiene la posición de la cámara para centrar el skybox y se desactiva el test de profundidad de opengl para dibujar el skybox correctamente. Una vez hecho esto, establecemos el shader del nodo establecido en la función anterior y dibujamos el mismo nodo. Para finalizar, restablecemos tanto el shader anterior como el test de profundidad.

Por último, solo quedaría programar los shaders para obtener el efecto deseado. En el vertex shader necesitamos pasar al espacio levógiro la posición del vértice y pasar este dato al fragment shader mediante la variable **f_texCoord**:

```
.
.
.
vec3 tmp = v_position;
tmp.z = -1.0 * tmp.z;
f_texCoord = tmp;
.
.
.
```

En el fragment shader solo se debe calcular el color del píxel:

```
void main(){
gl_FragColor = textureCube(cubemap, f_texCoord);
}
```

2.2. Environment mapping

Una vez ya implementada la técnica *cube mapping*, solo quedaría implementar la otra, *environment mapping*. Lo primero que se ha tenido que hacer, ha sido incluir la capacidad **cube_env** en **ShaderProgram::beforedraw**. El código nuevo se ha añadido justo al final de la función. Este comprueba la capacidad mencionada y, si se tiene, envía las variables correspondientes al shader:

```
if (this->has_capability("cube_env")) {
      tex = TextureManager::instance()->find("CubeEnv");
      if (tex != 0) {
            tex->bindGLUnit(Constants::gl_texunits::envmap);
            this->send_uniform("envmap",
                               Constants::gl_texunits::envmap);
            this->send_uniform("campos",
                               rs->getCamera()->getPosition());
      }
}
```

Ahora, habría que crear los shaders correspondientes. El vertex shader simplemente pasará las variables **varying** al fragment shader. En vez de pasar únicamente las variables de siempre, también pasará la posición del vértice y de su normal en el sistema de coordenadas del mundo:

```
void main() {
      //coordenadas del mundo
      f_positionw = (modelToWorldMatrix * vec4(v_position,
                                               1.0)).xyz;
      f_normalw = (modelToWorldMatrix * vec4(v_normal, 0.0)).xyz;
      //lo común
      f_position = (modelToCameraMatrix * vec4(v_position,
                                               1.0)).xyz;
      f_normal = (v_normal * vec4(v_position, 0.0)).xyz;
      f_viewDirection = -1.0 * f_position;
      f_texCoord = v_texCoord;

      gl_Position = modelToClipMatrix * vec4(v_position, 1.0);
}
```

El fragment shader es una copia de **perfragment**, pero se calcula el vector R y se pasa al sistema levógiro al final para poder obtener el texel reflejado:

```
void main(){
.
.
.
      vec3 Iw = normalize(campos - f_positionw);
      vec3 Nw = normalize(f_normalw);
      vec3 Rw = 2.0 * dot(Nw, Iw) * Nw - Iw;
```

```
        Rw.z = -1.0 * Rw.z;
        vec4 texColor;
        texColor = textureCube(envmap, Rw);
        gl_FragColor = f_color * texColor;
}
```

Por último, la profesora de la asignatura cambió/corrigió una parte de **node.cc** para que el reflejo se viese correctamente. Lo que se hace es añadir, en la función **draw**, que la transformación se fije en la pila del modelo:

```
rs->loadTrfm(RenderState::model, m_placementWC);
```

3. Resultado en ejecución

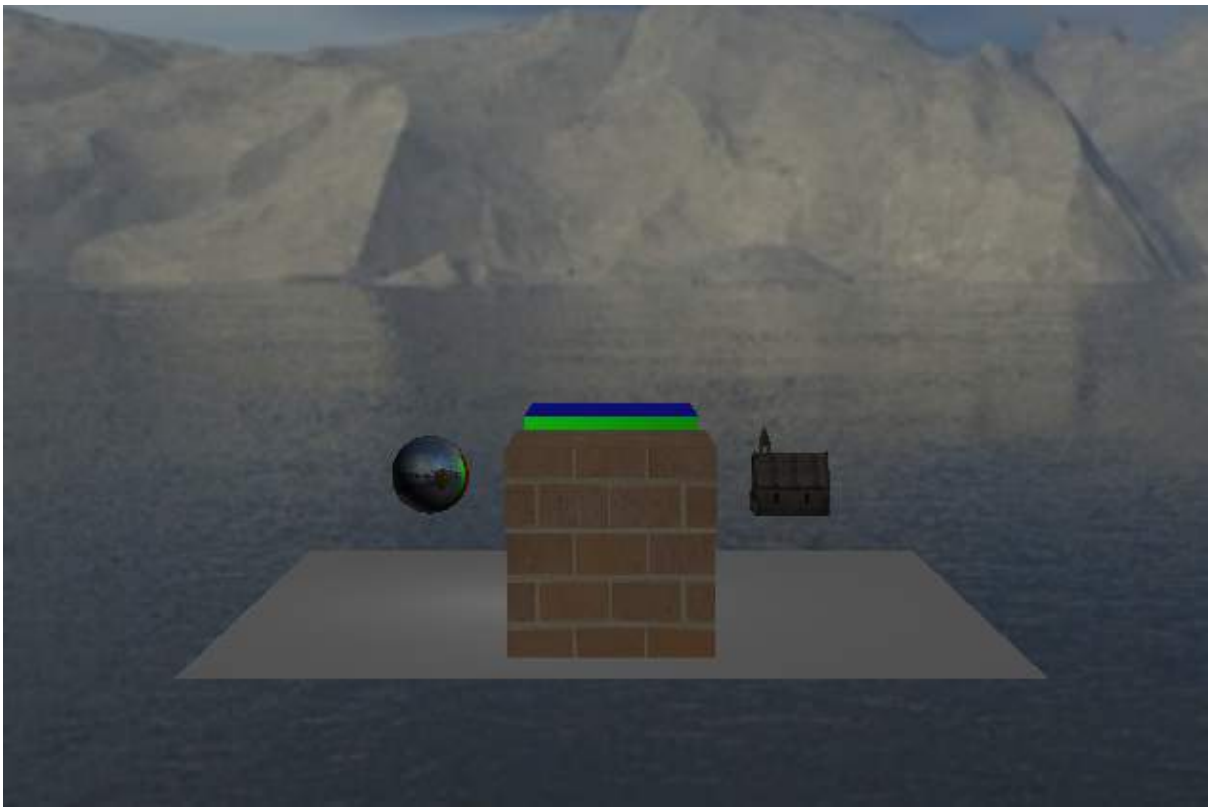

Fig. 1. Visualización de la escena

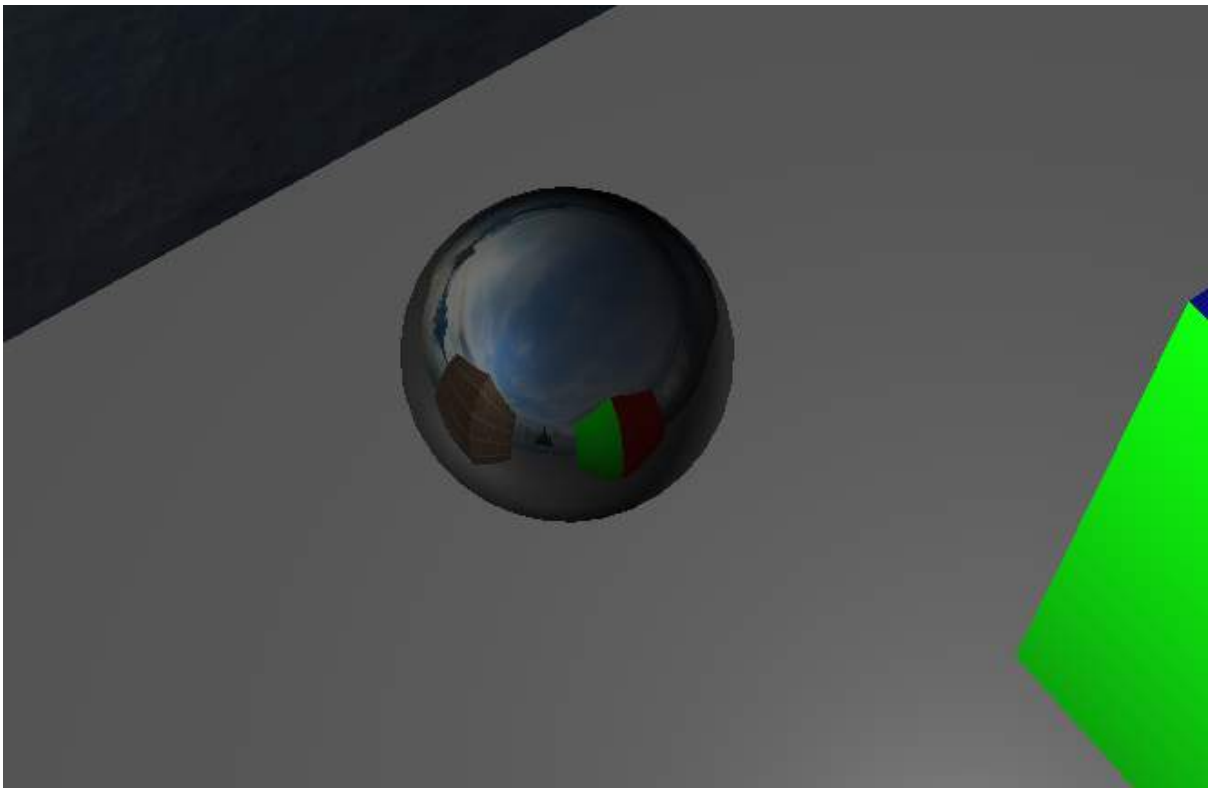

Fig. 2. Visualización de la esfera con mapeado ambiental

4. Fuentes externas

- The book of shaders
- LearnOpenGL