Universidad Rey Juan Carlos

ETSII

Práctica 3: Introducción a la programación en OpenGL

Informática Gráfica
Grado en Diseño y Desarrollo de Videojuegos

Miguel Ángel Otaduy
Marcos García Lorenzo

Tabla de contenido

Introducción 3
Normativa 3
Parte guiada 3
Paso 1: creación del contexto de OpenGL 3
Paso 2: configuración del cauce 4
Paso 3: compilación de *shaders* 5
Paso 3: carga del modelo 7
Paso 3: Renderizado del modelo 9
Paso 4: Ajuste del viewport 9
Paso 5: Animación 9
Paso 6: Texturas 10
Parte obligatoria 12
Parte opcional 12
Apéndice A: Anisotropic Filtering 13

Introducción

El objetivo de esta práctica es que el alumno programe la etapa de aplicación utilizando el API de OpenGL 3.3. Cada grupo de prácticas deberá implementar la funcionalidad detallada en este guion de prácticas.

Normativa

Esta práctica se divide en tres bloques:

- Bloque guiado: este bloque se realizará de forma guiada en el aula de prácticas. La **asistencia es obligatoria**. El bloque guiado no es re-evaluable por lo que la no asistencia acarreará el suspenso de la asignatura.
- Bloque obligatorio: los grupos de prácticas deberán realizar este bloque de forma supervisada (pero no guiada) en las horas de laboratorio que se habiliten para tal efecto. El trabajo que no pueda realizarse dentro de las horas de laboratorio podrá realizarse fuera del horario de la asignatura. El trabajo realizado, tanto en el bloque guiado como en el bloque obligatorio, se evaluará mediante un examen de prácticas que podrá aprobarse tanto en la convocatoria de diciembre como en la de junio. El código de ambos bloques se entregará junto, a través de campus virtual en el plazo que indique en la página web de la asignatura. Deberá adjuntarse una pequeña memoria explicativa.
- Bloque opcional: las partes opcionales se realizarán por el alumno fuera del horario de la asignatura. Las tareas de este bloque no son obligatorias pero la nota final dependerá en gran medida de las partes opcionales realizadas. El bloque opcional se entregará junto con una memoria explicativa a través de la herramienta habilitada en campus virtual, en el plazo que se indique.

Las prácticas podrán realizarse tanto de forma individual como en grupos de dos personas.

Parte guiada

Paso 1: creación del contexto de OpenGL

En esta primera parte de la práctica se utilizará la librería auxiliar *GLUT* para inicializar el contexto de OpenGL, crear el *Frame Buffer*, crear la ventana de renderizado y definir las funciones encargadas de tratar los eventos enviados por el sistema operativo. Además, se utilizará la librería GLEW para inicializar las extensiones.

1. El profesor explicará el entorno compuesto de:
    a. Un proyecto de Visual Studio en el que se deberá implementar la funcionalidad del cliente.
    b. El modelo 3D de un cubo.
    c. Una carpeta con los *shaders* que se utilizarán en esta práctica.

2. En la función: `void initContext(int argc, char** argv)`, crea el contexto utilizando la librería *GLUT:*

```
glutInit(&argc, argv);
glutInitContextVersion(3, 3);
glutInitContextFlags(GLUT_FORWARD_COMPATIBLE);
glutInitContextProfile(GLUT_CORE_PROFILE);
//glutInitContextProfile(GLUT_COMPATIBILITY_PROFILE);
```

3. Define el *Frame Buffer* y crea la ventana:

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
glutInitWindowSize(500, 500);
glutInitWindowPosition(0, 0);
glutCreateWindow("Prácticas GLSL");
```

4. Inicializa las extensiones:

```
glewExperimental = GL_TRUE;
GLenum err = glewInit();
if (GLEW_OK != err)
{
        std::cout << "Error: " << glewGetErrorString(err) << std::endl;
        exit (-1);
}

const GLubyte *oglVersion = glGetString(GL_VERSION);
std::cout << "This system supports OpenGL Version: " << oglVersion << std::endl;
```

5. Indica que funciones tratarán los distintos eventos:

```
glutReshapeFunc(resizeFunc);
glutDisplayFunc(renderFunc);
glutIdleFunc(idleFunc);
glutKeyboardFunc(keyboardFunc);
glutMouseFunc(mouseFunc);
```

6. En la función `int main(int argc, char** argv)`, añade el bucle de eventos:

```
glutMainLoop();
```

7. En la función `void renderFunc()`, añade el siguiente código:

```
glutSwapBuffers();
```

Paso 2: configuración del cauce

1. En la función `void initOGL()`, activa el test de profundidad y establece el color de fondo:

```
glEnable(GL_DEPTH_TEST);
glClearColor(0.2f, 0.2f, 0.2f, 0.0f);
```

2. Indica la orientación de la cara *front*, configura la etapa de *rasterizado* y activa el *culling*:

```
glFrontFace(GL_CCW);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glEnable(GL_CULL_FACE);
```

3. Define la matriz de vista y la matriz de proyección:

```
proj = glm::perspective(glm::radians(60.0f), 1.0f, 0.1f, 50.0f);
view = glm::mat4(1.0f);
view[3].z = -6;
```

4. En la función `void renderFunc()`, limpia el buffer de color y el buffer de profundidad antes de cada renderizado.

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Paso 3: compilación de *shaders*

1. Crea tres variables globales que nos permitan manejar el *shader* del vértices, el *shader* de fragmentos y el programa que los enlaza:

```
unsigned int vshader;
unsigned int fshader;
unsigned int program;
```

2. Estudia los *shaders* de vértices y fragmentos. Crea variables globales que nos permitan acceder a las variables uniformes y a los atributos.

```
//Variables Uniform
int uModelViewMat;
int uModelViewProjMat;
int uNormalMat;

//Atributos
int inPos;
int inColor;
int inNormal;
int inTexCoord;
```

3. En la función `GLuint loadShader(const char *fileName, GLenum type)`, añade el código de carga y compilación de un *shader* genérico:

```
unsigned int fileLen;
char *source = loadStringFromFile(fileName, fileLen);

///////////////////////////////////////////
//Creación y compilación del Shader
GLuint shader;
shader = glCreateShader(type);
glShaderSource(shader, 1,
      (const GLchar **)&source, (const GLint *)&fileLen);
glCompileShader(shader);
delete source;
```

4. Añade el código de comprobación de errores:

```
//Comprobamos que se compiló bien
GLint compiled;
glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
if (!compiled)
{
	//Calculamos una cadena de error
	GLint logLen;
	glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &logLen);

	char *logString = new char[logLen];
	glGetShaderInfoLog(shader, logLen, NULL, logString);
	std::cout << "Error: " << logString << std::endl;
	delete logString;

	glDeleteShader(shader);
	exit (-1);
}
```

5. Devuelve el ID del *shader* compilado:

```
return shader;
```

6. En la función `void initShader(const char *vname, const char *fname)`, crea un *shader* de vértices y uno de fragmentos:

```
vshader = loadShader(vname, GL_VERTEX_SHADER);
fshader = loadShader(fname, GL_FRAGMENT_SHADER);
```

7. Enlaza los dos *shaders* en un programa:

```
program = glCreateProgram();
glAttachShader(program, vshader);
glAttachShader(program, fshader);

glLinkProgram(program);
```

8. Comprueba los errores de la fase de enlazado:

```
int linked;
glGetProgramiv(program, GL_LINK_STATUS, &linked);
if (!linked)
{
	//Calculamos una cadena de error
	GLint logLen;
	glGetProgramiv(program, GL_INFO_LOG_LENGTH, &logLen);

	char *logString = new char[logLen];
	glGetProgramInfoLog(program, logLen, NULL, logString);
	std::cout << "Error: " << logString << std::endl;
	delete logString;

	glDeleteProgram(program);
	program = 0;
	exit (-1);
}
```

9. Asigna un identificador a los atributos del programa (antes de enlazar el programa):

```
glBindAttribLocation(program, 0, "inPos");
glBindAttribLocation(program, 1, "inColor");
glBindAttribLocation(program, 2, "inNormal");
glBindAttribLocation(program, 3, "inTexCoord");
```

10. Crea los identificadores de las variables uniformes:

```
uNormalMat = glGetUniformLocation(program, "normal");
uModelViewMat = glGetUniformLocation(program, "modelView");
uModelViewProjMat = glGetUniformLocation(program, "modelViewProj");
```

11. Crea los identificadores de los atributos:

```
inPos = glGetAttribLocation(program, "inPos");
inColor = glGetAttribLocation(program, "inColor");
inNormal = glGetAttribLocation(program, "inNormal");
inTexCoord = glGetAttribLocation(program, "inTexCoord");
```

12. En la función `void renderFunc()`, activa y desactiva el programa:

```
glUseProgram(program);
//→ pintado del  objeto!!!!
glUseProgram(NULL);
```

13. En la función `void destroy()`, libera los recursos utilizados:

```
glDetachShader(program, vshader);
glDetachShader(program, fshader);
glDeleteShader(vshader);
glDeleteShader(fshader);
glDeleteProgram(program);
```

Paso 3: carga del modelo

1. Crea las variables globales que nos permitirán configurar el objeto:

```
//VAO
unsigned int vao;

//VBOs que forman parte del objeto
unsigned int posVBO;
unsigned int colorVBO;
unsigned int normalVBO;
unsigned int texCoordVBO;
unsigned int triangleIndexVBO;
```

2. En la función `void initObj()`, crea y activa el VAO en el que se almacenará la configuración del objeto:

```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

3. Crea y configura todos los atributos de la malla:

```
if (inPos != -1)
{
	glGenBuffers(1, &posVBO);
	glBindBuffer(GL_ARRAY_BUFFER, posVBO);
	glBufferData(GL_ARRAY_BUFFER, cubeNVertex*sizeof(float) * 3,
		cubeVertexPos, GL_STATIC_DRAW);
	glVertexAttribPointer(inPos, 3, GL_FLOAT, GL_FALSE, 0, 0);
	glEnableVertexAttribArray(inPos);
}

if (inColor != -1)
{
	glGenBuffers(1, &colorVBO);
	glBindBuffer(GL_ARRAY_BUFFER, colorVBO);
	glBufferData(GL_ARRAY_BUFFER, cubeNVertex*sizeof(float) * 3,
		cubeVertexColor, GL_STATIC_DRAW);
	glVertexAttribPointer(inColor, 3, GL_FLOAT, GL_FALSE, 0, 0);
	glEnableVertexAttribArray(inColor);
}

if (inNormal != -1)
{
	glGenBuffers(1, &normalVBO);
	glBindBuffer(GL_ARRAY_BUFFER, normalVBO);
	glBufferData(GL_ARRAY_BUFFER, cubeNVertex*sizeof(float) * 3,
		cubeVertexNormal, GL_STATIC_DRAW);
	glVertexAttribPointer(inNormal, 3, GL_FLOAT, GL_FALSE, 0, 0);
	glEnableVertexAttribArray(inNormal);
}


if (inTexCoord != -1)
{
	glGenBuffers(1, &texCoordVBO);
	glBindBuffer(GL_ARRAY_BUFFER, texCoordVBO);
	glBufferData(GL_ARRAY_BUFFER, cubeNVertex*sizeof(float) * 2,
		cubeVertexTexCoord, GL_STATIC_DRAW);
	glVertexAttribPointer(inTexCoord, 2, GL_FLOAT, GL_FALSE, 0, 0);
	glEnableVertexAttribArray(inTexCoord);
}
```

4. Crea la lista de índices:

```
glGenBuffers(1, &triangleIndexVBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triangleIndexVBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
	cubeNTriangleIndex*sizeof(unsigned int) * 3, cubeTriangleIndex,
	GL_STATIC_DRAW);
```

5. Inicializa la matriz *model* de este objeto:

```
model = glm::mat4(1.0f);
```

6. En la función `void destroy()`, libera los recursos utilizados:

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
if (inPos != -1) glDeleteBuffers(1, &posVBO);
if (inColor != -1) glDeleteBuffers(1, &colorVBO);
if (inNormal != -1) glDeleteBuffers(1, &normalVBO);
if (inTexCoord != -1) glDeleteBuffers(1, &texCoordVBO);
glDeleteBuffers(1, &triangleIndexVBO);

glBindVertexArray(0);
glDeleteVertexArrays(1, &vao);
```

Paso 3: Renderizado del modelo

1. En la función `void renderFunc()`, calcula y sube las matrices requeridas por el *shader* de vértices:

```
glm::mat4 modelView = view * model;
glm::mat4 modelViewProj = proj * view * model;
glm::mat4 normal = glm::transpose(glm::inverse(modelView));

if (uModelViewMat != -1)
	glUniformMatrix4fv(uModelViewMat, 1, GL_FALSE,
		&(modelView[0][0]));
if (uModelViewProjMat != -1)
	glUniformMatrix4fv(uModelViewProjMat, 1, GL_FALSE,
		&(modelViewProj[0][0]));
if (uNormalMat != -1)
	glUniformMatrix4fv(uNormalMat, 1, GL_FALSE,
		&(normal[0][0]));
```

2. Activa el VAO con la configuración del objeto y pinta la lista de triángulos:

```
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, cubeNTriangleIndex*3,
	GL_UNSIGNED_INT, (void*)0);
```

Paso 4: Ajuste del viewport

1. Ajusta el *viewport* en la función `void resizeFunc(int width, int height)`:

```
glViewport(0, 0, width, height);
```

2. Planifica un evento de renderizado:

```
glutPostRedisplay();
```

Paso 5: Animación

1. Modifica la matriz *model* en la función `void idleFunc()`:

```
model = glm::mat4(1.0f);
static float angle = 0.0f;
angle = (angle > 3.141592f * 2.0f) ? 0 : angle + 0.01f;

model = glm::rotate(model, angle, glm::vec3(1.0f, 1.0f, 0.0f));
```

2. Lanza un evento de renderizado:

```
glutPostRedisplay();
```

Paso 6: Texturas

1. Los *shaders* utilizados hasta el momento no soportan texturas, cámbialos por *shader.v1.vert* y *shader.v1.frag.*
2. Crea dos variables que te permitan configurar 2 texturas:

```
//Texturas
unsigned int colorTexId;
unsigned int emiTexId;
```

3. Estudia los *shaders* de vértices y fragmentos. Crea variables globales que nos permitan acceder a las variables uniformes que aún no hemos utilizado (recuerda que ya hemos definido el atributo *inTexCoord* que utilizaremos en este paso).

```
//Texturas Uniform
int uColorTex;
int uEmiTex;
```

4. Utilizaremos la función `unsigned int loadTex(const char *fileName)`, para cargar y configurar una textura genérica. Carga la textura almacenada en el fichero indicado:

```
unsigned char *map;
unsigned int w, h;
map = loadTexture(fileName, w, h);

if (!map)
{
	std::cout << "Error cargando el fichero: "
	<< fileName << std::endl;
	exit(-1);
}
```

5. Crea una textura, actívala y súbela a la tarjeta gráfica:

```
unsigned int texId;
glGenTextures(1, &texId);
glBindTexture(GL_TEXTURE_2D, texId);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, w, h, 0, GL_RGBA,
	GL_UNSIGNED_BYTE, (GLvoid*)map);
```

6. Libera la memoria de la CPU:

```
delete[] map;
```

7. Crea los *mipmaps* asociados a la textura:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

8. Configura el modo de acceso:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
       GL_LINEAR_MIPMAP_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
```

9. Devuelve el identificador de la textura:

```
return texId;
```

10. En la función `void initObj()`, crea dos texturas (una con el color y otra con un mapa emisivo):

```
colorTexId = loadTex("../img/color2.png");
emiTexId = loadTex("../img/emissive.png");
```

11. En la función `void initShader(const char *vname, const char *fname)`, crea los identificadores de las variables uniformes:

```
uColorTex = glGetUniformLocation(program, "colorTex");
uEmiTex = glGetUniformLocation(program, "emiTex");
```

12. Liberar recursos en la función `void destroy()`

```
glDeleteTextures(1, &colorTexId);
glDeleteTextures(1, &emiTexId);
```

13. En la función `void renderFunc()`, activa las texturas y enlázalas con el programa activo:

```
//Texturas
if (uColorTex != -1)
{
       glActiveTexture(GL_TEXTURE0);
       glBindTexture(GL_TEXTURE_2D, colorTexId);
       glUniform1i(uColorTex, 0);
}

if (uEmiTex != -1)
{
       glActiveTexture(GL_TEXTURE0 + 1);
       glBindTexture(GL_TEXTURE_2D, emiTexId);
       glUniform1i(uEmiTex, 1);
}
```

Parte obligatoria

En este bloque deberá implementarse la siguiente funcionalidad:

1. **Modifica las propiedades (intensidad y posición) de la luz a través del teclado.**
2. **Define una matriz de proyección que conserve el *aspect ratio*** cuando cambiamos el tamaño de la ventana.
3. **Añade un nuevo cubo a la escena**. El segundo cubo deberá orbitar alrededor del primero describiendo una circunferencia a la vez que rota sobre su eje Y. **NOTA: No tienes por que crear otro objeto. Pinta el mismo VAO con otra matriz *model.***
4. **Control de la cámara con el teclado**. Controles mínimos que deberán incluirse: movimiento hacia adelante, retroceso, movimientos laterales (izquierda y derecha) y giros (izquierda y derecha). **Importante: la posición de la luz debe ser invariante con respecto a la posición de la cámara.**

Parte opcional

Este apartado describe las partes que podrán realizarse de forma opcional.

1. Mejora el comportamiento de las texturas utilizando un filtro anisotrópico (tu tarjeta gráfica debe de soportarlo - *EXT_texture_filter_anisotropic*):
    a. Ver apéndice A.
2. Implementa la funcionalidad de las prácticas 1 y 2:
    a. Controla el giro de la cámara utilizando el ratón.
    b. Crea un tercer cubo y hazlo orbitar alrededor del primero. Define su movimiento utilizando curvas de *Bézier*, *splines* cúbicos o polinomios de interpolación de *Catmull-Rom.*
    c. Crea un nuevo modelo y añádelo a la escena.
        i. Calcula las normales y/o las tangentes de dicho modelo.
    d. Ilumina el objeto con luces de distinto tipo.
    e. Añade atenuación con la distancia.
    f. Implementa *Bump Mapping.*
    g. Añade una textura especular.
3. Puedes sugerir nuevas partes opcionales a los profesores de prácticas.

Apéndice A: Anisotropic Filtering

Información sacada de "*OpenGL SuperBible: Comprehensive Tutorial and Reference, 4th Edition*" capítulo "9: Texture Mapping in OpenGL: Beyond the Basics". El capítulo entero está disponible en Intenet.

Anisotropic Filtering

Anisotropic texture filtering is not a part of the core OpenGL specification, but it is a widely supported extension that can dramatically improve the quality of texture filtering operations. Texture filtering is covered in the preceding chapter, where you learned about the two basic texture filters: nearest neighbor (`GL_NEAREST`) and linear (`GL_LINEAR`). When a texture map is filtered, OpenGL uses the texture coordinates to figure out where in the texture map a particular fragment of geometry falls. The texels immediately around that position are then sampled using either the `GL_NEAREST` or the `GL_LINEAR` filtering operations.

This process works perfectly when the geometry being textured is viewed directly perpendicular to the viewpoint, as shown on the left in Figure 9.4. However, when the geometry is viewed from an angle more oblique to the point of view, a regular sampling of the surrounding texels results in the loss of some information in the texture (it looks blurry!). A more realistic and accurate sample would be elongated along the direction of the plane containing the texture. This result is shown on the right in Figure 9.4. Taking this viewing angle into account for texture filtering is called *anisotropic filtering.*

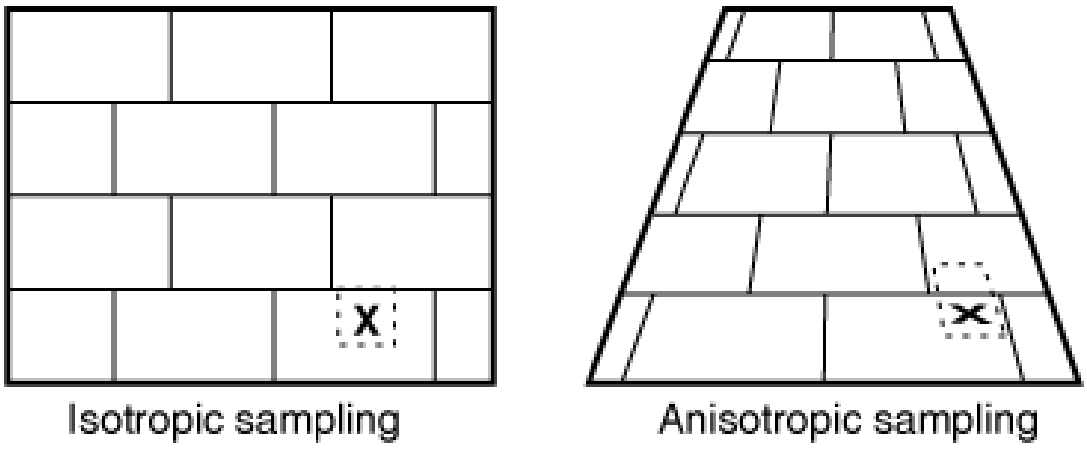

Figure 9.4 Normal texture sampling versus anisotropic sampling.

You can apply anisotropic filtering to any of the basic or mipmapped texture filtering modes; applying it requires three steps. First, you must determine whether the extension is supported. You can do this

by querying for the extension string `GL EXT texture filter anisotropic`. You can use the `glTools`[1] function named `gltIsExtSupported` for this task:

```
if(gltIsExtSupported("GL_EXT_texture_filter_anisotropic"))

    // Set Flag that extension is supported
```

After you determine that this extension is supported, you can find the maximum amount of *anisotropy*supported. You can query for it using `glGetFloatv` and the parameter`GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT`:

```
GLfloat fLargest;

. . .

. . .

glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &fLargest);
```

The larger the amount of anisotropy applied, the more texels are sampled along the direction of greatest change (along the strongest point of view). A value of `1.0` represents normal texture filtering (called *isotropic* filtering). Bear in mind that anisotropic filtering is not free. The extra amount of work, including other texels, can sometimes result in substantial performance penalties. On modern hardware, this feature is getting quite fast and is becoming a standard feature of popular games, animation, and simulation programs.
Finally, you set the amount of anisotropy you want applied using `glTexParameter` and the constant`GL_TEXTURE_MAX_ANISOTROPY_EXT`. For example, using the preceding code, if you want the maximum amount of anisotropy applied, you would call `glTexParameterf` as shown here:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, fLargest);
```

[1] En lugar de esta librería puede usar glew: `glewIsSupported`

This modifier is applied per texture object just like the standard filtering parameters.

The sample program ANISOTROPIC provides a striking example of anisotropic texture filtering in action. This program displays a tunnel with walls, a floor, and ceiling geometry. The arrow keys move your point of view (or the tunnel) back and forth along the tunnel interior. A right mouse click brings up a menu that allows you to select from the various texture filters, and turn on and off anisotropic filtering. Figure 9.5 shows the tunnel using trilinear filtered mipmapping. Notice how blurred the patterns become in the distance, particularly with the bricks.

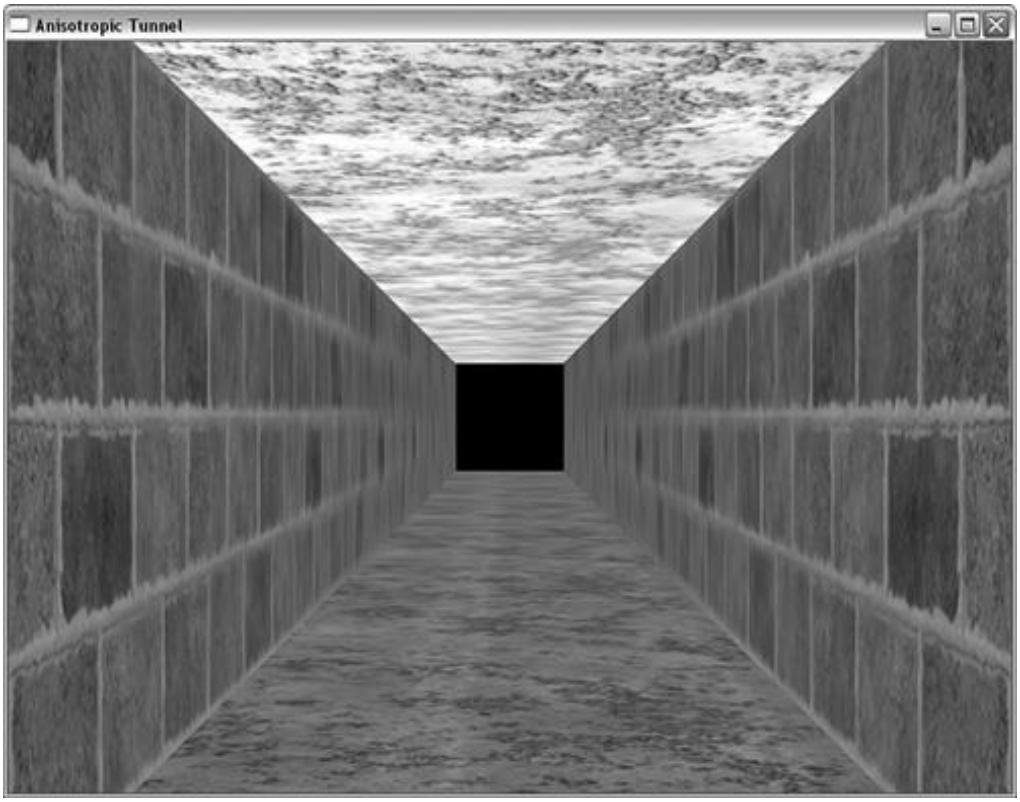

Figure 9.5 ANISOTROPIC tunnel sample with trilinear filtering.

Now compare Figure 9.5 with Figure 9.6, in which anisotropic filtering has been enabled. The mortar between the bricks is now clearly visible all the way to the end of the tunnel. In fact, anisotropic filtering can also greatly reduce the visible mipmap transition patterns for the `GL_LINEAR_MIPMAP_NEAREST` and `GL_NEAREST_MIPMAP_NEAREST` mipmapped filters.

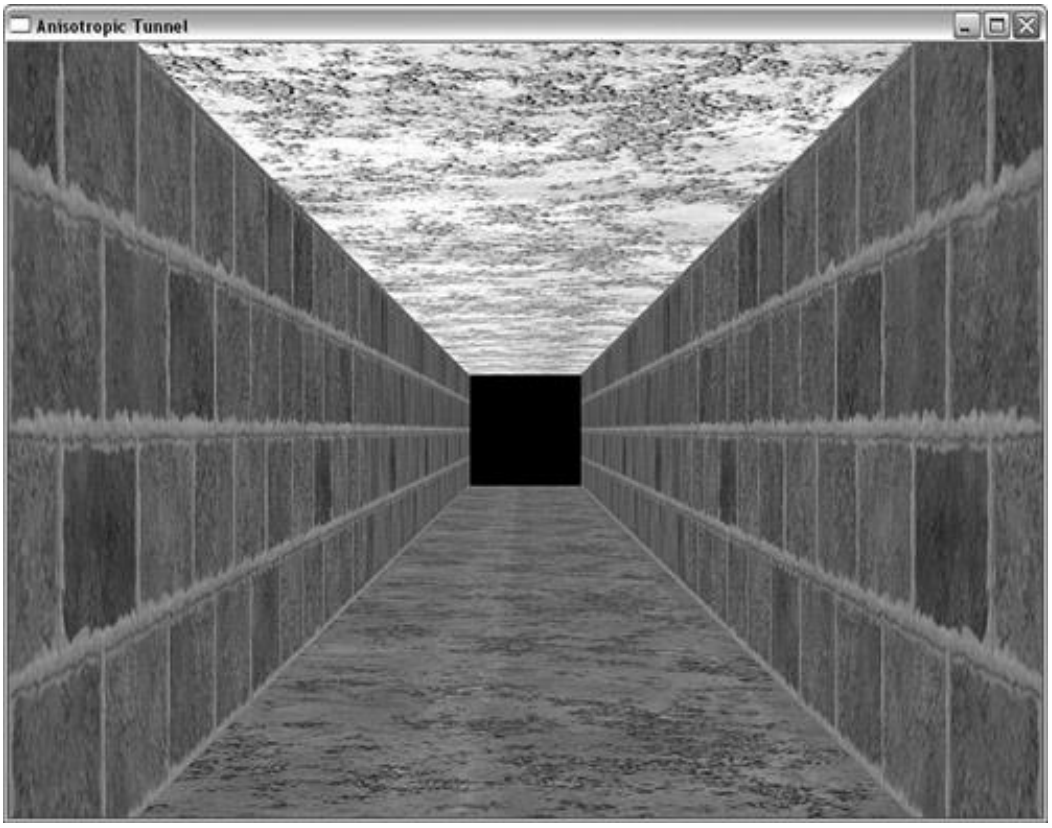

Figure 9.6 ANISOTROPIC tunnel sample with anisotropic filtering.