Universidad Rey Juan Carlos

ETSII

Práctica 4: Aumento de realismo – Post-Proceso (MRT)

Informática Gráfica
Grado en Diseño y Desarrollo de Videojuegos

Marcos García Lorenzo

Tabla de contenido

Introducción

El objetivo de esta práctica es que el alumno entienda que los métodos de iluminación local no pueden simular muchos de los comportamientos típicos de la luz. La posibilidad de renderizar sobre una textura que se pueda utilizar posteriormente en la escena nos permite diseñar técnicas que suplan de forma más o menos efectiva alguna de estas carencias.

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: entorno de prácticas

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. El modelo 3D de un plano.
    d. Una carpeta con los *shaders* que se utilizarán en esta práctica.

2. Compila y ejecuta el proyecto tal y como está.

Paso 2: renderiza un cuadrado sobre el plano de proyección

En este apartado se subirá la geometría y se diseñarán los *shaders* encargados de pintar un cuadrado que ocupe todos los píxeles de la pantalla. Comenzaremos diseñando el *shader* de vértices y el de fragmentos (postProcessing.v0.vert y postProcessing.v0.frag).

1. Crea un *shader* de vértices que pase las coordenadas de textura de los vértices del cuadrado al *shader* de fragmentos:

```
//Variables Variantes
out vec2 texCoord;

//Código del Shader
texCoord = inPos.xy*0.5+vec2(0.5);
gl_Position =  vec4 (inPos,1.0);
```

2. Crea un *shader* de fragmentos, compatible con el *shader* anterior. Asigna las coordenadas de textura al color de salida. Añade la textura que se pegará sobre el plano:

```
//Color de salida
out vec4 outColor;

//Variables Variantes
in vec2 texCoord;

//Textura
uniform sampler2D colorTex;

//Código del Shader
//NOTA: Sería más adecuado usar texelFetch.
//NOTA: No lo hacemos porque simplifica el paso 5
//outColor = vec4(textureLod(colorTex, texCoord,0).xyz, 0.6);
outColor = vec4(texCoord,vec2(1.0));
```

A continuación compilaremos y enlazaremos los *shaders* anteriores.

3. Define las variables que nos darán acceso a los *shaders* anteriores:

```
unsigned int postProccesVShader;
unsigned int postProccesFShader;
unsigned int postProccesProgram;

//Uniform
unsigned int uColorTexPP;

//Atributos
int inPosPP;
```

4. Crea la función `void initShaderPP(const char *vname, const char *fname)` (declaración y definición):

```
postProccesVShader = loadShader(vname, GL_VERTEX_SHADER);
postProccesFShader = loadShader(fname, GL_FRAGMENT_SHADER);
```

```
postProccesProgram = glCreateProgram();
glAttachShader(postProccesProgram, postProccesVShader);
glAttachShader(postProccesProgram, postProccesFShader);

glBindAttribLocation(postProccesProgram, 0, "inPos");

glLinkProgram(postProccesProgram);

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

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

	glDeleteProgram(postProccesProgram);
	postProccesProgram = 0;
	exit(-1);
}

uColorTexPP = glGetUniformLocation(postProccesProgram, "colorTex");
inPosPP = glGetAttribLocation(postProccesProgram, "inPos");
```

5. Añade el código que libera los recursos utilizados al salir de la aplicación (`void destroy()`):

```
glDetachShader(postProccesProgram, postProccesVShader);
glDetachShader(postProccesProgram, postProccesFShader);
glDeleteShader(postProccesVShader);
glDeleteShader(postProccesFShader);
glDeleteProgram(postProccesProgram);
```

6. Carga los *shaders* de post-proceso en la función `int main(int argc, char** argv)`:

```
initShaderPP( "../shaders_P4/postProcessing.v0.vert",
              "../shaders_P4/postProcessing.v0.frag");
```

Carga la geometría del cuadrado y píntala en la función de render.

7. Define las variables que nos darán acceso al objeto cuadrado:

```
unsigned int planeVAO;
unsigned int planeVertexVBO;
```

8. Crea la función `void initPlane()` (declaración y definición):

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

glGenBuffers(1, &planeVertexVBO);
glBindBuffer(GL_ARRAY_BUFFER, planeVertexVBO);
glBufferData(GL_ARRAY_BUFFER, planeNVertex*sizeof(float) * 3,
             planeVertexPos, GL_STATIC_DRAW);
glVertexAttribPointer(inPosPP, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

```
glEnableVertexAttribArray(inPosPP);
```

9. Añade el código que libera los recursos utilizados al salir de la aplicación (`void destroy()`):

```
glDeleteBuffers(1, &planeVertexVBO);
glDeleteVertexArrays(1, &planeVAO);
```

10. Inicializa la geometría en la función `int main(int argc, char** argv)`:

```
initPlane();
```

11. Comenta el código que *renderiza* los cubos en la función `void renderFunc()`.
12. Añade a la función `void renderFunc()` el código encargado de pintar el plano:

```
glUseProgram(postProccesProgram);
glDisable(GL_CULL_FACE);
glDisable(GL_DEPTH_TEST);

glBindVertexArray(planeVAO);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

glEnable(GL_CULL_FACE);
glEnable(GL_DEPTH_TEST);
```

Ahora crearemos un FBO para poder *renderizar* la primera escena directamente sobre una textura. Posteriormente asociaremos esa textura al *shader* encargado de *renderizar* el plano.

13. Define las variables que nos darán acceso al objeto cuadrado:

```
unsigned int fbo;
unsigned int colorBuffTexId;
unsigned int depthBuffTexId;
```

14. Crea la función `void initFBO()` (declaración y definición):

```
glGenFramebuffers(1, &fbo);
glGenTextures(1, &colorBuffTexId);
glGenTextures(1, &depthBuffTexId);
```

15. Crea la función `void resizeFBO(unsigned int w, unsigned int h)` (declaración y definición):

```
glBindTexture(GL_TEXTURE_2D, colorBuffTexId);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, w, h, 0,
       GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);

glBindTexture(GL_TEXTURE_2D, depthBuffTexId);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT24, w, h, 0,
       GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
	GL_TEXTURE_2D, colorBuffTexId, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
	depthBuffTexId, 0);

const GLenum buffs[1] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, buffs);

if (GL_FRAMEBUFFER_COMPLETE != glCheckFramebufferStatus(GL_FRAMEBUFFER))
{
	std::cerr << "Error configurando el FBO" << std::endl;
	exit(-1);
}
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

16. Modifica "*fwRendering.v0.frag*" para que el color se envíe al **buffer 0**:

```
layout(location = 0) out vec4 outColor;
```

17. Añade el código que libera los recursos utilizados al salir de la aplicación (`void destroy()`):

```
glDeleteFramebuffers(1, &fbo);
glDeleteTextures(1, &colorBuffTexId);
glDeleteTextures(1, &depthBuffTexId);
```

18. Inicializa el FBO en la función `int main(int argc, char** argv)`:

```
initFBO();
resizeFBO(SCREEN_SIZE);
```

19. Cambia el tamaño del FBO cada vez que se modifique el tamaño de la pantalla:

```
resizeFBO(width, height);
```

20. Descomenta el código que *renderiza* los cubos en la función `void renderFunc()`.
21. Activa el FBO antes del *renderizado* de los cubos y desactívalo justo después (`void renderFunc()`):

```
Situar antes de limpiar el FB
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
…
Situar después de renderizar los cubos
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

22. Sube la textura al *shader* de post-proceso (`void renderFunc()`):

```
if (uColorTexPP != -1)
{
	glActiveTexture(GL_TEXTURE0);
	glBindTexture(GL_TEXTURE_2D, colorBuffTexId);
	glUniform1i(uColorTexPP, 0);
}
```

23. **Cambia el *shader* de post-proceso para que calcule el color del fragmento en función de la textura.**

Paso 3: *Motion Blur*

1. Utiliza las funciones de *blending* para fusionar el *frame* actual con el anterior (`void renderFunc()` – identifica posibles problemas con el doble buffer):

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glBlendEquation(GL_FUNC_ADD);

…
Post-proceso
…

glDisable(GL_BLEND);
```

2. Cambia la función de *blending* para mejorar el control sobre el algoritmo anterior (`void renderFunc()`):

```
//glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glBlendFunc(GL_CONSTANT_COLOR, GL_CONSTANT_ALPHA);
glBlendColor(0.5f, 0.5f, 0.5f, 0.6f);
glBlendEquation(GL_FUNC_ADD);
```

Paso 4: *Gaussian Blur*

1. Guarda los ficheros fwRendering.*v0.vert,* fwRendering.*v0.frag, postProcessing.v0.vert y postProcessing.v1.frag, main.cpp* como fwRendering.*v1.vert,* fwRendering.*v1.frag, postProcessing.v1.vert, postProcessing.v1.frag y main.v0.cpp.*
2. Modifica la función `int main(int argc, char** argv)` para que utilicen los nuevos *shaders* de vértices y fragmentos.
3. Desactiva el *blending* en la función `void renderFunc()`.
4. Añade una máscara de convolución[1] 3x3 en "*postProcessing.v1.frag*":

```
#define MASK_SIZE 9u
const float maskFactor = float (1.0/14.0);
const vec2 texIdx[MASK_SIZE] = vec2[](
	vec2(-1.0,1.0),  vec2(0.0,1.0),  vec2(1.0,1.0),
	vec2(-1.0,0.0),  vec2(0.0,0.0),  vec2(1.0,0.0),
	vec2(-1.0,-1.0), vec2(0.0,-1.0), vec2(1.0,-1.0));

const float mask[MASK_SIZE] = float[](
	float (1.0*maskFactor), float (2.0*maskFactor), float (1.0*maskFactor),
	float (2.0*maskFactor), float (2.0*maskFactor), float (2.0*maskFactor),
	float (1.0*maskFactor), float (2.0*maskFactor), float (1.0*maskFactor));

void main()
{
	//Sería más rápido utilizar una variable uniform el tamaño de la textura.
	vec2 ts = vec2(1.0) / vec2 (textureSize (colorTex,0));

	vec4 color = vec4 (0.0);
	for (uint i = 0u; i < MASK_SIZE; i++)
	{
```

[1] http://www.fit.vutbr.cz/study/courses/ISS/public/demos/conv/
http://mathworld.wolfram.com/Convolution.html

```
		vec2 iidx = texCoord + ts * texIdx[i];
		color += texture(colorTex, iidx,0.0) * mask[i];
	}

	outColor = color;
}
```

5. Añade una máscara de convolución 5x5 en "*postProcessing.v1.frag*":

```
#define MASK_SIZE 25u
const vec2 texIdx[MASK_SIZE] = vec2[](
	vec2(-2.0,2.0),  vec2(-1.0,2.0),  vec2(0.0,2.0),  vec2(1.0,2.0),  vec2(2.0,2.0),
	vec2(-2.0,1.0),  vec2(-1.0,1.0),  vec2(0.0,1.0),  vec2(1.0,1.0),  vec2(2.0,1.0),
	vec2(-2.0,0.0),  vec2(-1.0,0.0),  vec2(0.0,0.0),  vec2(1.0,0.0),  vec2(2.0,0.0),
	vec2(-2.0,-1.0), vec2(-1.0,-1.0), vec2(0.0,-1.0), vec2(1.0,-1.0), vec2(2.0,-1.0),
	vec2(-2.0,-2.0), vec2(-1.0,-2.0), vec2(0.0,-2.0), vec2(1.0,-2.0), vec2(2.0,-2.0));

const float maskFactor = float (1.0/65.0);
const float mask[MASK_SIZE] = float[](
	1.0*maskFactor, 2.0*maskFactor, 3.0*maskFactor,2.0*maskFactor, 1.0*maskFactor,
	2.0*maskFactor, 3.0*maskFactor, 4.0*maskFactor,3.0*maskFactor, 2.0*maskFactor,
	3.0*maskFactor, 4.0*maskFactor, 5.0*maskFactor,4.0*maskFactor, 3.0*maskFactor,
	2.0*maskFactor, 3.0*maskFactor, 4.0*maskFactor,3.0*maskFactor, 2.0*maskFactor,
	1.0*maskFactor, 2.0*maskFactor, 3.0*maskFactor,2.0*maskFactor, 1.0*maskFactor);
```

Paso 5: *Depth of field*

Modificaciones en el *shader* de fragmentos "*postProcessing.v1.frag*":

1. Añade una textura con la posición del fragmento en coordenadas de la cámara:

```
uniform sampler2D vertexTex;
```

2. Añade la distancia de enfoque y el desenfoque máximo:

```
const float focalDistance = -25.0;
const float maxDistanceFactor = 1.0/5.0;
```

3. Modifica el código de la función principal:

```
void main()
{
	//Sería más rápido utilizar una variable uniform el tamaño de la textura.
	vec2 ts = vec2(1.0) / vec2 (textureSize (colorTex,0));

	float dof = abs(texture(vertexTex,texCoord).z -focalDistance)
			* maxDistanceFactor;
	dof = clamp (dof, 0.0, 1.0);
	dof *= dof;


	vec4 color = vec4 (0.0);
	for (uint i = 0u; i < MASK_SIZE; i++)
	{
		vec2 iidx = texCoord + ts * texIdx[i]*dof;
		color += texture(colorTex, iidx,0.0) * mask[i];
	}

	outColor = color;
}
```

En el fichero “main.cpp”:

4. Cambia el modo de acceso de la textura de color (`void resizeFBO(unsigned int w, unsigned int h)`):

```
glBindTexture(GL_TEXTURE_2D, colorBuffTexId);
…

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_FILTER, GL_LINEAR);
…
```

Modifica el *shader* de fragmentos encargado de la primera pasada para que almacene la posición cada vértice:

5. Añade una nueva salida en “fwRendering.v1.frag”:

```
layout(location = 1) out vec4 outVertex;
```

6. Asígnale valor en la función principal del *shader:*

```
outVertex = vec4(pos,1.0);
```

En el fichero “main.cpp”, modifica el FBO para que incluya una textura adicional:

7. Añade las variables que nos permitan configurar esta textura:

```
unsigned int uVertexTexPP;
unsigned int vertexBuffTexId;
```

8. Configura la carga de los *shaders* de post-proceso para acceder a la nueva textura `void initShaderPP(const char *vname, const char *fname)`:

```
uVertexTexPP = glGetUniformLocation(postProccesProgram, "vertexTex");
```

9. Añade una nueva textura al FBO:

```
En la función: void initFBO()
...
glGenTextures(1, &vertexBuffTexId);


En la función: void resizeFBO(unsigned int w, unsigned int h)
...
glBindTexture(GL_TEXTURE_2D, vertexBuffTexId);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, w, h, 0,
        GL_RGBA, GL_UNSIGNED_BYTE, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

//... Después de activar el FBO

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
       GL_TEXTURE_2D, vertexBuffTexId, 0);

const GLenum buffs[2] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, buffs);
```

10. Libera el los recursos reservados (`void destroy()`):

```
glDeleteTextures(1, &vertexBuffTexId);
```

11. Configura la textura antes de renderizar (`void renderFunc()`):

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

Parte obligatoria

En este bloque deberá implementarse la siguiente funcionalidad:

1. **Controla los parámetros del *Motion Blur* a través de teclado.**
2. **Controla los parámetros del DOF por teclado** (distancia focal y distancia de desenfoque máximo).
3. **Utiliza el buffer de profundidad para controlar el DOF (ver Apéndice A)**.
4. Sube nuevas máscaras de convolución a través de variables *uniform.* Selecciona entre varias máscaras a través de teclado.
    a. http://docs.gimp.org/nl/plug-in-convmatrix.html

Parte opcional

En este apartado se describen las partes que podrán realizarse de forma opcional.

1. Concatena varios filtros Gausianos.
2. Concatena varios post-proceso distintos.
3. Implementa la funcionalidad de las prácticas 1, 2 y 3:
    a. Mejora el comportamiento de las texturas utilizando un filtro anisotrópico. (Ojo excluye las texturas del FBO)
    b. Controla el giro de la cámara utilizando el ratón.
    c. 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.*
    d. Crea un nuevo modelo y añádelo a la escena.
        i. Calcula las normales y/o las tangentes de dicho modelo.
    e. Ilumina el objeto con luces de distinto tipo.
    f. Añade atenuación con la distancia.
    g. Implementa *Bump Mapping.*
    h. Añade una textura especular.
4. **Implementa un sombrado basado en Deferred Shading.**

Apéndice A: Cálculo de la profundidad de un fragmento en coordenadas de la cámara a partir del buffer de profundidad

OpenGl utiliza la siguiente matriz **Mp** para proyectar los vértices del modelo en coordenadas de la cámara (los valores proyectados se normalizan en el rango [1, -1]):

$$M_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} a_{00} & 0 & a_{02} & 0 \\ 0 & a_{11} & a_{12} & 0 \\ 0 & 0 & a_{22} & a_{23} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Si $v_c = \{v_{c,x}, v_{c,y}, v_{c,z}, 1\}$ es el vértice en coordenadas de la cámara y es el $v_p = \{v_{p,x}, v_{p,y}, v_{p,z}, v_{p,w}\}$ vértice proyectado en coordenadas normalizadas:

$$M_p \cdot v_c = v_p,$$

luego:

$$M_p^{-1} \cdot v_p = v_c$$

Nota: para pasar a coordenadas cartesianas se divide por la coordenada homogénea.

Conociendo al inversa de Mp

$$M_p^{-1} = \begin{bmatrix} \frac{1}{a_{00}} & 0 & 0 & \frac{1}{a_{02}} \\ 0 & \frac{1}{a_{11}} & 0 & \frac{a_{12}}{a_{00}} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & \frac{1}{a_{23}} & \frac{a_{22}}{a_{23}} \end{bmatrix},$$

podemos volver a calcular el valor del vértice en coordenadas de la cámara:

$$M_p^{-1} \cdot \begin{bmatrix} v_{p,x} \\ v_{p,y} \\ v_{p,z} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{a_{02}}{a_{00}} + \frac{v_{p,x}}{a_{00}} \\ \frac{a_{12}}{a_{11}} + \frac{v_{p,y}}{a_{11}} \\ -1 \\ \frac{a_{22}}{a_{23}} + \frac{v_{p,z}}{a_{23}} \end{bmatrix}$$

Fijémonos en el cálculo de la coordenada z.

$$v_{c,z} = -\frac{a_{2,3}}{v_{c,z} + a_{2,2}} = -\frac{2nf}{f + n + v_{p,z}(n - f)}$$

<u>En el buffer de profundidad el valor $v_{b,z}$ se almacena normalizado entre [0,1] de forma que:</u>

$$v_{p,z} = 2v_{b,z} - 1$$

de esta forma

$$v_{c,z} = -\frac{nf}{f + v_{b,z}(n - f)}$$