Escuela Politécnica Superior de Jaén
Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

PROTOTIPO DE APLICACIÓN PARA CONFIGURAR AUTOMÁTICAMENTE PARÁMETROS DE VISIÓN E ILUMINACIÓN DE ESCENAS 3D

Alumno

Jesús Moreno Arjonilla

Tutor

Francisco de Asís Conde Rodríguez
(Departamento de Informática)

Julio, 2019

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Don Francisco de Asís Conde Rodríguez , tutor del Trabajo Fin de Grado titulado: '**Prototipo de aplicación para configurar automáticamente parámetros de visión e iluminación de escenas 3D**', que presenta Don Jesús Moreno Arjonilla, otorga el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, julio de 2019

El alumno: El tutor:

Jesús Moreno Arjonilla Francisco de Asís Conde Rodríguez

(Página intencionalmente en blanco)

Agradecimientos

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	Específico
TFT en equipo	No
Fecha de asignación	12/11/2018
Descripción corta	Para un estudiante que programa por primera vez una aplicación gráfica realista, es frustrante, comprobar que los primeros resultados que obtiene no son del todo satisfactorios. Ello no se debe a que los algoritmos estén mal programados, sino a que los ajustes de los parámetros de visión e iluminación no son los correctos. Es necesaria mucha experiencia para conseguir escenas atractivas visualmente. Este Trabajo fin de grado, pretende ayudar a los estudiantes a conseguir esta experiencia a través de ejemplos interactivos y de la capacidad de sugerir parámetros estándares para las escenas de los estudiantes. En este TFG se desarrollará un prototipo de herramienta que permitirá cargar modelos de escenas, analizar dichos modelos y sugerir posiciones y ajustes de cámara y fuentes luminosas para conseguir una visualización atractiva de esa escena. También permitirá interactuar con las configuraciones sugeridas y ver el efecto de cambiar algún parámetro en tiempo real. El prototipo también permitirá visualizar algunas escenas predefinidas de ejemplo, para que el estudiante las pueda explorar y aprender mediante exploración qué configuraciones funcionan y qué configuraciones no. Por último, el prototipo permitirá salvar a disco los ajustes de una escena en un formato que el estudiante pueda incorporar de forma sencilla a sus propios programas. El resultado de este TFG se usará en la asignatura Programación de Aplicaciones Gráficas.

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

Contenido

1 - Especificación del proyecto 19

1.1 - Índice general 19
1.2 - Memoria 19
1.2.1 - Introducción 19
1.2.2 - Objeto del proyecto 19
1.2.3 - Antecedentes 20
1.2.4 - Descripción de la situación actual 21
1.2.4.1 - Descripción del entorno actual 21
1.2.4.2 - Resumen de las principales deficiencias identificadas 21
1.2.5 - Normas y referencias 22
1.2.5.1 - Métodos, herramientas, modelos, métricas y prototipos 22
1.2.5.2 - Mecanismos de control de calidad aplicados durante la redacción del proyecto 23
1.2.6 - Definiciones y abreviaturas 23
1.2.7 - Requisitos iniciales 32
1.2.8 - Alcance 33
1.2.9 - Hipótesis y restricciones 33
1.2.10 - Estudio de alternativas y viabilidad 33
1.2.11 - Descripción de la solución propuesta 35
1.2.12 - Organización y gestión del proyecto 35
1.2.13 - Planificación temporal 36
1.2.14 - Resumen del presupuesto 37
1.2.15 - Orden de prioridad de los documentos básicos del proyecto 37
1.3 - Especificaciones del sistema 37
1.4 - Presupuesto 40
1.5 - A1: Documentación de entrada 40
1.6 - A2: Análisis y Diseño del sistema 40
1.6.1 - Metodología de desarrollo 47
1.7 - A3: Estimación del tamaño y esfuerzo 47

2 - Desarrollo del proyecto 48

2.1 - Tecnologías utilizadas 48
2.2 - Diseño 49
2.2.1 - Diseño arquitectónico del sistema 49
2.2.2 - Diagramas de clases 49
2.2.3 - Diagramas de casos de uso 50
2.2.4 - Diagramas de secuencia 61
2.2.5 - Diseño de la interfaz y storyboards 68
2.3 - Implementación 102
2.4 - Pruebas finales 155
2.4.1 - Pruebas de verificación del sistema 155
2.4.2 - Pruebas de validación del sistema 156

3 - Conclusiones y trabajos futuros 157

4 - Apéndices 157

4.1 - Instalación y configuración del sistema 157

4.2 - Manuales de usuario 158
4.3 - Guía original del Trabajo Fin de Título 191

5 - Bibliografía 191

Índice de ilustraciones

Ilustración 1.1 Movimiento Orbit 43
Ilustración 1.2 Diagrama lightsource-lightapplicator....47
Ilustración 2.1 Diagrama de clase 50
Ilustración 2.2 Casos de uso: Añadir luz....51
Ilustración 2.3 Casos de uso: Eliminar luz 51
Ilustración 3.4 Casos de uso: Editar luz....52
Ilustración 2.5 Casos de uso: Cambiar objeto escena 53
Ilustración 2.6 Casos de uso: Guardar configuración 54
Ilustración 2.7 Casos de uso: Guardar configuración 55
Ilustración 2.8 Casos de uso: Activar/desactivar atenuación 55
Ilustración 2.9 Casos de uso: Modificar atenuación....56
Ilustración 2.10 Casos de uso: Activar/desactivar FOG 57
Ilustración 2.11 Casos de uso: Modificar FOG 57
Ilustración 2.12 Activar/desactivar conjunto de 3 luces....58
Ilustración 2.13 Casos de uso: Activar/desactivar sombras 59
Ilustración 2.14 Casos de uso: Modificar mapa de sombras....60
Ilustración 2.15 Casos de uso: Rotar alrededor del objeto....60
Ilustración 2.16 Diagramas de secuencia: Añadir luz 61
Ilustración 2.17 Diagramas de secuencia: Editar luz 62
Ilustración 2.18 Diagramas de secuencia: Eliminar luz....63
Ilustración 2.19 Diagramas de secuencia: Cambiar objeto escena....63
Ilustración 2.20 Diagramas de secuencia: Guardar configuración 64
Ilustración 2.21 Diagramas de secuencia: Cargar configuración 64
Ilustración 2.22 Diagramas de secuencia: Activar/desactivar atenuación....65
Ilustración 2.23 Diagramas de secuencia: Modificar atenuación....65
Ilustración 2.24 Diagramas de secuencia: Activar/desactivar FOG....66
Ilustración 2.25 Diagramas de secuencia: Modificar FOG 66
Ilustración 2.26 Diagramas de secuencia: Activar/desactivar conjunto de 3 luces....67
Ilustración 2.27 Diagramas de secuencia: Activar desactivar sombras....67
Ilustración 2.28 Diagramas de secuencia: Modificar mapa de sombras....68
Ilustración 2.29 Diagramas de secuencia: Rotar alrededor del objeto 68
Ilustración 2.30 Storyboards: Añadir luz. Selección de tipo 69
Ilustración 2.31 Storyboards: Añadir luz. Modificación de parámetros....70
Ilustración 2.32 Storyboards: Añadir luz. Confirmación....70
Ilustración 2.33 Storyboards: Añadir luz. Nueva luz en la lista de luces....71
Ilustración 2.34 Storyboards: Editar luz. Selección de la luz....72
Ilustración 2.35 Storyboards: Editar luz. Modificación de parámetros....72
Ilustración 2.36 Storyboards: Editar luz. Parámetros correctos....73
Ilustración 2.37 Storyboards: Editar luz. Parámetros incorrectos....73
Ilustración 2.38 Storyboards: Eliminar luz. Selección de la luz 74
Ilustración 2.39 Storyboards: Eliminar luz. Confirmación....75
Ilustración 2.40 Storyboards: Eliminar luz. Luz eliminada de la lista de luces....75
Ilustración 2.41 Storyboards: Cargar configuración. Desplegar menú “Archivo” 76

Ilustración 2.42 Storyboards: Cargar configuración. Opción “Cargar configuración” 77
Ilustración 2.43 Storyboards: Cargar configuración. Elección del archivo 77
Ilustración 2.44 Storyboards: Guardar configuración. Desplegar menú “Archivo” 78
Ilustración 2.45 Storyboards: Guardar configuración. Opción “Guardar configuración” 79
Ilustración 2.46 Storyboards: Guardar configuración. Selección de nombre y confirmación .79
Ilustración 2.47 Storyboards: Activar/desactivar atenuación. Desplegar menú “Herramientas” 80
Ilustración 2.48 Storyboards: Activar/desactivar atenuación. Opción “Atenuación” 81
Ilustración 2.49 Storyboards: Activar/desactivar atenuación. Atenuación activa 81
Ilustración 2.50 Storyboards: Activar/desactivar atenuación. Atenuación sin activar 82
Ilustración 2.51 Storyboards: Activar/desactivar FOG. Desplegar menú “Herramientas” 83
Ilustración 2.52 Storyboards: Activar/desactivar FOG. Opción “FOG” 83
Ilustración 2.53 Storyboards: Activar/desactivar FOG. FOG activo 84
Ilustración 2.54 Storyboards: Activar/desactivar FOG. FOG sin activar 84
Ilustración 2.55 Storyboards: Activar/desactivar conjunto de 3 luces. Desplegar menú “Herramientas” 85
Ilustración 2.56 Storyboards: Activar/desactivar conjunto de 3 luces. Opción “Conjunto de 3 luces” 86
Ilustración 2.57 Storyboards: Activar/desactivar conjunto de 3 luces. Conjunto de 3 luces activo 86
Ilustración 2.58 Storyboards: Activar/desactivar conjunto de 3 luces. Conjunto de 3 luces sin activar 87
Ilustración 2.59 Storyboards: Activar/desactivar sombras. Desplegar menú “Herramientas” 88
Ilustración 2.60 Storyboards: Activar/desactivar sombras. Opción “Sombras” 88
Ilustración 2.61 Storyboards: Activar/desactivar sombras. Sombras activadas 89
Ilustración 2.62 Storyboards: Activar/desactivar sombras. Sombras sin activar 89
Ilustración 2.63 Storyboards: Editar atenuación. Pestaña “Extras” 90
Ilustración 2.64 Storyboards: Editar atenuación. Modificación de los campos 91
Ilustración 2.65 Storyboards: Editar atenuación. Parámetros correctos 91
Ilustración 2.66 Storyboards: Editar FOG. Pestaña “Extras” 92
Ilustración 2.67 Storyboards: Editar FOG. Modificar campos 93
Ilustración 2.68 Storyboards: Editar FOG. Parámetros correctos 93
Ilustración 2.69 Storyboards: Editar mapa de sombras. Pestaña “Mapa de sombras” 94
Ilustración 2.70 Storyboards: Editar mapa de sombras. Selección de luz 95
Ilustración 2.71 Storyboards: Editar mapa de sombras. Modificación de campos 95
Ilustración 2.72 Storyboards: Editar mapa de sombras. Parámetros correctos 96
Ilustración 2.73 Storyboards: Editar mapa de sombras. Parámetros incorrectos 96
Ilustración 2.74 Storyboards: Cambiar objeto. Desplegar menú “Archivo” 97
Ilustración 2.75 Storyboards: Cambiar objeto. Selección submenú “Objetos” 98
Ilustración 2.76 Storyboards: Cambiar objeto. Selección objeto 98
Ilustración 2.77 Storyboards: Cambiar objeto. Objeto seleccionado 99
Ilustración 2.78 Storyboards: Mover cámara. Click y arrastre en escena 100
Ilustración 2.79 Storyboards: Salir de la aplicación. Opción 1: Desplegar menú “Archivo”..101
Ilustración 2.80 Storyboards: Salir de la aplicación. Opción 1: Click en salir 101
Ilustración 2.81 Storyboards: Salir de la aplicación. Opción 2: Click en la “X” 102
Ilustración 2.82 Implementación: Clase glwindow. Constructor glwindow 103

Ilustración 2.83 Implementación: Clase glwindow. Instancia glwindow 103
Ilustración 2.84 Implementación: Clase glwindow. Método static getInstance 103
Ilustración 2.85 Implementación: Clase glwindow. Método initializeGL.....104
Ilustración 2.86 Implementación: Clase glwindow. Método resizeGL.....104
Ilustración 2.87 Implementación: Clase glwindow. Método paintGL 104
Ilustración 2.88 Implementación: Clase glwindow. Método eventFilter 105
Ilustración 2.89 Implementación: Clase glwindow. Método window_refresh 105
Ilustración 2.90 Implementación: Clase renderer. Constructor renderer 105
Ilustración 2.91 Implementación: Clase renderer. Instancia renderer 105
Ilustración 2.92 Implementación: Clase renderer. Método static getInstance.....105
Ilustración 2.93 Implementación: Clase renderer. Método redimensionarAreaDibujo 106
Ilustración 2.94 Implementación: Clase renderer. Método refreshCallback.....106
Ilustración 2.95 Implementación: Clase renderer. Calcular matrices de visión y proyección de la luz.....107
Ilustración 2.96 Implementación: Clase renderer. Activar FBO y textura 107
Ilustración 2.97 Implementación: Clase renderer. Limpiar buffer de profundidad.....108
Ilustración 2.98 Implementación: Clase renderer. Cambiar viewport 108
Ilustración 2.99 Implementación: Clase renderer. Test de profundidad 108
Ilustración 2.100 Implementación: Clase renderer. Activar cull face y asignar cara frontal. 108
Ilustración 2.101 Implementación: Clase renderer. Dibujar objetos 109
Ilustración 2.102 Implementación: Clase renderer. Enlazar framebuffer principal.....109
Ilustración 2.103 Implementación: Clase renderer. Definir color de fondo 109
Ilustración 2.104 Implementación: Clase renderer. Activar multisample 109
Ilustración 2.105 Implementación: Clase renderer. Limpiar buffers de color y profundidad.110
Ilustración 2.106 Implementación: Clase renderer. Activar cull face y asignar caras traseras 110
Ilustración 2.107 Implementación: Clase renderer. Cambiar viewport 110
Ilustración 2.108 Implementación: Clase renderer. Función de profundidad.....110
Ilustración 2.109 Implementación: Clase renderer. Función de blend.....110
Ilustración 2.110 Implementación: Clase renderer. Función de blend.....111
Ilustración 2.111 Implementación: Clase renderer. Aplicar uniforms.....111
Ilustración 2.112 Implementación: Clase camera. Constructor camera 113
Ilustración 2.113 Implementación: Clase camera. Función calcularVectores.....114
Ilustración 2.114 Implementación: Clase camera. Función calcularMatrices 114
Ilustración 2.115 Implementación: Clase camera. Función cambiarRelacionAspecto.....115
Ilustración 2.116 Implementación: Clase camera. Función orbit_horizontal.....115
Ilustración 2.117 Implementación: Clase camera. Función orbit_vertical.....116
Ilustración 2.118 Implementación: Clase element3d. Atributo modelMatrix 116
Ilustración 2.119 Implementación: Clase element3d. Atributo texturaActual.....116
Ilustración 2.120 Implementación: Clase element3d. Función drawAsTriangles.....116
Ilustración 2.121 Implementación: Clase subdivisionprofile. Algoritmo subdivisión.....117
Ilustración 2.122 Implementación: Clase revolutionobject. Constructor revolutionobject.....118
Ilustración 2.123 Implementación: Clase revolutionobject. Algoritmo de revolución.....118
Ilustración 2.124 Implementación: Clase revolutionobject. Vector entrante 119
Ilustración 2.125 Implementación: Clase revolutionobject. Vector saliente 119
Ilustración 2.126 Implementación: Clase revolutionobject. Rotación de vectores 119

Ilustración 2.127 Implementación: Clase revolutionobject. Cálculo de la normal 120
Ilustración 2.128 Implementación: Clase relovutionobject. Revolución de tangentes.....120
Ilustración 2.129 Implementación: Clase revolutionobject. Coordenada u de textura 121
Ilustración 2.130 Implementación: Clase revolutionobject. Coordenada v de textura.....121
Ilustración 2.131 Implementación: Clase revolutionobject. Coordenadas de textura.....122
Ilustración 2.132 Implementación: Clase revolutionobject. Algoritmo topología 123
Ilustración 2.133 Implementación: Clase revolutionobject. Rellenar VBOs 124
Ilustración 2.134 Implementación: Clase revolutionobject. VBO tapa inferior 124
Ilustración 2.135 Implementación: Clase revolutionobject. VBO tapa superior 124
Ilustración 2.136 Implementación: Clase revolutionobject. Enlazar VAO cuerpo 125
Ilustración 2.137 Implementación: Clase revolutionobject. Enlazar IBO cuerpo.....125
Ilustración 2.138 Implementación: Clase revolutionobject. Dibujar 125
Ilustración 2.139 Implementación: Clase revolutionobject. Enlazar VAO tapa inferior 125
Ilustración 2.140 Implementación: Clase revolutionobject. Enlazar IBO tapa inferior.....125
Ilustración 2.141 Implementación: Clase revolutionobject. Enlazar IBO tapa superior.....125
Ilustración 2.142 Implementación: Clase revolutionobject. Enlazar IBO tapa superior.....125
Ilustración 2.143 Implementación: Clase revolutionobject. Activar shader.....126
Ilustración 2.144 Implementación: Clase revolutionobject. Paro de uniforms.....126
Ilustración 2.145 Implementación: Clase revolutionobject. Aplicar textura.....126
Ilustración 2.146 Implementación: Clase revolutionobject. Enlazar VAO cuerpo 126
Ilustración 2.147 Implementación: Clase revolutionobject. Enlazar IBO cuerpo.....126
Ilustración 2.148 Implementación: Clase revolutionobject. Dibujar 127
Ilustración 2.149 Implementación: Clase plane. Constructor plane.....127
Ilustración 2.150 Implementación: Clase plane. Creación del plano 128
Ilustración 2.151 Implementación: Clase plane. Algoritmo topología plano.....128
Ilustración 2.152 Implementación: Clase plane. Método calcularIndicesNM 129
Ilustración 2.153 Implementación: Clase texture. Constructor texture 129
Ilustración 2.154 Implementación: Clase texture. Carga textura 129
Ilustración 2.155 Implementación: Clase texture. Creación textura 130
Ilustración 2.156 Implementación: Clase texture. Creación textura color.....130
Ilustración 2.157 Implementación: Clase texture. Método aplicar 130
Ilustración 2.158 Implementación: Clase bibliotexture 131
Ilustración 2.159 Implementación: Clase vao. VAo y VBOs.....131
Ilustración 2.160 Implementación: Clase vao. IBO 131
Ilustración 2.161 Implementación: Clase vao. Constructor VAO.....132
Ilustración 2.162 Implementación: Clase vao. Método generarVBOPosNotm.....132
Ilustración 2.163 Implementación: Clase vao. Método generarVBOTexturas 132
Ilustración 2.164 Implementación: Clase vao. Método generarVBOTangentes 132
Ilustración 2.165 Implementación: Clase vao. Método generarIBOMallaTriangulos.....133
Ilustración 2.166 Implementación: Clase vao. Método enlazarVBO.....133
Ilustración 2.167 Implementación: Clase vao. Método enlazarIBO 133
Ilustración 2.168 Implementación: Clase lightsource. Atributos 134
Ilustración 2.169 Implementación: Clase lightsource. Método aplicar 134
Ilustración 2.170 Implementación: Clase lightapplicator 135
Ilustración 2.171 Implementación: Clase lightapplicatorambiental. Método aplicar 135
Ilustración 2.172 Implementación: Clase lightapplicatordireccional. Método aplicar 135

Ilustración 2.173 Implementación: Clase lightapplicatorpuntual. Método aplicar.....135
Ilustración 2.174 Implementación: Clase lightapplicatorspot. Método aplicar.....136
Ilustración 2.175 Implementación: Clase lightsource. Método aplicar.....136
Ilustración 2.176 Implementación: Clase lightsource. Método calcularVPMatrix.....137
Ilustración 2.177 Implementación: Clase lightsource. Generar textura mapa de sombras..137
Ilustración 2.178 Implementación: Clase lightsource. Tamaño textura.....137
Ilustración 2.179 Implementación: Clase lightsource. Parámetros textura.....138
Ilustración 2.180 Implementación: Clase lightsource. Definición del border color.....138
Ilustración 2.181 Implementación: Clase lightsource. Asignación del border color.....138
Ilustración 2.182 Implementación: Clase lightsource. Función de comparación.....138
Ilustración 2.183 Implementación: Clase lightsource. Creación del FBO.....139
Ilustración 2.184 Implementación: Clase lightsource. Configuración del FBO.....139
Ilustración 2.185 Implementación: Clase lightsource. Comprobación FBO.....139
Ilustración 2.186 Implementación: Clase lightsource. Enlazar framebuffer principal.....139
Ilustración 2.187 Implementación: Shader luz ambiental. Vertex shader luz ambiental.....141
Ilustración 2.188 Implementación: Shader luz ambiental. Fragment shader luz ambiental.142
Ilustración 2.189 Implementación: Modelo de reflexión difuso.....143
Ilustración 2.190 Implementación: Modelo de reflexión especular.....144
Ilustración 2.191 Implementación: Gráfico FOG.....145
Ilustración 2.192 Implementación: Función coseno.....146
Ilustración 2.193 Implementación: Función coseno modificada.....147
Ilustración 2.194 Implementación: Shader luz direccional sin sombraas. Entradas fragment shader.....147
Ilustración 2.195 Implementación: Shader luz direccional sin sombras. Función ads fragment shader.....148
Ilustración 2.196 Implementación: Shader luz direccional sin sombras. Función main fragment shader.....148
Ilustración 2.197 Implementación: Shader mapa de sombras. Vertex shader.....149
Ilustración 2.198 Implementación: Shader mapa de sombras. Fragment shader.....149
Ilustración 2.199 Implementación: Shader luz direccional con sombras. Vertex shader.....150
Ilustración 2.200 Implementación: Shader luz direccional como rim light. Fragment shader.....152
Ilustración 2.201 Implementación: Shader luz puntual. Cálculo vector l.....153
Ilustración 2.202 Implementación: Dibujo luz spot.....154
Ilustración 4.1 Instalación y configuración del sistema: Carpeta descomprimida.....157
Ilustración 4.2 Instalación y configuración del sistema: Contenido carpeta.....158
Ilustración 4.3 Manual de usuario: Ventana principal.....158
Ilustración 4.4 Manual de usuario: Añadir luz. Elección tipo de luz.....159
Ilustración 4.5 Manual de usuario: Añadir luz. Carga predeterminada.....160
Ilustración 4.6 Manual de usuario: Añadir luz. Click “Añadir luz”.....160
Ilustración 4.7 Manual de usuario: Añadir luz. Luz añadida.....161
Ilustración 4.8 Manual de usuario: Editar luz. Elección luz.....162
Ilustración 4.9 Manual de usuario: Editar luz. Campos cargados.....162
Ilustración 4.10 Manual de usuario: Editar luz. Campo editado correctamente.....163
Ilustración 4.11 Manual de usuario: Editar luz. Campo editado incorrectamente.....163
Ilustración 4.12 Manual de usuario: Eliminar luz. Elección luz.....164

Ilustración 4.13 Manual de usuario: Eliminar luz. Click “Eliminar luz”.....165
Ilustración 4.14 Manual de usuario: Eliminar luz. Luz eliminada.....165
Ilustración 4.15 Manual de usuario: Activar/desactivar atenuación. Desplegar menú “Herramientas” 166
Ilustración 4.16 Manual de usuario: Activar/desactivar atenuación. Click en la opción “Atenuación”.....167
Ilustración 4.17 Manual de usuario: Activar/desactivar atenuación. Atenuación activa.....167
Ilustración 4.18 Manual de usuario: Editar atenuación. Click en pestaña “Extras” 168
Ilustración 4.19 Manual de usuario: Editar atenuación. Modificar campos.....169
Ilustración 4.20 Manual de usuario: Editar atenuación. Campos modificados correctamente 169
Ilustración 4.21 Manual de usuario: Activar desactivar FOG. Desplegar menú “Herramientas” 170
Ilustración 4.22 Manual de usuario: Activar/desactivar FOG. Click en la opción “FOG” 171
Ilustración 4.23 Manual de usuario: Activar/desactivar FOG. FOG activado.....171
Ilustración 4.24 Manual de usuario: Editar FOG. Click en la pestaña “Extras” 172
Ilustración 4.25 Manual de usuario: Editar FOG. Modificar campos.....173
Ilustración 4.26 Manual de usuario: Activar/desactivar FOG. Campos modificados correctamente 173
Ilustración 4.27 Manual de usuario: Activar/desactivar conjunto de 3 luces. Desplegar menú “Herramientas” 174
Ilustración 4.28 Manual de usuario: Activar/desactivar conjunto de 3 luces. Click en la opción “Conjunto de 3 luces” 175
Ilustración 4.29 Manual de usuario: Activar/desactivar conjunto de 3 luces. Conjunto de 3 luces activo.....175
Ilustración 4.30 Manual de usuario: Activar/desactivar sombras. Desplegar menú “Herramientas” 176
Ilustración 4.31 Manual de usuario: Activar/desactivar sombras. Click en la opción “Sombras” 177
Ilustración 4.32 Manual de usuario: Activar/desactivar sombras. Sombras activas.....177
Ilustración 4.33 Manual de usuario: Editar mapa de sombras. Click en la pestaña “Mapa de sombras” 178
Ilustración 4.34 Manual de usuario: Editar mapa de sombras. Seleccionar luz.....179
Ilustración 4.35 Manual de usuario: Editar mapa de sombras. Carga de los campos 179
Ilustración 4.36 Manual de usuario: Editar mapa de sombras. Campo editado correctamente 180
Ilustración 4.37 Manual de usuario: Editar mapa de sombras. Campo editado incorrectamente 180
Ilustración 4.38 Manual de usuario: Mover cámara. Click y arrastre sobre la escena.....181
Ilustración 4.39 Manual de usuario: Mover cámara. Resultado.....182
Ilustración 4.40 Manual de usuario: Cambiar objeto. Desplegar menú “Archivo” 183
Ilustración 4.41 Manual de usuario: Cambiar objeto. Desplegar opción “Objetos” 183
Ilustración 4.42 Manual de usuario: Cambiar objeto. Click en el objeto 184
Ilustración 4.43 Manual de usuario: Cambiar objeto. Resultado 184
Ilustración 4.44 Manual de usuario: Cargar configuración. Desplegar menú “Archivo” 185

Ilustración 4.45 Manual de usuario: Cargar configuración. Click en la opción “Cargar configuración”.....186
Ilustración 4.46 Manual de usuario: Cargar configuración. Selección archivo.....186
Ilustración 4.47 Manual de usuario: Cargar configuración. Resultado 187
Ilustración 4.48 Manual de usuario: Guardar configuración. Desplegar menú “Archivo” 188
Ilustración 4.49 Manual de usuario: Guardar configuración. Click en la opción “Guardar configuración”.....189
Ilustración 4.50 Manual de usuario: Guardar configuración. Ruta y nombre del archivo.....189
Ilustración 4.51 Manual de usuario: Salir de la aplicación. Opción 1. Click en la “X”.....190
Ilustración 4.52 Manual de usuario: Salir de la aplicación. Opción 2. Desplegar menú “Archivo” y click en “Salir”.....190

Índice de tablas

Tabla 1.1 Presupuesto 40
Tabla 2.1 Casos de uso: Añadir luz 50
Tabla 2.2 Casos de uso: Eliminar luz 51
Tabla 2.3 Casos de uso: Editar luz 52
Tabla 2.4 Casos de uso: Cambiar objeto escena 52
Tabla 2.5 Casos de uso: Guardar configuración 53
Tabla 2.6 Casos de uso: Cargar configuración 54
Tabla 2.7 Casos de uso: Activar/desactivar atenuación 55
Tabla 2.8 Casos de uso: Modificar atenuación 56
Tabla 2.9 Casos de uso: Activar/desactivar FOG 56
Tabla 2.10 Casos de uso: Modificar FOG 57
Tabla 2.11 Casos de uso: Activar/desactivar conjunto de 3 luces 58
Tabla 2.12 Casos de uso: Activar/desactivar sombras 58
Tabla 2.13 Casos de uso: Modificar mapa de sombras 59
Tabla 2.14 Casos de uso: Rotar alrededor del objeto 60
Tabla 2.15 Pruebas de verificación del sistema 156
Tabla 2.16 Pruebas de validación del sistema 156

1 - ESPECIFICACIÓN DEL PROYECTO

En este capítulo se presenta la especificación del proyecto, con una estructura y contenidos que siguen los criterios y recomendaciones que establece la norma UNE 157801:2007 - "*Criterios Generales para la elaboración de proyectos de Sistemas de Información*".

1.1 - Índice general

Como índice de la documentación requerida por la norma UNE 157801 se utilizará en su lugar la tabla de contenido global del presente TFT, ya que todos los documentos indicados en dicha norma están contenidos en el mismo documento maestro.

1.2 - Memoria

A continuación, se presentan los contenidos de la especificación del proyecto definido en el presente TFT, y en el que se describen todos los elementos que deberán entregarse al final de la ejecución del proyecto, así como la especificación de los métodos, herramientas y métricas utilizadas para garantizar el éxito del mismo.

1.2.1 - Introducción

El proyecto consiste en una aplicación interactiva destinada a ayudar a los usuarios a una mejor comprensión de las técnicas y métodos de iluminación en escenas 3D.

El usuario tendrá la posibilidad de cargar distintos modelos 3D sobre los que se aplicará una iluminación predeterminada que, posteriormente, puede ser modificada a gusto del usuario. Esto le permitirá poder visualizar de forma interactiva el funcionamiento de distintos tipos de iluminación y de distintas técnicas que aportan mayor realismo a la iluminación de una escena.

1.2.2 - Objeto del proyecto

Desarrollar un prototipo de aplicación que permita:

- cargar escenas 3D generadas en prácticas de PAG.
- estudiar la escena cargada para sugerir automáticamente parámetros de visión e iluminación.
- interactuar con la configuración sugerida para ver el efecto que esos cambios producen en la imagen resultado.
- salvar a disco los parámetros de visión e iluminación para que los estudiantes de PAG puedan incorporarlos de forma sencilla en sus propias aplicaciones.
- mostrar escenas preconfiguradas de ejemplo para que los estudiantes puedan exportarlas y aprender las configuraciones que mejor funcionan.

1.2.3 - Antecedentes

Para un usuario que se esté cursando la asignatura de Programación de Aplicaciones Gráficas por primera vez, como era mi caso, la parte de la iluminación puede ser algo bastante complejo de entender y, más aún, si no puedes saber si los resultados obtenidos son o no los correctos.

Cuando te dispones a iluminar una escena 3D, tienes en mente el resultado que deseas obtener y cómo crees que podrías obtener dicho resultado en la realidad, pero, a la hora de reproducirlo en una escena virtual no es tan sencillo ya que hay muchos factores de la luz que no se pueden traducir directamente de la luz real a la escena y, por lo tanto, hay que usar ciertos trucos para conseguir el resultado esperado o, al menos, un resultado satisfactorio.

Es por esto por lo que me dispuse a realizar un proyecto sobre una aplicación que permitiese a un usuario novato poder probar distintas configuraciones de luces de una forma rápida e interactiva hasta conseguir la configuración que el usuario buscaba. Además, si el usuario está cursando la asignatura de Programación de Aplicaciones Gráficas, podría probar una configuración de luces que estuviese usando en sus prácticas y ver si el resultado que está obteniendo es o no el correcto, pudiendo así identificar más rápido errores de iluminación en sus prácticas.

1.2.4 - Descripción de la situación actual

En la actualidad disponemos de una variedad de aplicaciones que nos permiten iluminar escenas 3D como es el caso, por ejemplo, de Blender o Maya.

El problema es que todas estas aplicaciones no están enfocadas sólo a la iluminación de una escena, sino que también se extienden a otros ámbitos como son: el modelado de objetos, personajes, paisajes, terrenos y la animación de escenas, entre otros.

Esto hace que las aplicaciones sean muy pesadas y tengan una gran cantidad de opciones y herramientas que se ponen a disposición de usuario haciendo, en la mayoría de los casos, interfaces complejas que requieren de mucho tiempo y práctica para poder entenderlas y dominarlas.

Desde el punto de vista de un usuario novato en el ámbito de la iluminación de escenas 3D cuyo objetivo es poder probar distintos tipos de iluminación de una forma no tediosa para, después, poder exportar dichas configuraciones a cualquier aplicación deseada, este tipo de aplicaciones son demasiado complejas de usar y requieren de mucho conocimiento de las mismas para poder realizar pequeñas pruebas.

1.2.4.1 - Descripción del entorno actual

El entorno sobre el cual se va a enfocar mi proyecto son las prácticas de la asignatura de Programación de Aplicaciones Gráficas.

1.2.4.2 - Resumen de las principales deficiencias identificadas

Las principales deficiencias que quedan superadas con este proyecto son:

- dificultad a la hora de entender en medios escritos los distintos modelos de reflexión, iluminación y sombreado, así como sus aplicaciones.
- escasez de aplicaciones interactivas sencillas que permitan la configuración y exportación de la iluminación de una escena.
- dificultad a la hora de comprobar si una configuración de iluminación establecida en otra aplicación, está produciendo los resultados correctos.

- dificultad para poder experimentar de forma fácil y rápida distintas configuraciones de iluminación en una escena 3D.

1.2.5 - Normas y referencias

A continuación, se presentan las normas, reglamentos y referencias de cualquier tipo que han sido de aplicación en la elaboración del proyecto o en la ejecución del mismo. La norma UNE 157801 incluye aquí un apartado sobre bibliografía. Sin embargo, al integrarse el documento de especificación del proyecto en el documento global del TFG, la bibliografía de este capítulo se integra junto con la bibliografía del documento maestro.

Para el desarrollo de la interfaz se han tenido en cuenta una serie de reglas ofrecidas por la asignatura de Interacción Persona-Ordenador. Estas reglas han sido:

- Vincular significados prácticos e intuitivos a los colores primarios, rojo, verde, amarillo y azul, que son fáciles de aprender y recordar.
- Mantener el mensaje sencillo: no sobrecargar el significado del color vinculando más de un concepto a un solo color.
- Evitar cambiar el significado de los colores en diferentes pantallas, sobre todo cuando se usa para codificar o agrupar información.
- Hay que delimitar claramente el centro de interés, el que atraerá la mirada del espectador, y que depende de la composición.

1.2.5.1 - Métodos, herramientas, modelos, métricas y prototipos

Herramientas utilizadas en el desarrollo del proyecto:

- Qt: Software para el desarrollo multiplataforma de aplicaciones.
- Git: Sistema de control de versiones.
- Sourcetree: Cliente gratuito de git.
- Github: Consiste en una forja para alojar proyectos utilizando el sistema de control de versiones de git.

1.2.5.2 - Mecanismos de control de calidad aplicados durante la redacción del proyecto

El aseguramiento de la calidad en la redacción del proyecto implica la verificación de la completitud (falta de omisiones), la integridad de la documentación del proyecto, así como una redacción clara, concisa y entendible por todos los participantes e interesados en el proyecto. Además, deben establecerse mecanismos de verificación de la integridad y completitud de la documentación.

Al estar el proyecto desarrollado por un solo autor y verificado por un tutor, no es necesario llevar un documento de control de edición y revisión de la documentación. De esta forma, el autor del proyecto ha utilizado mecanismos básicos para la verificación de la integridad y completitud de la documentación del proyecto, que incluyen un control de versiones a nivel de documentación y un control sencillo de la trazabilidad de requisitos y especificaciones del proyecto.

1.2.6 - Definiciones y abreviaturas

- Ia: Intensidad de la luz ambiente. Pertenece al modelo de reflexión ambiente.
- Ka: Color de la luz ambiente
- Id: Intensidad de la luz difusa. Consiste en la intensidad del modelo de reflexión lambertiano o difuso en el que cada punto de la superficie se ven con la misma intensidad sin importar el ángulo con el que se mire. Define la intensidad del color en la escena.
- Kd: Color difuso. Define el color del objeto
- Is: Intensidad de la luz especular. Consiste en la intensidad con la que la luz produce brillo en la escena y pertenece el modelo de reflexión especular.
- Ks: Color especular. Define el color del brillo del objeto.
- $\hat{I}$: vector de iluminación que define la dirección de la luz. En los cálculos de iluminación se utiliza invertido, dirigido hacia la fuente luminosa, para poder realizar los cálculos.
- $\hat{n}$: Vector normal de un punto. El vector normal es siempre perpendicular a la superficie del punto.

- $\hat{r}$: Vector de reflexión. Consiste en la reflexión de -$\hat{I}$ con respecto a la normal en cada punto. Se utiliza para el modelo de reflexión especular.
- $\hat{v}$: Vector de visión de la cámara. En los cálculos de iluminación se utiliza invertido, dirigido hacia la cámara, para poder realizar los cálculos correctamente. Se utiliza en el modelo de reflexión especular.
- Ángulo menor: Consiste en el ángulo que se define en una luz tipo spot por debajo del cual la iluminación siempre vale 1 (máximo valor).
- Ángulo mayor: Consiste el ángulo que se define en una luz tipo spt por encima del cual la iluminación siempre vale 0 (no hay iluminación).
- Key light: Es la luz predominante en la escena. Establece el ángulo principal de iluminación y es la que tiene un mayor impacto en la imagen. Su posición y calidad tienen un papel muy importante a la hora de determinar el estado de la imagen y el relieve de los objetos. Es la responsable de la mayoría de sombras. No es necesariamente la luz más brillante de la escena.
- Fill light: La fill light es la luz que ilumina o rellena las zonas de sombra. Tiene siempre una menor intensidad que la key light. Tiene un papel de apoyo sobre la key light, controla el contraste y afecta al estado de la imagen. Las imágenes con una fill light débil tienen un mayor contraste y una mayor tensión visual mientras que, las imágenes con una fill light más fuerte, tienen menor contraste y menor tensión visual.
- Rim light: La rim light proporciona una fina línea de iluminación alrededor del objeto de interés. Su función principal es separar la parte frontal del objeto de interés del fondo. Esta separación realza la ilusión de profundidad y enfatiza el objeto de interés atrayendo el foco de atención.
- Posición: Posición de la fuente luminosa.
- Look At: Punto sobre el espacio al que está dirigida la fuente luminosa.
- FOG: También llamado niebla. Consiste en la atenuación de una fuente luminosa debido a la profundidad.
- ZMin: Plano a partir del cual empieza a haber atenuación atmosférica.
- ZMax: Plano a partir del cual la atenuación atmosférica es total.

$$fog = \begin{cases} 0, & z < zmin \\ \dfrac{z - zmin}{zmax - zmin}, & zmin < z < zmax \\ 1, & z > zmax \end{cases}$$

- Znear: Plano visible más cercano a la cámara.
- ZFar: Plano visible más lejano a la cámara.
- C1: Coeficiente de atenuación debido a la distancia.
- C2: Coeficiente de atenuación debido a la distancia.
- C3: Coeficiente de atenuación debido a la distancia.
- d: Distancia de la fuente luminosa al objeto iluminado

$$fatt = \min\left(\frac{1}{c_1 + c_2 * d + c_3 * d^2}, 1\right)$$

- Direccional: Consiste en un modelo de iluminación en el cual los rayos llegan todos paralelos a la escena.
- Spot: Consiste en un modelo de iluminación que genera un cono de iluminación.
- Puntual: Modelo de iluminación que define una luz que se propaga en todas las direcciones.
- Conjunto de 3 luces: Consiste en una configuración de la escena que parte de una luz principal (KeyLight) y crea a partir de ésta una Rim Light y una Fill light.
- Shadow map: Consiste en una textura que almacena los valores de profundidad de la escena desde el punto de vista de una fuente luminosa. Se utiliza para el proceso de shadow mapping.
- Shadow mapping: Proceso mediante el cual se añaden sombras a una escena 3D.

- Pipeline de rendering: Conjunto de procesos por los que pasa una geometría almacenada en la GPU hasta convertirse en una imagen en pantalla.
- Shader: Son pasos del pipeline de rendering que son programados por el usuario y que el mismo usuario puede colocar para que se ejecuta en una parte del pipeline de rendering. A cada programa se le denomina shader object. Estos shaders han de ser compilados por el usuario y, si la compilación es correcta, se enlazan a un shader program. Cada shader object va en un archivo que es leído en tiempo de ejecución.
- Shader program: Consiste en un conjunto de shader objects que son utilizados en todo un proceso de dibujo. Un shader program recibe los shader object que han sido correctamente compilados y se enlaza para su uso.
- Vertex shader: Consiste en un shader object que recibe la información de cada vértice de los objetos a dibujar y la procesa en función de lo que el usuario le haya definido. La salida de este shader consiste en la entrada de la siguiente etapa en el pipeline de rendering.
- Fragment shader: Consiste en un shader object que recibe la información de cada fragmento proveniente del rasterizador y la procesa en función de lo programado por el usuario para asignarle un color a cada fragmento.
- Depth buffer: Es una parte de la GPU que se encarga de gestionar los valores de profundidad de las imágenes en los gráficos tridimensionales.
- Viewport: Consiste en las coordenadas de mi ventana de visualización. Es necesario definirlo correctamente mediante la función glviewport para pasar de coordenadas de dispositivo a coordenadas de ventana.
- Relación de aspecto: Consiste en la relación entre el ancho y el alto de una ventana.
- Color buffer: Es un buffer que almacena la información de color de una imagen. Se encuentra dentro de un framebuffer.
- Depth buffer: Es un buffer que almacena la información de profundidad de una imagen. Se encuentra dentro de un framebuffer.

- VAO: Vertex Array Object. Consiste en un objeto de OpenGL que permite pasar una malla de triángulos a la GPU. Dentro contiene uno o más VBOs y uno o más IBOs.
- VBO: Vertex Buffer Object. Consiste en un objeto de OpenGL que contiene datos sobre la geometría de los vértices de una malla de triángulos (posiciones,normales…)
- IBO: Index Buffer Object. Objeto de OpenGL que contiene los índices que indican la topología de una malla de triángulos.
- FBO: Frame Buffer Object. Es un objeto de OpenGL que permite la creación de framebuffers definidos por el usuario. Con ellos, se puede renderizar en un framebuffer distinto al principal de forma que no afecte a la ventana principal.
- Framebuffer: Consiste en una colección de buffers que pueden ser usador para renderizar. OpenGL tiene dos tipos: el framebuffer por defecto que es suministrado por el contexto de OpenGL, y los framebuffers creados por los usuarios (FBOs).
- Singleton: Consiste en un patrón de diseño que nos permite tener una única instancia de una clase y, además, tener acceso a ella desde cualquier módulo de la aplicación.
- Geometría: Conjunto de puntos que forman un objeto.
- Topología: Orden en el que hay que utilizar la geometría para dibujar el objeto.
- Location: Atributo del cual hay que obtener un dato o en el cuál se va a almacenar un dato.
- Magnification: Es un fenómeno que ocurre cuando hay un único texel de textura que cae en varios píxeles.
- Minification: Es un fenómeno que ocurre cuando hay un único texel de textura que cae en muchos píxeles.
- Texel: Es la unidad mínima de una textura aplicada a una superficie.

- GL_Linear: Filtro que devuelve el promedio ponderado de los cuatro elementos que están más cerca de las coordenadas de textura especificadas.
- Built-in variables: Variables que se utilizan para pasar información de una etapa del pipeline de rendering a otra que son muy usadas y vienen predefinidas en la GPU.
- Shadow acné: Consiste en el autosombreado erróneo de un objeto debido a errores de precisión.
- Antialiasing: Proceso que permite minimizar el solapamiento que se produce al intentar representar una señal de alta resolución en un sustrato de más baja resolución.
- Modelos de reflexión: Describen la interacción de la luz sobre una superficie en términos de las propiedades de la superficie y la naturaleza de la luz que recibe.
- Modelos de iluminación: Describe la naturaleza de la luz que emana de una fuente luminosa, la geometría de su distribución, etc.
- atenuacionEnabled: Variable booleana que se encarga de almacenar si la atenuación de la luz está activada o no.
- fogEnabled: Variable booleana que se encarga de almacenar si el fog está activado o no.
- keyIndex: Índice de la lista de luces en el cual se encuentra la Key light.
- sombras: Variable booleana que se encarga de almacenar si las sombras están activas o no.
- shadowBias: Consiste en una matriz que transforma nuestras coordenadas de sombra que van del intervalo [-1,1] al intervalo [0,1] que es el intervalo que utiliza nuestro mapa de sombras para así poder muestrear la información de él.
- visionMatrix: Matriz de visión de la cámara. Permite trasladar el mundo al espacio de visión, el espacio de la cámara

- proyectionMatrix: Matriz de proyección de la cámara. Nos permite proyectar nuestro mundo en el espacio de perspectiva de la cámara
- modelMatrix: Matriz de modelado del objeto. Nos permite colocar nuestro objeto en la posición que deseemos del mundo.
- visionMatrixLuz: Matriz de visión de la luz. Nos permite transladar el mundo al espacio de la luz.
- proyectionMatrixLuz: Matriz de proyección de la luz. Nos permite proyectar nuestro mundo en el espacio de perspectiva de la luz.
- fovYLuz: Consiste en el ángulo de apertura de la luz. Si se consigue ajustar muy bien a la escena, enfocándola bien, se consigue toda la información necesaria para el mapa de sombras.
- vpMatrixLuz: Matriz de proyección de la luz * matriz de visión de la luz.
- shadowFBO: FBO que se utiliza para generar el mapa de sombras
- threeLights: Variable booleana que almacena si están activas las luces rim y fill
- orbit horizontal: Movimiento de la cámara alrededor del punto de interés en su eje horizontal.
- orbit vertical: Movimiento de la cámara alrededor del punto de interés en su eje vertical.
- slices: Número de cortes longitudinales del objeto de revolución
- subdivisiones: Número de veces que hay que aplicar el algoritmo de subdivisión de polilíneas para conseguir curvas suaves en mi objeto.
- Objeto de subdivisión: Consiste en un objeto generado a partir de un perfil de subdivisión que se ha revolucionado alrededor de un eje.
- Perfil de subdivisión. Consiste en el perfil inicial de un objeto de revolución que va a definir la forma del objeto una vez que éste se revolucione.
- vboPosNorm: VBO que contiene la información de posiciones y normales de cada vértice

- vboTangentes: VBO que contiene información de las tangentes de cada vértice.
- vboTexturas: VBO que almacena información de las coordenadas de texturas de cada vértice.
- iboNubePuntos: IBO que almacena los índices para poder dibujar mi objeto como nube de puntos
- iboAlambre: IBO que almacena los índices para poder dibujar mi objeto como alambre
- iboMallaTriangulos: IBO que almacena los índices para poder dibujar mi objeto como malla de triángulos.
- posNormCuerpo: Array que almacena la información de posiciones y normales de cada vértice del cuerpo del objeto. Esta información será después pasada a su correspondiente VBO.
- texturasCuerpo: Array que almacena la información de las coordenadas de textura de cada vértice del cuerpo del objeto. Esta información será después pasada a su correspondiente VBO.
- tangentesCuerpo: Array que almacena la información de las tangentes de cada vértice del cuerpo del objeto. Esta información será después pasada a su correspondiente VBO.
- posNormTapaI: Array que almacena la información de posiciones y normales de la tapa inferior del objeto (si hubiese). Esta información será después pasada a su correspondiente VBO.
- texturasTapaI: Array que almacena la información de las coordenadas de textura de la tapa inferior del objeto (si hubiese). Esta información será después pasada a su correspondiente VBO.
- tangentesTapaI: Array que almacena la información de las tangentes de cada vértice de la tapa inferior del objeto (si hubiese). Esta información será después pasada a su correspondiente VBO.

- posNormTapaS: Array que almacena la información de posiciones y normales de la tapa superior del objeto (si hubiese). Esta información será después pasada a su correspondiente VBO.
- texturasTapaS: Array que almacena la información de las coordenadas de textura de la tapa superior del objeto (si hubiese). Esta información será después pasada a su correspondiente VBO.
- tangentesTapaS: Array que almacena la información de las tangentes de cada vértice de la tapa superior del objeto (si hubiese). Esta información será después pasada a su correspondiente VBO.
- indicesCuerpoNM: Array que almacena los índices para poder dibujar el cuerpo del objeto como nube de puntos o como malla de triángulos. Esta información será después pasada a su correspondiente IBO.
- indicesTapaINM: Array que almacena los índices para poder dibujar la tapa inferior del objeto como nube de puntos o como malla de triángulos. Esta información será después pasada a su correspondiente IBO.
- indicesTapaSNM: Array que almacena los índices para poder dibujar la tapa superior del objeto como nube de puntos o como malla de triángulos. Esta información será después pasada a su correspondiente IBO.
- indicesCuerpoA: Array que almacena los índices para poder dibujar el cuerpo del objeto como alambre. Esta información será después pasada a su correspondiente IBO.
- indicesTapaIA: Array que almacena los índices para poder dibujar laa tapa inferior del objeto como alambre. Esta información será después pasada a su correspondiente IBO.
- indicesTapaSA: Array que almacena los índices para poder dibujar laa tapa superior del objeto como alambre. Esta información será después pasada a su correspondiente IBO.
- vaoCuerpo: VAO que contiene los VBOs e IBOs necesarios para poder dibujar el cuerpo del objeto

- vaoTapaS: VAO que contiene los VBOs e IBOs necesarios para poder dibujar la tapa superior del objeto (si hubiese).
- vaoTapaI: VAO que contiene los VBOs e IBOs necesarios para poder dibujar la tapa inferior del objeto (si hubiese).

1.2.7 - Requisitos iniciales

- El usuario debe tener la posibilidad añadir nuevas luces a la escena.
- El usuario debe tener la posibilidad de editar luces ya existentes, así como eliminarlas.
- El usuario debe poder ver una escena 3D con un objeto dónde se aplicarán todas las configuraciones establecidas.
- El usuario debe tener la posibilidad de exportar e importar configuraciones de luces.
- El sistema debe responder al usuario con el feedback necesario cuando éste interactúe con el mismo.
- El usuario debe tener la posibilidad de cambiar el objeto en escena por otro objeto predeterminado.
- El usuario debe tener la posibilidad de activar o desactivar el uso de la atenuación.
- El usuario debe tener la posibilidad de activar y desactivar el uso del FOG.
- El usuario debe tener la posibilidad de activar y desactivar el uso de sombras.
- El usuario debe tener la posibilidad de activar o desactivar el uso de las luces fill y rim.
- El usuario debe tener la posibilidad de mover la cámara alrededor del objeto.
- El usuario debe tener la posibilidad de modificar los parámetros que afectan a la atenuación si ésta está activa.
- El usuario debe tener la posibilidad de modificar los parámetros del FOG si éste está activo.

- El usuario debe tener la posibilidad de modificar los parámetros que afectan a las sombras que produce una luz si la opción de sombras está activa.
- Un estudiante de la asignatura de Programación de Aplicaciones Gráficas debe poder utilizar la aplicación tras 10 minutos de entrenamiento.
- El sistema debe de refrescar la escena de forma instantánea cada vez que el usuario modifique algún valor de la iluminación.

1.2.8 - Alcance

El proyecto incluye los siguientes entregables:

- Ejecutable
- Manual de usuario
- Documentación del proyecto

1.2.9 - Hipótesis y restricciones

El TFT se define como una asignatura de 12 créditos, lo que supone que la duración total del proyecto será de 300 horas, incluyendo todas las etapas del ciclo de vida, con la excepción del mantenimiento. Por consiguiente, la principal restricción aplicable es la limitación de la duración del trabajo.

1.2.10 - Estudio de alternativas y viabilidad

En lo que respecta al estudio de las alternativas, se han contemplado varias alternativas tanto para el uso del software de desarrollo como para el desarrollo completo de algunas funcionalidades.

Voy a pasar a enumerar algunas alternativas que se contemplaron para la creación de la interfaz gráfica:

- NanoGUI: Es una librería minimalista multiplataforma para OpenGL 3 o superior. Soporta la creación automática de layouts, callbacks y una gran variedad de widgets. El motivo por lo que finalmente no fue elegida es porque los widgets se crearían como ventanas aparte a la ventana de visualización, no estaría todo integrado en una misma ventana que es lo que realmente quería.

- Qt: Es un software para desarrollo multiplataforma que integra una librería propia para el desarrollo de aplicaciones en C++ o Python. Además, su software incluye un apartado para la creación de interfaces gráficas muy intuitivo y que nos permite tener todo organizado dentro de una misma ventana. También posee mucha documentación y una comunidad muy activa. Todo esto me llevó a elegir Qt como la librería y software de desarrollo.

En cuanto al software de desarrollo, hubo otra alternativa a Qt que era Microsoft Visual Studio. La librería de Qt que elegí para la interfaz gráfica proporcionaba integración con Visual Studio, pero, al tener Qt su propio software de desarrollo, pensé que era mejor utilizarlo para evitar problemas que pudiera darme el enlazado de las librerías a Visual Studio.

El resto de alternativas que han surgido han sido en la implementación de funcionalidades. Voy a pasar a enumerar algunas:

- En la implementación de una luz spot, se puede considerar implementarla con un único ángulo que define la zona iluminada y la zona no iluminada o, mediante dos ángulos. Un ángulo (el menor) definiría la zona de máxima iluminación y, el otro ángulo (el mayor) definiría la zona a partir de la cual no habría iluminación. Entre los dos ángulos existiría una atenuación de la luz. Se decidió por implementar la luz spot con dos ángulos puesto que daba un resultado mucho más realista.

- En la implementación de la atenuación debida a la profundidad (fog), esta funcionalidad podía implementarse mediante una interpolación lineal o, haciendo uso de la función coseno. Se decidió finalmente por la función coseno ya que daba resultados mucho más realistas y el salto de atenuación entre puntos cercanos era mucho más suave.

- En la implementación de las sombras, podía implementarse el factor de sombra como el factor resultante de consultar en el mapa de sombras unas coordenadas de textura dadas o, podía implementarse este factor como la media del resultado de consultar en el mapa de sombras unas coordenadas de sombra dadas y las coordenadas vecinas. Se decidió la segunda implementación porque evitaba el aliasing.

En cuanto a la viabilidad, desde un primer momento este proyecto fue definido como una herramienta de ayuda gratuita a los usuarios que cursasen la asignatura de Programación de Aplicaciones Gráficas, así como para aquellas personas novatas en el ámbito de la programación gráfica que quisieran distintas configuraciones de iluminación de una forma más sencilla e interactiva para, después, poder exportarlas a sus proyectos.

Se puede contemplar este proyecto como un prototipo de aplicación para los usuarios a través del cual, en un futuro, podría recopilarse información de su uso o información proveniente de los propios usuarios (sugerencias) para realizar una aplicación con mayores funcionalidades y mejoras que pudiese salir al mercado.

1.2.11 - Descripción de la solución propuesta

La propuesta realizada se trata de una aplicación de escritorio, altamente interactiva e intuitiva, que permite a usuarios novatos en el ámbito de la informática gráfica experimentar con la iluminación de una escena con el propósito de conseguir los resultados buscados.

Entre las características más significativas de la aplicación tenemos:

- Posibilidad de añadir nuevas luces a la escena, así como poder editar o eliminar luces ya existentes.
- Posibilidad de importar y exportar configuraciones realizadas
- Posibilidad de activar y desactivar distintas opciones que afectan a la iluminación de la escena como son la atenuación, el FOG, las luces fill y rim y el uso de sombras.
- Posibilidad de poder editar los parámetros de las distintas opciones mencionadas anteriormente para aproximarse al resultado deseado.

1.2.12 - Organización y gestión del proyecto

1. Organigrama y Matriz de responsabilidades en el proyecto:
    - Jefe de proyecto: Francisco de Asís Conde Rodríguez.
    - Desarrollador: Jesús Moreno Arjonilla.
2. Directrices para el seguimiento del proyecto:

Para un correcto seguimiento del proyecto se utilizará el control de versiones de Git y la plataforma Github donde se deberán de publicar todos los cambios realizados y comunicar las tareas finalizadas.

3. Directrices a seguir para la aprobación de los entregables:

   Se deberá entregar el ejecutable de la aplicación diseñada junto con un manual de usuario donde se explique detalladamente el funcionamiento de la aplicación y como hacer uso de las distintas opciones.

   Además, se entregará toda la documentación asociada al proyecto.

4. Lugar donde se realizará el trabajo:

   El desarrollador podrá realizar el trabajo desde su casa.

1.2.13 - Planificación temporal

Para la realización del proyecto he seguido una metodología tradicional de análisis-diseño-implementación-pruebas.

La realización del proyecto se ha dividido en las siguientes tareas:

1. Elección de la librería a utilizar para la interfaz gráfica. Tiempo empleado: 28 enero-7 febrero.
2. Adaptación de las prácticas de Programación de Aplicaciones Gráficas al entorno de programación y lenguaje de Qt. Tiempo empleado: 8 febrero-5 junio.
3. Creación de la interfaz gráfica de la aplicación. Tiempo empleado: 6 junio-11 junio.
4. Implementación de la opción de atenuación. Tiempo empleado: 12 junio-21 junio.
5. Implementación de la opción de FOG. Tiempo empleado: 21 junio-26 junio.
6. Implementación de la opción de luces key light, fill light y rim light. Tiempo empleado: 26 junio-5 julio.
7. Implementación de la opción de exportar e importar configuraciones. Tiempo empleado: 5 julio- 6 julio.

8. Implementación de la opción de sombras. Tiempo empleado: 7 julio-20 agosto.

1.2.14 - Resumen del presupuesto

Parala realización de este proyecto será necesario:

- Un desarrollador informático con conocimientos sobre OpenGL y la librería Qt. Coste estimado: 4500€.
- Depreciación o uso informático de la computadora del desarrollador. Coste estimado: 240€.
- Utilización de los servicios eléctricos y de internet. Coste estimado: 120€.

Para llevar a cabo el proyecto es necesario disponer de un presupuesto total estimado de 4860 €.

1.2.15 - Orden de prioridad de los documentos básicos del proyecto

El proyecto se presenta dentro de este capítulo del documento global con los contenidos indicados en la norma UNE 157801. El orden de prioridad utilizado es el siguiente:

1. Memoria del proyecto.
2. Especificaciones del sistema.
3. Presupuesto.
4. Estudios con entidad propia.
5. Anexos según la norma (numerados como A1...AN).

1.3 - Especificaciones del sistema

1. Requisitos funcionales:
    - El usuario debe tener la posibilidad de añadir nuevas luces a la escena. Para ello, deberá poder elegir el tipo de luz que desea añadir y modificar sus parámetros a gusto.

- El usuario debe tener la posibilidad de editar luces ya existentes, así como eliminarlas. Para conseguir esto, deberá poder elegir una luz de las existentes y poder modificar sus parámetros o eliminarla directamente.
- El usuario debe tener la posibilidad de visualizar una escena 3D con un objeto en la cual se apliquen todas las configuraciones realizadas. Para ello deberá haber una ventana dedicada a la visualización de la escena.
- El usuario debe tener la posibilidad de importar y exportar configuraciones de luces. Para esta funcionalidad, Para ello deberá tener la opción de poder exportar su configuración actual a un fichero para, después, poder importarla en cualquier momento.
- El sistema debe responder al usuario con el feedback necesario cuando éste interactúe con el mismo.
- El usuario debe tener la posibilidad de cambiar el objeto en escena por otro objeto predeterminado. Para esta funcionalidad, deberá de tener acceso a los distintos objetos predeterminados de la aplicación y deberá poder elegir en todo momento qué objeto quiere que aparezca en la escena.
- El usuario deberá tener la posibilidad de activar y desactivar el uso de la atenuación. Para ello, deberá tener acceso a un menú que le permita en todo momento activar o desactivar la atenuación.
- El usuario deberá activar y desactivar el uso del FOG. Para conseguir esto, deberá tener acceso en todo momento a un menú que le permita activar o desactivar el FOG.
- El usuario debe tener la posibilidad de activar y desactivar el uso de sombras. Para esta funcionalidad, deberá de tener acceso en todo momento a un menú que le permita activar y desactivar el uso de sombras.
- El usuario debe tener la posibilidad de activar y desactivar el uso de las luces fill y rim. Para ello, deberá de tener acceso en todo momento

a un menú que le permita activar y desactivar el uso del conjunto de 3 luces (key,fill,rim).

- El usuario debe tener la posibilidad de mover la cámara alrededor del objeto. Para conseguir esto, deberá poder interactuar con la escena mediante el uso del ratón.
- El usuario debe tener la posibilidad de modificar los parámetros que afectan a la atenuación si ésta está activa. Para esta funcionalidad, deberá poder acceder a una pestaña desde la cuál pueda realizar modificaciones en los parámetros que afecta a la atenuación.
- El usuario debe tener la posibilidad de modificar los parámetros que afectan al FOG si éste está activo. Para ello, deberá de tener acceso a una pestaña desde la cual pueda realizar modificaciones en los parámetros que afectan al FOG.
- El usuario debe tener la posibilidad de modificar los parámetros que afectan a las sombras que produce una luz si la opción de sombras está activa. Para conseguir esto, deberá de tener acceso a una pestaña que le permita realizar modificaciones en los parámetros que afectan a la generación de sombras en la escena.

2. Requisitos nos funcionales:

- Un estudiante de la asignatura de Programación de Aplicaciones Gráficas debe poder utilizar la aplicación tras 10 minutos de entrenamiento. Este tiempo de entrenamiento es necesario para que el usuario se familiarice con la interfaz y las distintas opciones de la misma. Pasado este tiempo el usuario será capaz de usar la interfaz con todas sus opciones sin problema alguno y con una mínima probabilidad de errores.
- El sistema debe refrescar la escena de forma instantánea cada vez que el usuario modifique algún valor de la iluminación. De esta forma el usuario podrá visualizar de forma rápida los resultados de los cambios que vaya realizando y comprobar si son los resultados buscados o no.

1.4 - Presupuesto

Componente	Meses	Horas/día	Cantidad	Costo unitario	Costo total
Mano de obra					
Desarrollador informático	3	8h	480	9,375€	4500€
Hardware					
Depreciación o uso informático de la computadora del desarrollador	3	8h	480	0,50€	240€
Software					
Qt	0	0	1	0	0
Git	0	0	1	0	0
Sourcetree	0	0	1	0	0
Servicios					
Energía eléctrica	3	0	3	0,15€	15€
Internet	3	0	3	35€	105€
					Total
					4860€

Tabla 1.1 Presupuesto

1.5 - A1: Documentación de entrada

Como equivalente al pliego de condiciones se incluye la documentación de entrada especificada en el Apéndice Guía original del Trabajo Fin de Título.

1.6 - A2: Análisis y Diseño del sistema

- Análisis del sistema:

Partiendo de los requisitos iniciales anteriormente descritos, el sistema se va a modelar de la siguiente forma:

Para permitir que el usuario pueda añadir nuevas luces a la escena, se implementará una interfaz gráfica que permita al usuario, mediante un desplegable, elegir el tipo de luz que quiere añadir a la escena (direccional, puntual o spot). Además, la interfaz dispondrá de un apartado dónde aparecerán todos los atributos posibles que puede tener una luz y, una vez elegido el tipo de luz, se cargarán unos valores por defecto para dicha luz en los campos necesarios, desactivando los campos que son innecesarios para ésta. El usuario tendrá la posibilidad de modificar

dichos valores a gusto. Finalmente, una vez que el usuario tenga su configuración deseada, podrá confirmarla y añadir dicha luz a la escena mediante el botón de añadir.

Para que el usuario pueda editar una luz ya existente, la interfaz dispondrá de un listado con las luces que hay en escena. El usuario podrá seleccionar una luz de este listado y se cargarán sus valores en los campos necesarios pudiendo el usuario en cualquier momento modificar dichos campos. Cada vez que se modifique un campo, el sistema comprobará que los parámetros introducidos son correctos y, de ser así, aplicará los cambios a la escena.

Del mismo modo, si el usuario desea eliminar una luz, deberá elegir la luz que desea eliminar de la lista de luces de la escena y hacer uso del botón de eliminar.

Todas estas funcionalidades (añadir, editar y eliminar luces) estarán agrupadas en un panel en la parte derecha de la aplicación. Este panel pertenecerá a una pestaña que se llamará “Luces”.

En cuanto a la funcionalidad que permita al usuario ver la escena 3D dónde se apliquen todas las configuraciones realizadas, la interfaz implementará una ventana de dedicada a la visualización 3D de una escena la cual se refrescará cada vez que el usuario haga un cambio en la configuración de la misma.

Para la funcionalidad de importar y exportar configuraciones, la interfaz debe disponer de un menú desplegable dónde se le dará la posibilidad al usuario de importar una configuración previamente guardada. Esta opción desplegará un explorador de archivos que permitirá al usuario buscar el archivo de configuración que desea cargar. El archivo deberá tener la extensión .txt para que además sea fácilmente legible por el usuario desde un bloc de notas o una aplicación de texto.

Dicho esto, la funcionalidad de exportar una configuración se implementaría de forma similar. El usuario podrá acceder a esta opción desde un menú desplegable y se le abrirá un explorador de archivos para que pueda elegir la ruta y nombre del fichero que guardará la configuración actual de su escena.

En cuanto a el objeto que se muestra en escena, el usuario debe de tener la posibilidad de cambiar el objeto por otro predeterminado de la aplicación. Para conseguir esto se implementará un menú desplegable con un submenú que le permita elegir entre los distintos tipos de objetos que ofrece la aplicación. De esta forma el usuario puede elegir en cada momento el objeto de la escena.

En lo que respecta a la atenuación, FOG, conjunto de tres luces (key,fill,rim) y las sombras, el usuario deberá poder activar y desactivar cualquiera de estas

funcionalidades extras cuando lo desee. Para ello se dispondrá de un menú desplegable con herramientas extras que ofrece la aplicación y que pueden ser activadas o desactivadas en cualquier momento. En este menú se encontrarán las opciones de activar/desactivar atenuación, activar/desactivar FOG, activar/desactivar conjunto de 3 luces y activar/desactivar sombras.

Para la edición de la atenuación, se creará una nueva pestaña en el panel derecho de la aplicación llamada “Extras”. Desde esta pestaña se podrá acceder a los parámetros de funcionalidades extras como son la atenuación y el FOG.

En el caso de la atenuación, si la opción de atenuación anteriormente descrita está activa, desde este apartado se podrán modificar los factores que afectan a la atenuación de una escena (c1, c2, c3).

$$fatt = \min\left(\frac{1}{c_1 + c_2 * d + c_3 * d^2}, 1\right)$$

En el caso del FOG, si la opción de FOG anteriormente descrita está activa, desde este apartado se podrán modificar los parámetros que afectan al FOG (zMin, zMax).

$$fog = \begin{cases} 0, & z < zmin \\ \dfrac{z - zmin}{zmax - zmin}, & zmin < z < zmax \\ 1, & z > zmax \end{cases}$$

Con respecto a la funcionalidad de las sombras, para poder editar correctamente esta los parámetros que afectan a las sombras se creará una nueva pestaña en el panel derecho de nuestra interfaz llamada “Mapa de sombras”. Desde esta sección, si el uso de sombras está activo, el usuario podrá modificar todos los parámetros que afectan a las sombras de una escena como son: zNear de la luz, zFar de la luz, fovX de la luz y ancho y alto del mapa de sombras. Estos parámetros necesitan ser ajustados para cada fuente luminosa si se quiere conseguir unas sombras más realistas.

En cuanto a la posibilidad de moverse alrededor del objeto. Se deberá implementar la gestión de entradas de ratón por parte del usuario en la ventana encargada de visualizar la escena 3D. Además, se implementará el movimiento de

cámara denominado “orbit” tanto en vertical como en horizontal, permitiendo al usuario rotar alrededor del objeto haciendo uso del ratón mediante el click y arrastre.

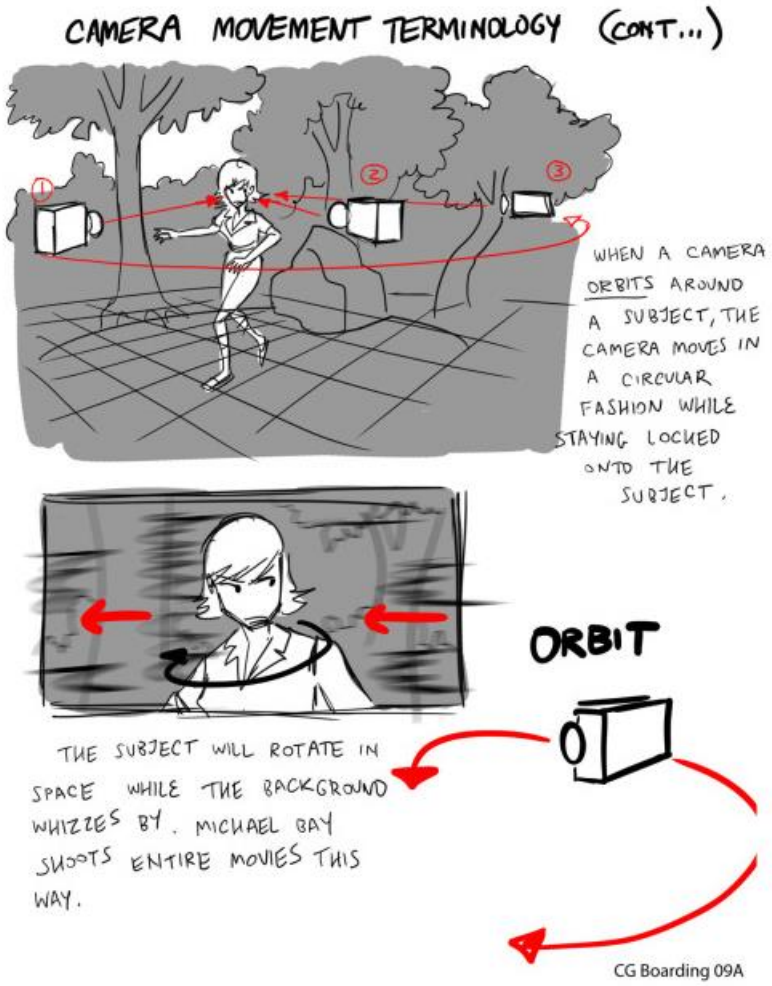


Ilustración 1.1 Movimiento Orbit

Por último, para los campos que sea editables por parte del usuario, se implementará una funcionalidad de feeback visual que permitirá saber en todo momento al usuario si los parámetros introducidos son correctos y, por lo tanto, se están aplicando los cambios a la escena, o, si los valores introducidos son incorrectos y, por lo tanto, la escena no se ha modificado.

Para el caso en el cual los valores sean correctos, el borde del campo en el cual se han introducido los parámetros se pondrá en verde.

Para el caso contrario, el borde se pondrá en rojo.

- Diseño del sistema:

Una vez realizado el análisis de nuestro sistema, voy a proceder a especificar el diseño de las clases que formarán el mismo.

La interfaz de usuario será creada y gestionada por una clase llamada “mainwindow”. Esta clase heredará de “QWidget”, que consiste en una clase que nos proporciona Qt para la utilización de elementos gráficos para nuestra interfaz y su procesamiento. Por lo tanto, esta clase se encargará de inicializar todos los elementos de la interfaz y de procesar todos los datos de entrada por parte del usuario, a excepción de las entradas realizadas sobre la ventana de visualización.

La ventana de visualización, a la que también llamaremos ventana de OpenGL, estará definida por la clase “glwindow”. La clase “glwindow” consistirá en un singleton que heredará de QOpenGLWidget y QOpenGLFunctions_4_1_Core. Ambas clases son proporcionadas por Qt para poder crear una ventana de visualización compatible con OpenGL y poder utilizar las funciones que proporciona OpenGL para trabajar sobre ella. La instancia de “glwindow” será inicializada por “mainwindow” ya que formará parte de la interfaz de usuario, pero, los eventos de ratón que se ejecuten sobre esta ventana, serán procesados por la propia clase “glwindow”.

Para la gestión de la ventana de visualización tendremos una clase llamada “renderer” que también actuará como un singleton. Esta clase será la encargada de dibujar la escena 3D y, por lo tanto, “mainwindow” deberá de comunicarle todos los cambios que el usuario haga en la interfaz para que puedan ser aplicados a la escena. Como ha de poder realizar modificaciones sobre la ventana de OpenGL, esta clase heredará de QOpenGLFunctions4_1_Core para poder usar las funciones de OpenGL para el dibujado de la escena.

Nuestra clase principal, “main”, será la encargada de crear una instancia de “mainwindow”, la cual generará la interfaz de usuario y lanzará la ventana de visualización. Además, se encargará de inicializar la instancia de la clase “renderer” y prepararla para dibujar sobre la ventana de OpenGL.

Para poder visualizar una escena 3D hacen falta varios elementos. Necesitamos una cámara que defina el punto de vista del usuario, los elementos que queremos visualizar en la escena y, un mecanismo de comunicación con la GPU que le pase la información necesaria para dibujar la escena, un shader program.

En primer lugar, la cámara será gestionada por una clase llamada “camera” que almacenará todos los atributos y métodos necesarios para poder definir una cámara en una escena 3D. Nuestra clase “renderer” será la encargada de almacenar esta cámara y pasar sus atributos necesarios a la GPU.

En segundo lugar, los elementos que se dibujarán serán dos, un objeto de revolución y un plano sobre el que estará situado el objeto. Para gestionar estos objetos partiremos de una clase llamada “element3d”. Esta clase será la encargada de definir los atributos y métodos necesarios para dibujar un objeto cualquiera en una escena 3D.

Nuestro objeto de revolución será gestionado por la clase “revolutionobject” la cual heredará de element3d. Esta clase almacenará la información necesaria para poder dibujar un objeto de revolución en una escena 3D y será la encargada de pasarle a la GPU toda esta información cuando “renderer” se lo pida. Como ha de establecer comunicación con la GPU, la clase “revolutionobject” también deberá heredar de QOpenGLFunctions_4_1_Core para poder hacer uso de las funciones de OpenGL.

Para pasar la información al shader, la clase “revolutionobject” hará uso de la clase “vao”.

Para la creación de un objeto de revolución es necesario partir de un perfil de subdivisión, Para la gestión de este perfil de subdivisión dispondremos de la clase “subdivisionprofile” la cual se encargará de almacenar el perfil de subdivisón de un objeto de revolución y de realizar las subdivisiones necesarias en el mismo.

El plano sobre el que estará situado el objeto de revolución será gestionado por la clase “plane”. Esta clase, al igual que “revolutionobject”, heredará tanto de “element3d” como de QOpenGLFunctions_4_1_Core y almacenará la información necesaria para poder dibujar un plano en una escena 3D. Toda esta información deberá de pasársela a la GPU cuando “renderer” se lo solicite.

Para pasar la información al shader, la clase “plane” hará uso de la clase “vao”.

Tanto el plano como el objeto de revolución llevan asociados una textura que es la encargada de darles su color y su aspecto. Todas las texturas vamos a gestionarlas a partir de la clase “texture” y vamos a almacenaras en una clase llamada “bibliotexture” que funcionará como un singleton para tener una biblioteca de texturas. De esta forma, mediante la clase “texture” podremos cargar una imagen y generar una

textura de ella para después almacenarla en la instancia de “bibliotexture” y tener acceso a ella desde cualquier módulo de nuestra aplicación. Las texturas también son necesarias pasarlas a la GPU, por lo tanto, nuestra clase “texture” heredará de QOpenGLFuncions_4_1_Core para poder comunicarse con la misma.

Todos los elementos que vayan a ser dibujados en la escena, serán almacenados por nuestra clase “renderer” ya que es la encargada del dibujado de la misma.

Toda la información sobre el dibujado de una malla de triángulos va almacenada en un VAO, por lo tanto, vamos a encapsular toda la información de un VAO en una clase denominada “vao”.

Por último, para poder pasarle toda la información a la GPU, necesitaremos un shader program. Para gestionar la creación, el enlazado y el paso de uniforms de un shader program crearemos la clase “shaderprogram”. Esta clase se encargará de crear un shader program a partir de un vertex shader y de un fragment shader que le pasemos, activarlo cuando sea necesario y pasarle a cada shader object los parámetros necesarios para su uso.

Finalmente nos queda la iluminación de la escena. Para gestionar todas las luces de la escena crearemos una clase denominada “lightsource”. Esta clase almacenará toda la información necesaria que puede necesitar una luz para dibujarse en la escena. Por lo tanto, esta clase almacenará los atributos necesarios de una luz ambiental, spot, direccional y puntual. Pero, cada luz sólo necesita de un conjunto de atributos para aplicarse en la escena, por lo tanto, cada luz solo necesita pasar una serie de atributos a la GPU, no todos. Para solucionar esto, a cada luz le asignaremos un “lightapplicator”.

La clase “lightapplicator” consistirá en una interfaz de la cuál heredarán las clases “lightapplicatorambiental”, “lightapplicatorspot”, “lightapplicatorpuntual”, “lightapplicatordireccional”. Cada una de estas clases se encargará de pasar a la GPU únicamente los parámetros necesarios para dibujar la luz a la que hacen referencia. De esta forma, cuando “renderer” le diga a “lightsource” que pase sus parámetros a la GPU, la clase “lightsource” avisará a “lightapplicator” para que pase los parámetros necesarios.

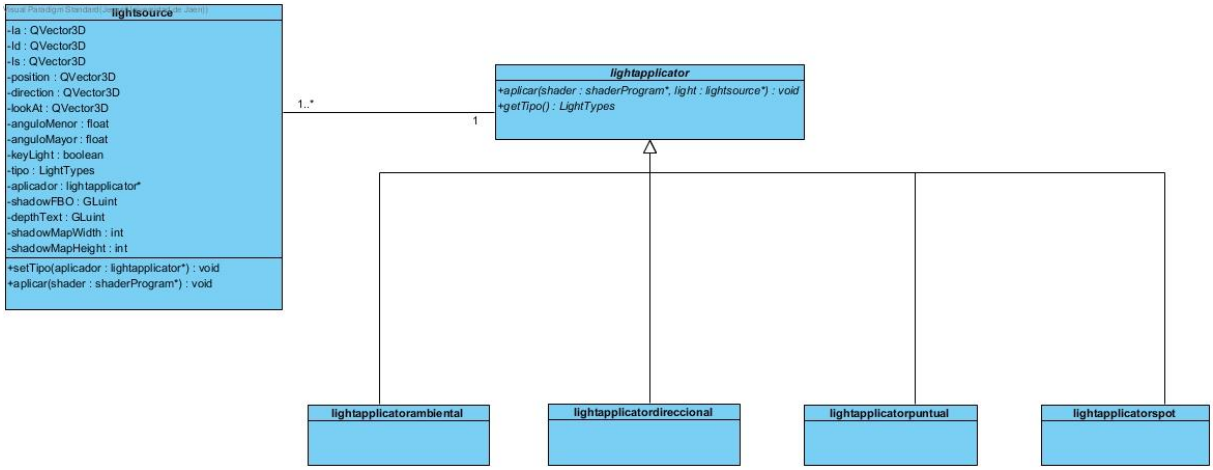


Ilustración 1.2 Diagrama lightsource-lightapplicator

Además, la clase “lightsource” será la encargada de gestionar los mapas de sombras, por lo tanto, necesitará heredar de QOpenGlFunctions_4_1_Core para crear el FBO en el cual se registrarán los valores de profundidad.

Todas las luces de la escena serán almacenadas por “renderer” para que, cuando se vaya a pintar la escena, “renderer” realice la llamada correspondiente a cada luz para que pasen sus parámetros a la GPU.

1.6.1 - Metodología de desarrollo

La metodología de desarrollo seguida para este proyecto ha sido la metodología tradicional de análisis-diseño-implementación-pruebas.

He decidido utilizar esta metodología porque tenía claro los requisitos que debería de cumplir mi proyecto y, por lo tanto, podría tener todos los requisitos definidos desde el primer momento y llevar a cabo esta metodología en cascada hasta conseguir el despliegue de la aplicación.

1.7 - A3: Estimación del tamaño y esfuerzo

Ya que el presente proyecto es un TFT, no existen restricciones de tipo económico, sino de tipo temporal (un número aproximado de horas). Por consiguiente, los cálculos de tamaño del proyecto están supeditados el tiempo disponible. En cuanto al esfuerzo, se dispone de tan un solo efectivo (el autor).

2 - DESARROLLO DEL PROYECTO

2.1 - Tecnologías utilizadas

Las tecnologías que se han utilizado en este proyecto han sido:

1. Lenguajes:
    - C++: Consiste en un lenguaje de programación multiparadigma a alto nivel.
    - Glsl: Es un lenguaje basado en C utilizado en la creación de shaders para OpenGL.
2. Librerías:
    - OpenGL: Es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.
    - Qt: Consiste en un framework multiplataforma orientado a objetos ampliamente usado para diseñar programas que utilicen interfaz gráfica de usuario.
3. Software:
    - Git: Es un sistema de control de versiones gratuito y open-source diseñado para manejar proyectos de una manera rápida y eficiente.
    - Sourcetree: Consiste en un cliente gratuito de git para Windows.
4. Servicios web:
    - Github: Es una plataforma de colaboración y control de versiones basada en web para desarrolladores de software.

2.2 - Diseño

2.2.1 - Diseño arquitectónico del sistema

Debido a la naturaleza del TFG, es necesario hacer uso de alguna librería gráfica que nos permita renderizar objetos. Es por ello por lo que se utiliza OpenGL, más concretamente, OpenGL 4.1 Core Profile.

La librería de OpenGL nos permite mandar información al a GPU sobre renderizado de geometría, permitiéndonos crear nuestros propios modelos 3D para visualizarlos en una ventana. Además, no solo permite renderizar una escena con una geometría, sino que nos permite realizar cálculos de iluminación para el renderizado de dicha escena, lo cual es primordial en este proyecto.

Aparte, para poder gestionar todo esto dentro de un programa, se hace uso de la librería de Qt, la cual proporciona múltiples clases que funcionan con OpenGL para conseguir simplificar el uso de dicha librería e integrarla dentro de una interfaz de usuario.

2.2.2 - Diagramas de clases

Debido a su tamaño, el diagrama de clases no puede apreciarse bien, pero puede ser descargado como imagen dando click derecho y “Guardar como imagen”

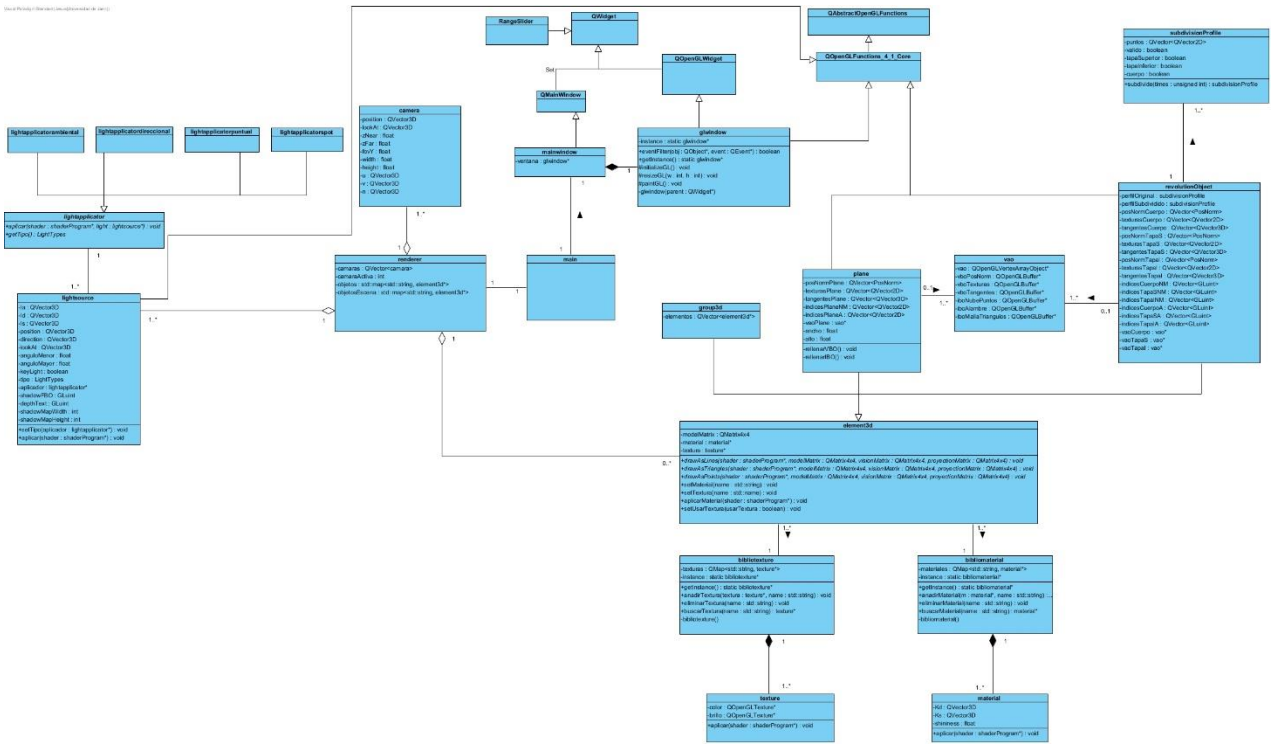


Ilustración 2.1 Diagrama de clase

2.2.3 - Diagramas de casos de uso

Caso de uso	Añadir luz
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	
Operaciones básicas	
1	Elegir tipo de luz
2	Elegir la configuración deseada
3	Hacer click en el botón “Añadir luz”
Alternativas	
1.A	¿Hay alguna luz elegida?
1.A.1	Si sí, continuar
1.A.2	Si no, volver a repetir paso 1
2.A	¿Está toda la configuración correcta?
2.A.1	Si sí, continuar
2.A.2	Si no, corregir las configuraciones no válidas

Tabla 2.1 Casos de uso: Añadir luz

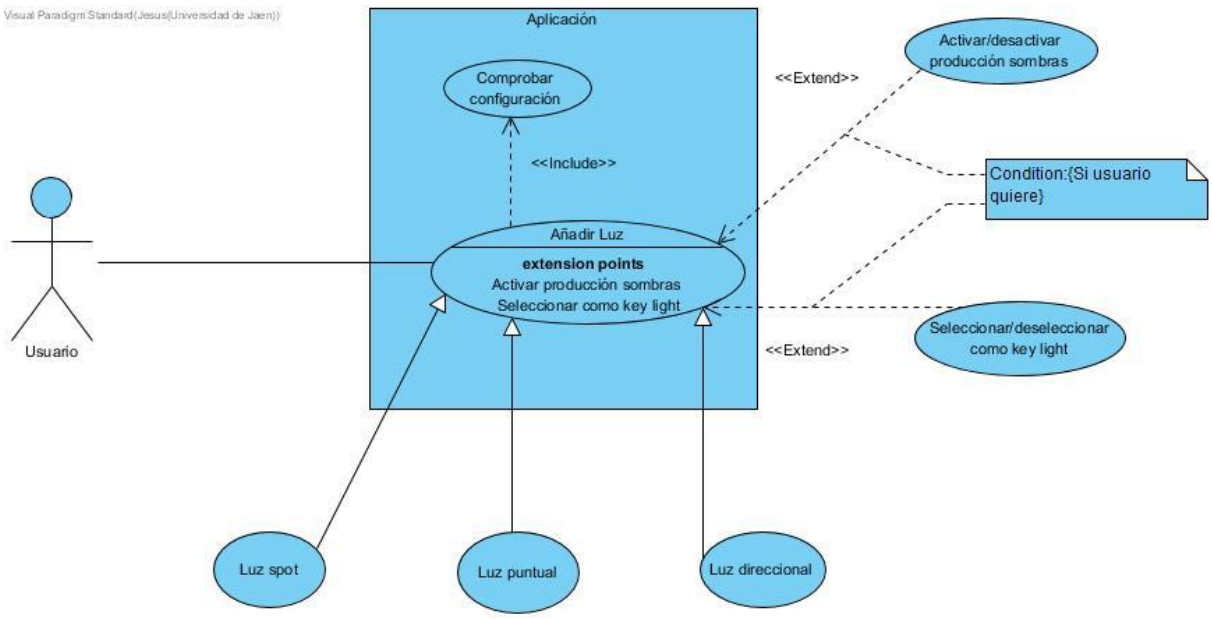


Ilustración 2.2 Casos de uso: Añadir luz

Caso de uso	Eliminar luz
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	Hay luces en la escena
Operaciones básicas	
1	Seleccionar la luz que se quiere eliminar
2	Hacer click en el botón “Eliminar luz”

Tabla 2.2 Casos de uso: Eliminar luz

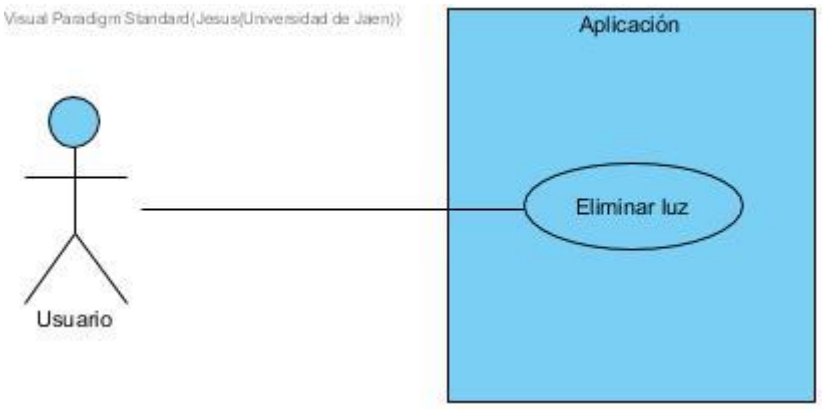


Ilustración 2.3 Casos de uso: Eliminar luz

Caso de uso	Editar luz
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	Hay luces en la escena
Operaciones básicas	
1	Seleccionar la luz que se quiera editar
2	Modificar su configuración
Alternativas	
2.A	¿La configuración editada es la correcta?
2.A.1	Si sí, continuar
2.A.2	Si no, volver al paso 2

Tabla 2.3 Casos de uso: Editar luz

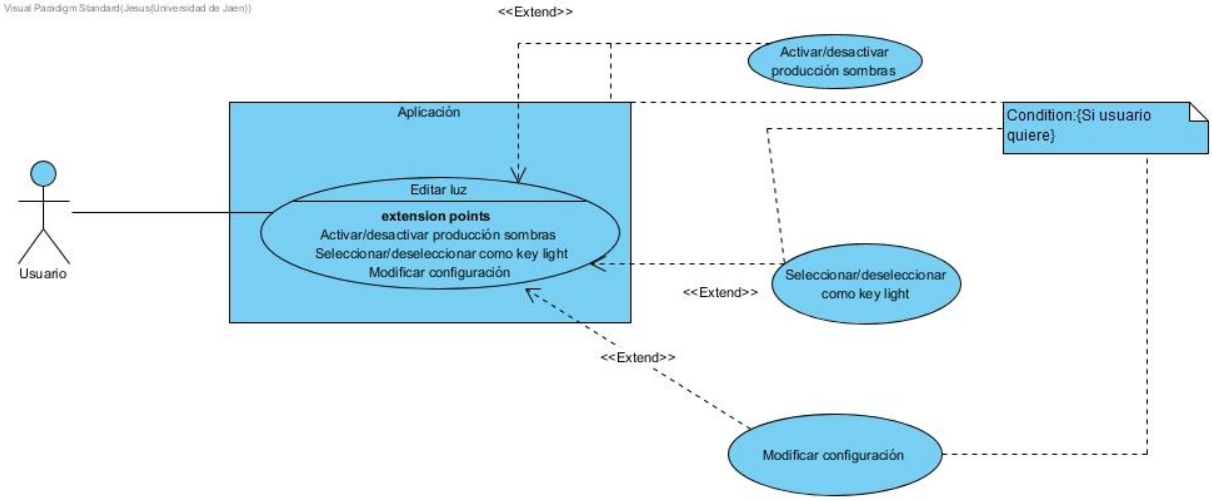


Ilustración 3.4 Casos de uso: Editar luz

Caso de uso	Cambiar objeto escena
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	
Operaciones básicas	
1	Desplegar el menú “Archivo”
2	Desplegar el submenú “Objetos”
3	Elegir objeto

Tabla 2.4 Casos de uso: Cambiar objeto escena

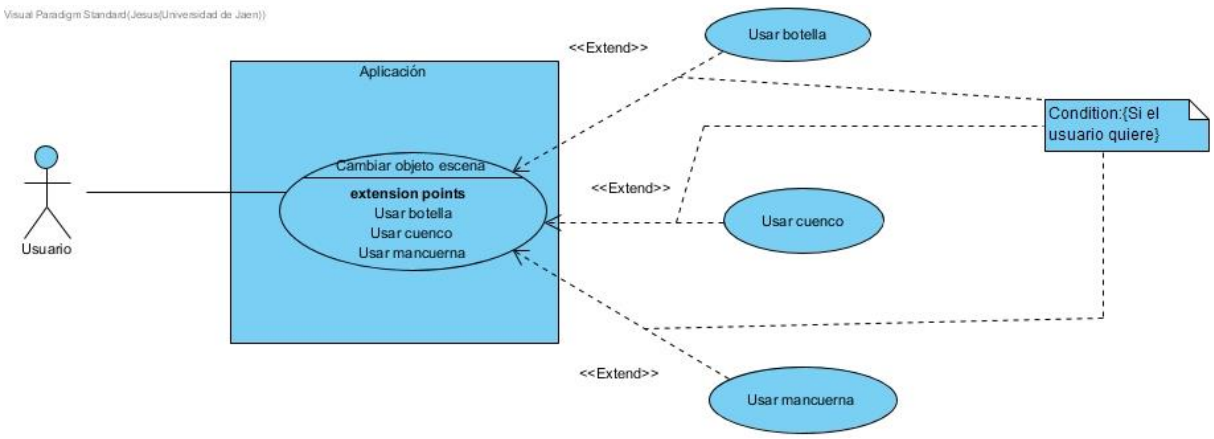


Ilustración 2.5 Casos de uso: Cambiar objeto escena

Caso de uso	Guardar configuración
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	
Operaciones básicas	
1	Desplegar el menú “Archivo”
2	Hacer click en la opción “Guardar configuración”
3	Elegir ruta del archivo
4	Elegir nombre del archivo
5	Hacer click en “Guardar”
Alternativas	
4.A	¿El nombre del archivo ya existe?
4.A.1	Si sí, ¿se desea sobrescribirlo?
4.A.1.1	Si sí, continuar
4.A.1.2	Si no, volver al paso 4
4.A.2	Si no, continuar

Tabla 2.5 Casos de uso: Guardar configuración

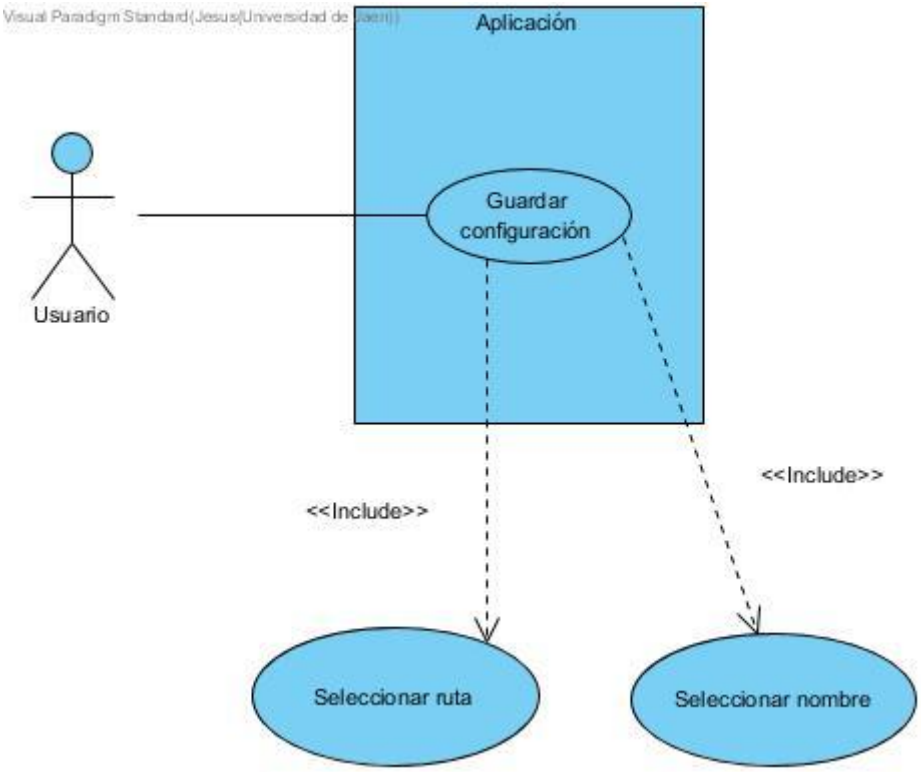


Ilustración 2.6 Casos de uso: Guardar configuración

Caso de uso	Cargar configuración
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	Haber guardado una configuración
Operaciones básicas	
1	Desplegar el menú “Archivo”
2	Hacer click en la opción “Cargar configuración”
3	Elegir el archivo
4	Hacer click en “Abrir”
Alternativas	
3.A	¿Tiene el archivo la extensión correcta?
3.A.1	Si sí, continuar
3.A.2	Si no, cancelar

Tabla 2.6 Casos de uso: Cargar configuración

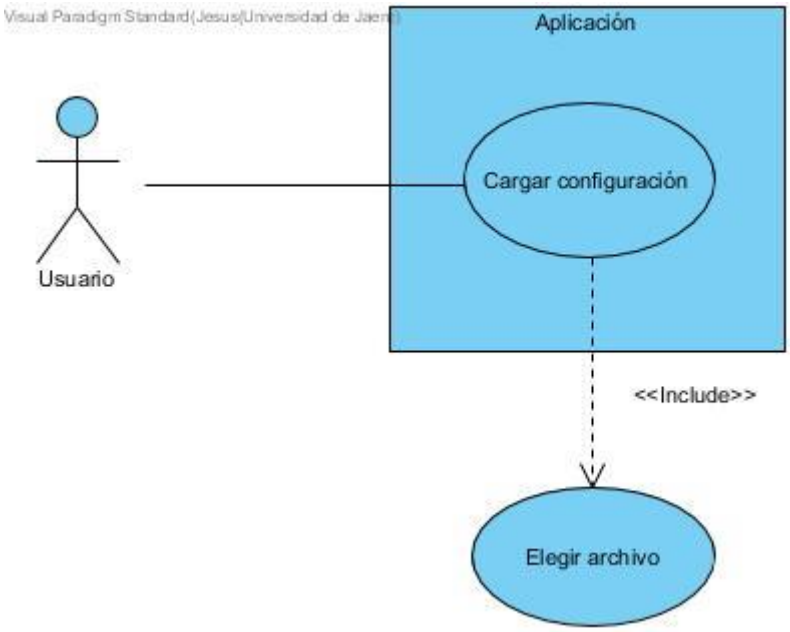


Ilustración 2.7 Casos de uso: Guardar configuración

Caso de uso	Activar/desactivar atenuación
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	
Operaciones básicas	
1	Desplegar el menú “Herramientas”
2	Hacer click en la opción “Atenuación”

Tabla 2.7 Casos de uso: Activar/desactivar atenuación

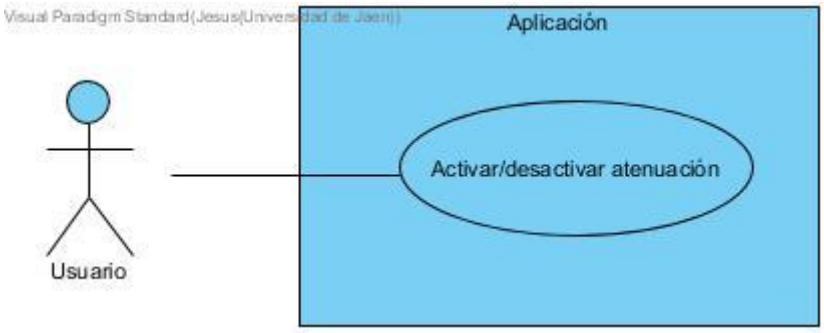


Ilustración 2.8 Casos de uso: Activar/desactivar atenuación

Caso de uso	Modificar atenuación
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	Tener activada la atenuación
Operaciones básicas	
1	Hacer click en la pestaña “Extras”
2	Modificar los coeficientes

Tabla 2.8 Casos de uso: Modificar atenuación

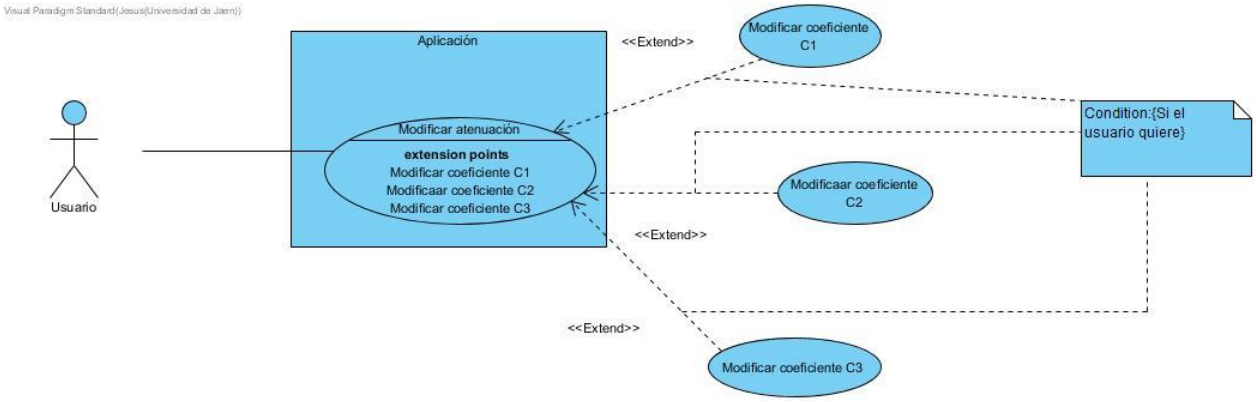


Ilustración 2.9 Casos de uso: Modificar atenuación

Caso de uso	Activar/desactivar FOG
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	
Operaciones básicas	
1	Desplegar el menú “Herramientas”
2	Hacer click en la opción “FOG”

Tabla 2.9 Casos de uso: Activar/desactivar FOG

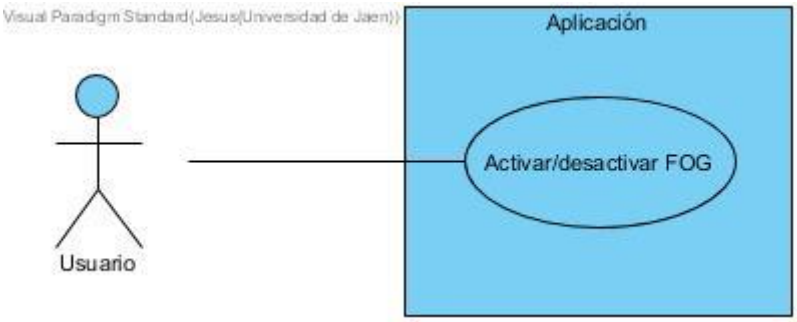


Ilustración 2.10 Casos de uso: Activar/desactivar FOG

Caso de uso	Modificar FOG
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	Tener activado el FOG
Operaciones básicas	
1	Hacer click en la pestaña “Extras”
2	Modificar valores

Tabla 2.10 Casos de uso: Modificar FOG

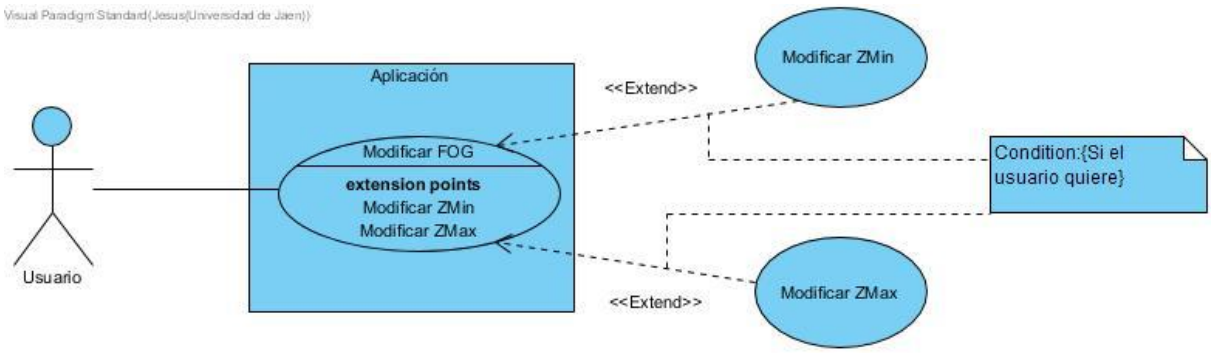


Ilustración 2.11 Casos de uso: Modificar FOG

Caso de uso	Activar/desactivar conjunto de 3 luces
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	
Operaciones básicas	
1	Desplegar el menú “Herramientas”
2	Hacer click en la opción “Conjunto de 3 luces”

Tabla 2.11 Casos de uso: Activar/desactivar conjunto de 3 luces

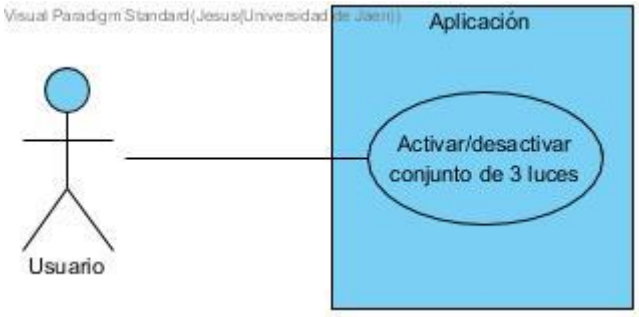


Ilustración 2.12 Activar/desactivar conjunto de 3 luces

Caso de uso	Activar/desactivar sombras
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	
Operaciones básicas	
1	Desplegar menú “Herramientas”
2	Hacer click en la opción “Sombras”

Tabla 2.12 Casos de uso: Activar/desactivar sombras

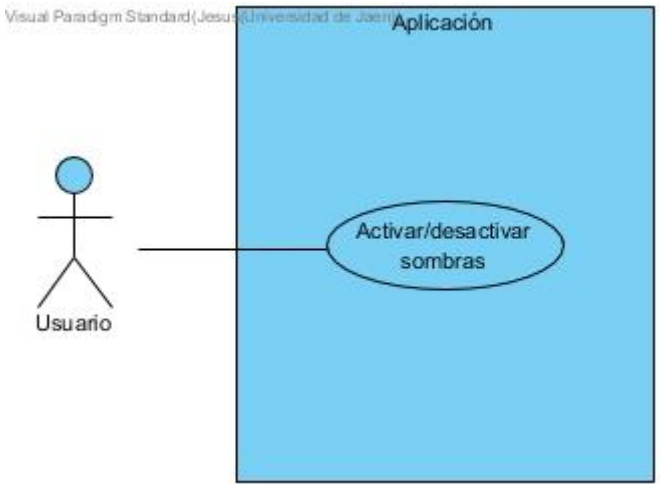


Ilustración 2.13 Casos de uso: Activar/desactivar sombras

Caso de uso	Modificar mapa de sombras
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo Usuario
Condición previa	Hay luces que producen sombras en la escena y la opción “sombras” está activada
Operaciones básicas	
1	Hacer click en la pestaña “Mapa de sombras”
2	Modificar valores
Alternativas	
2.A	¿Los valores modificados son correctos?
2.A.1	Si sí, continuar
2.A.2	Si no, volver al paso 2

Tabla 2.13 Casos de uso: Modificar mapa de sombras

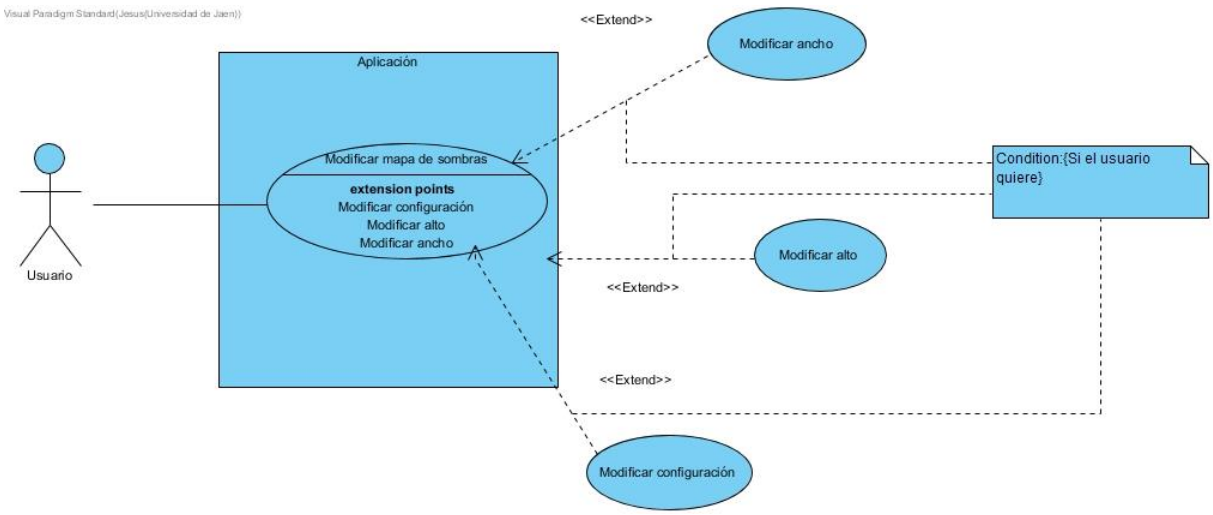


Ilustración 2.14 Casos de uso: Modificar mapa de sombras

Caso de uso	Rotar alrededor del objeto
Actor primario	Usuario
Sistema	Aplicación
Participantes	Usuario
Nivel	Objetivo usuario
Condición previa	
Operaciones básicas	
1	Click y arrastrar en la escena

Tabla 2.14 Casos de uso: Rotar alrededor del objeto

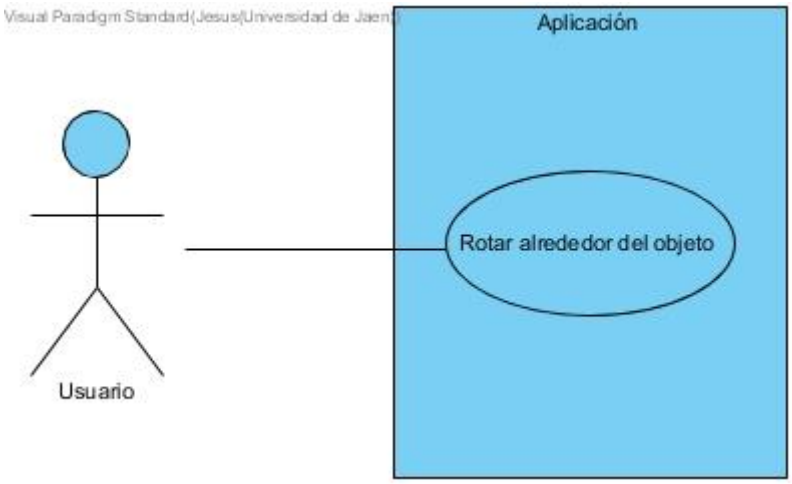


Ilustración 2.15 Casos de uso: Rotar alrededor del objeto

2.2.4 - Diagramas de secuencia

Debido a su tamaño, algunos diagramas no pueden apreciarse bien, pero pueden ser descargados como imagen dando click derecho y “Guardar como imagen”.

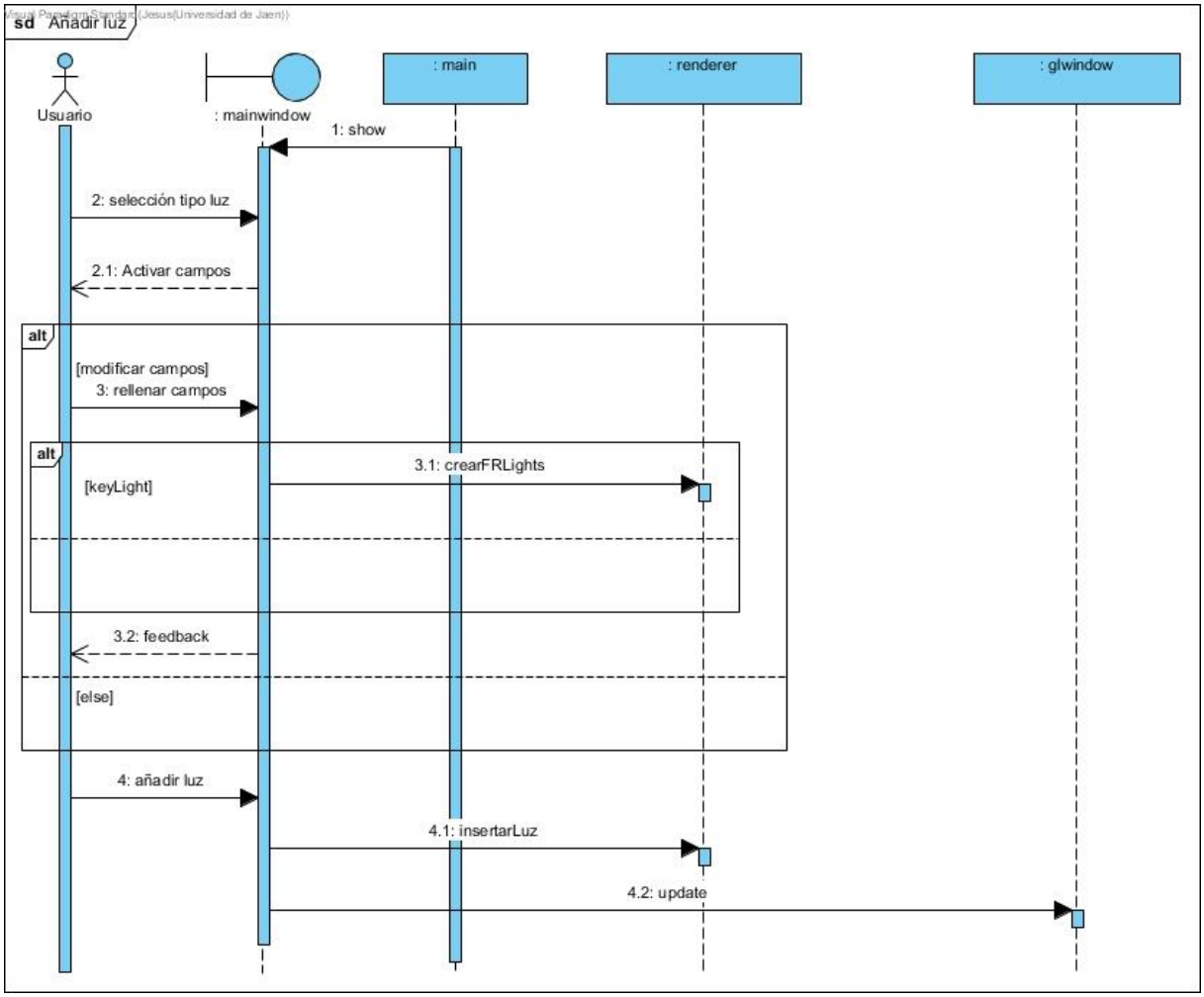


Ilustración 2.16 Diagramas de secuencia: Añadir luz

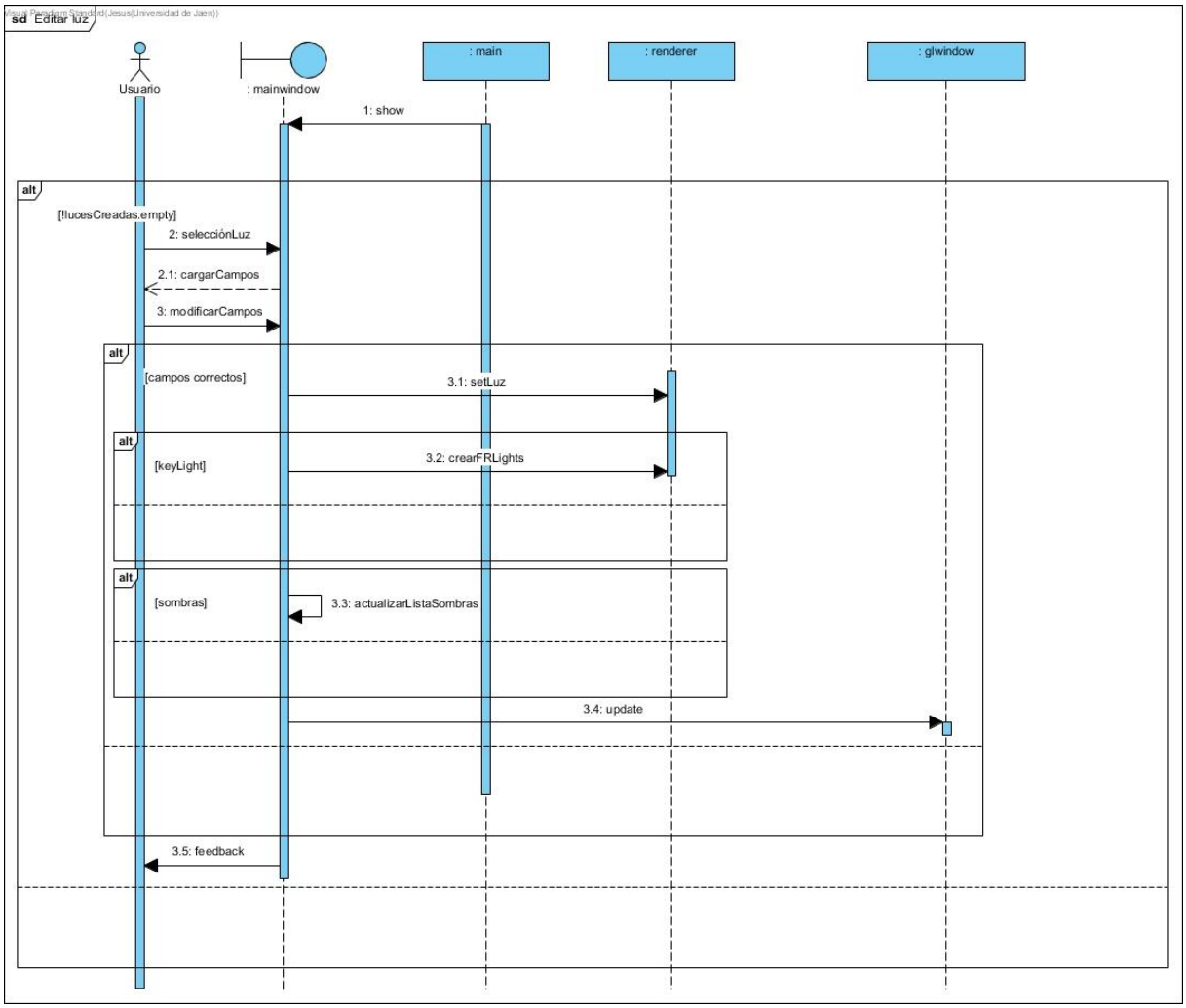


Ilustración 2.17 Diagramas de secuencia: Editar luz

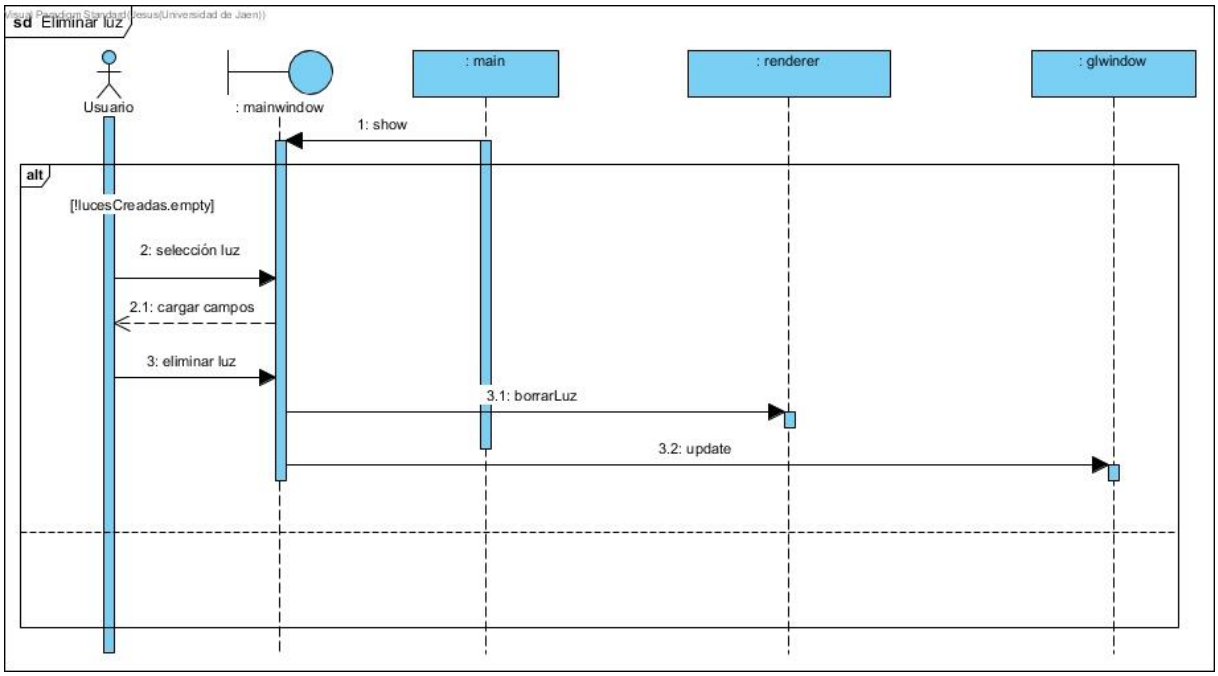


Ilustración 2.18 Diagramas de secuencia: Eliminar luz

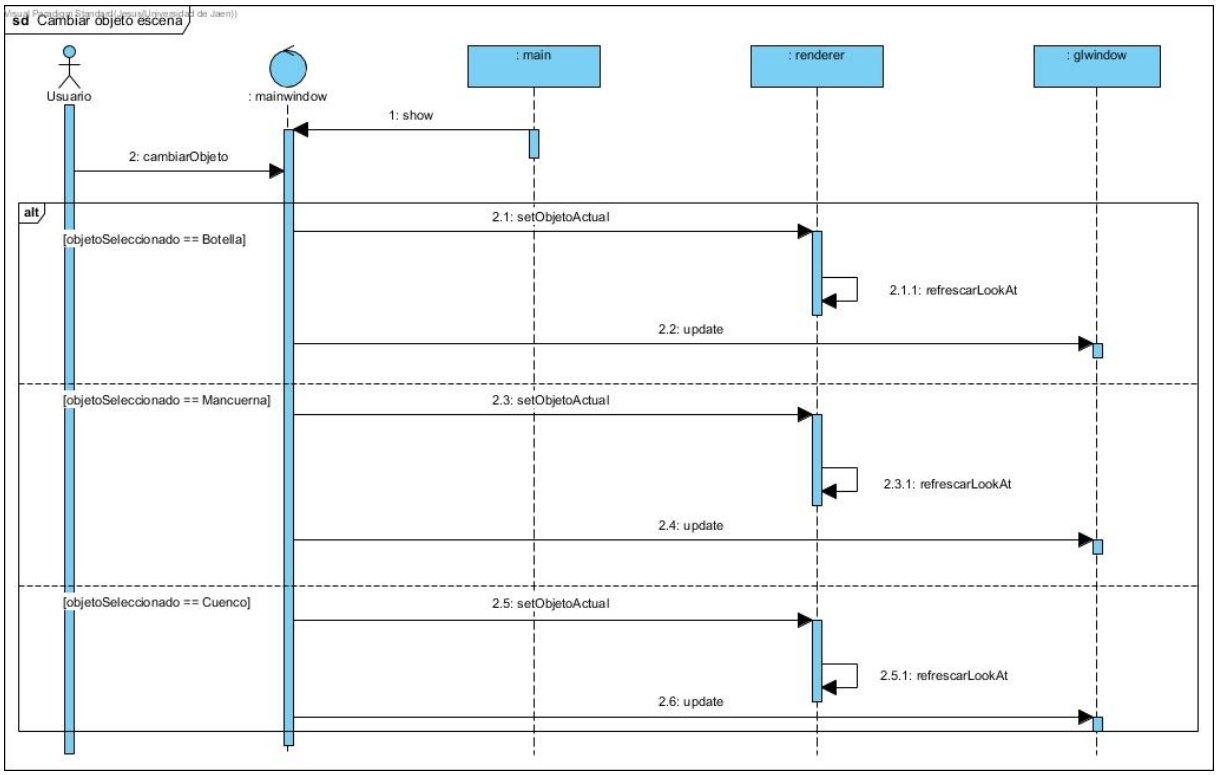


Ilustración 2.19 Diagramas de secuencia: Cambiar objeto escena

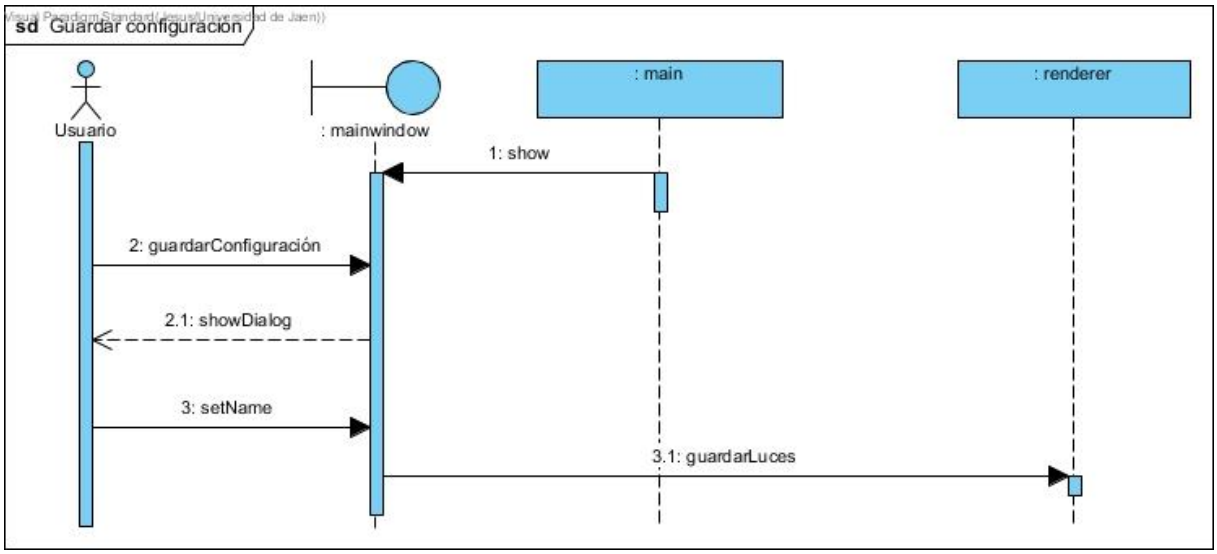


Ilustración 2.20 Diagramas de secuencia: Guardar configuración

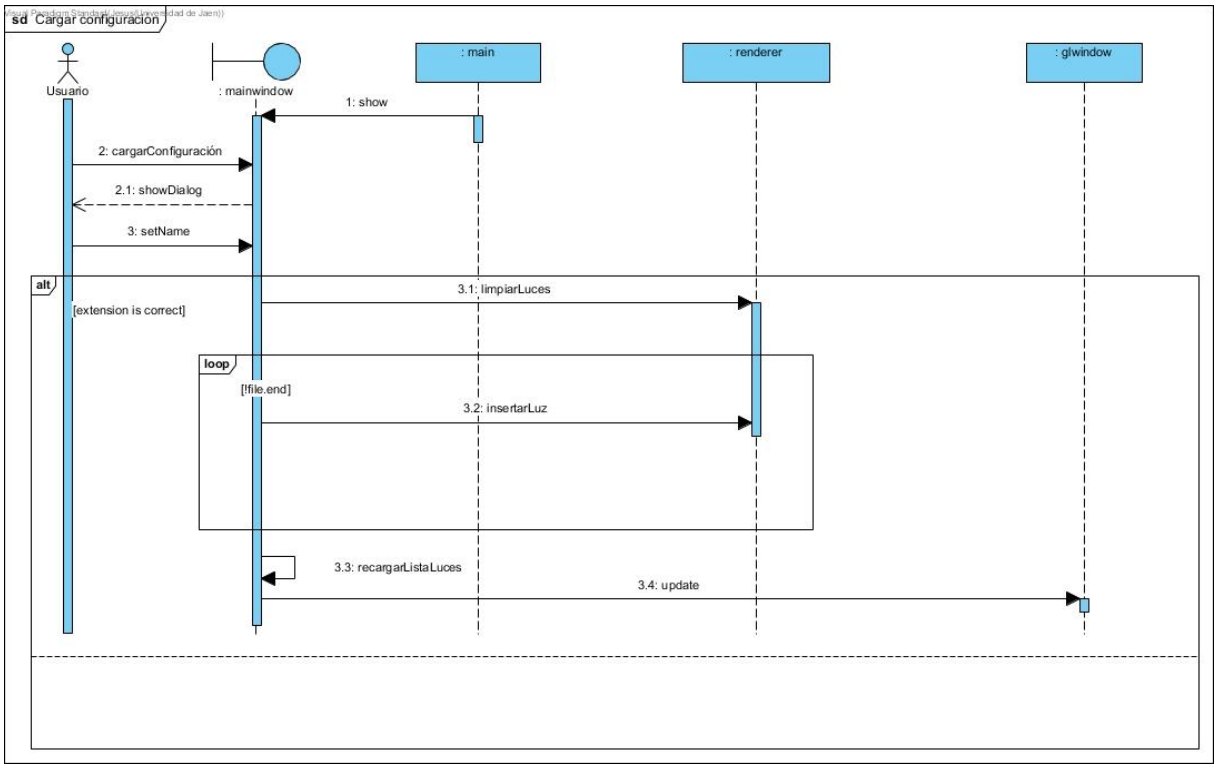


Ilustración 2.21 Diagramas de secuencia: Cargar configuración

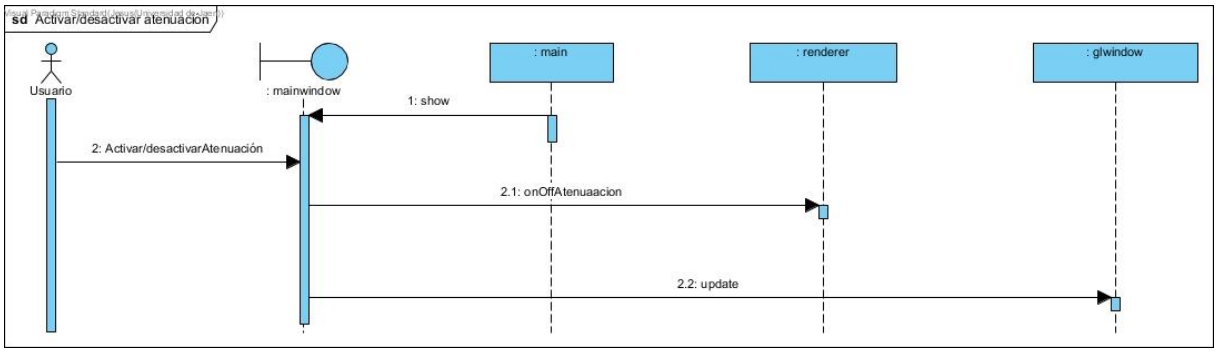


Ilustración 2.22 Diagramas de secuencia: Activar/desactivar atenuación

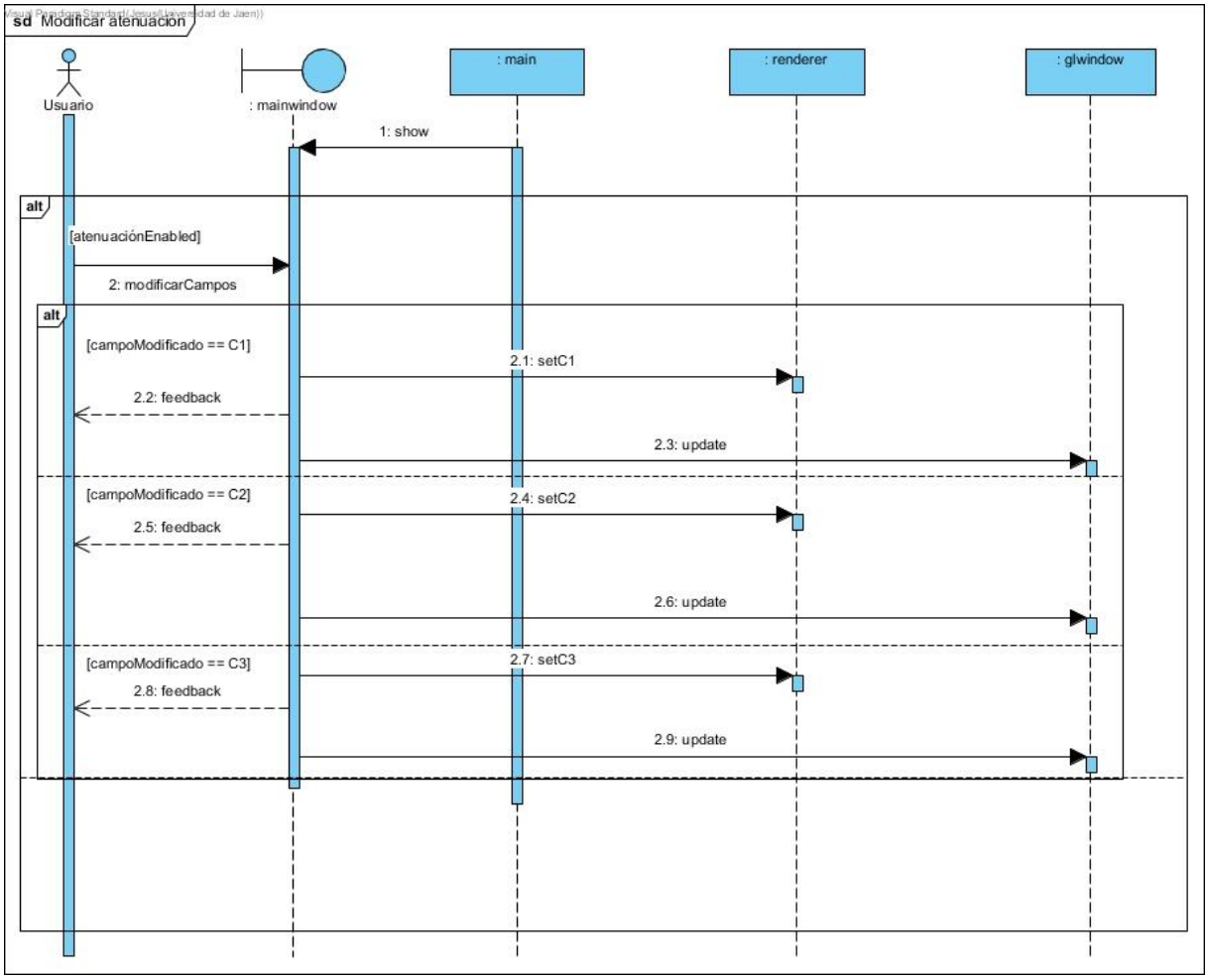


Ilustración 2.23 Diagramas de secuencia: Modificar atenuación

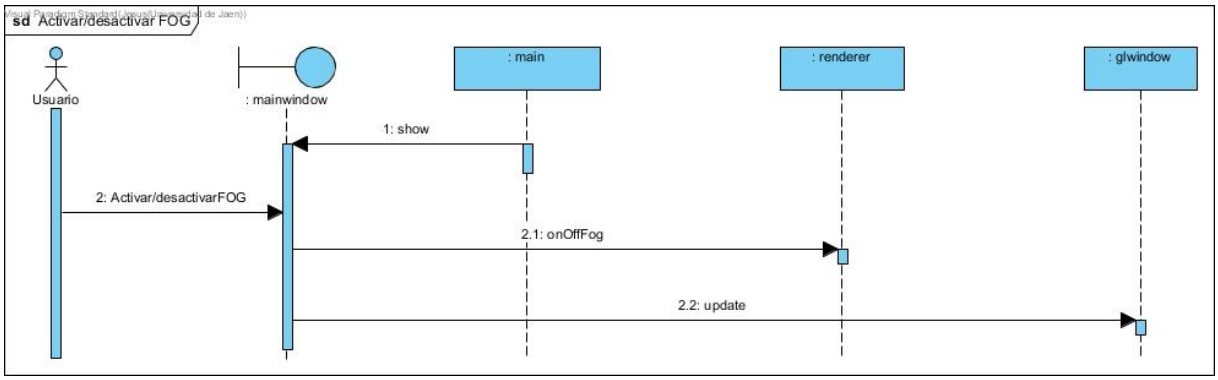


Ilustración 2.24 Diagramas de secuencia: Activar/desactivar FOG

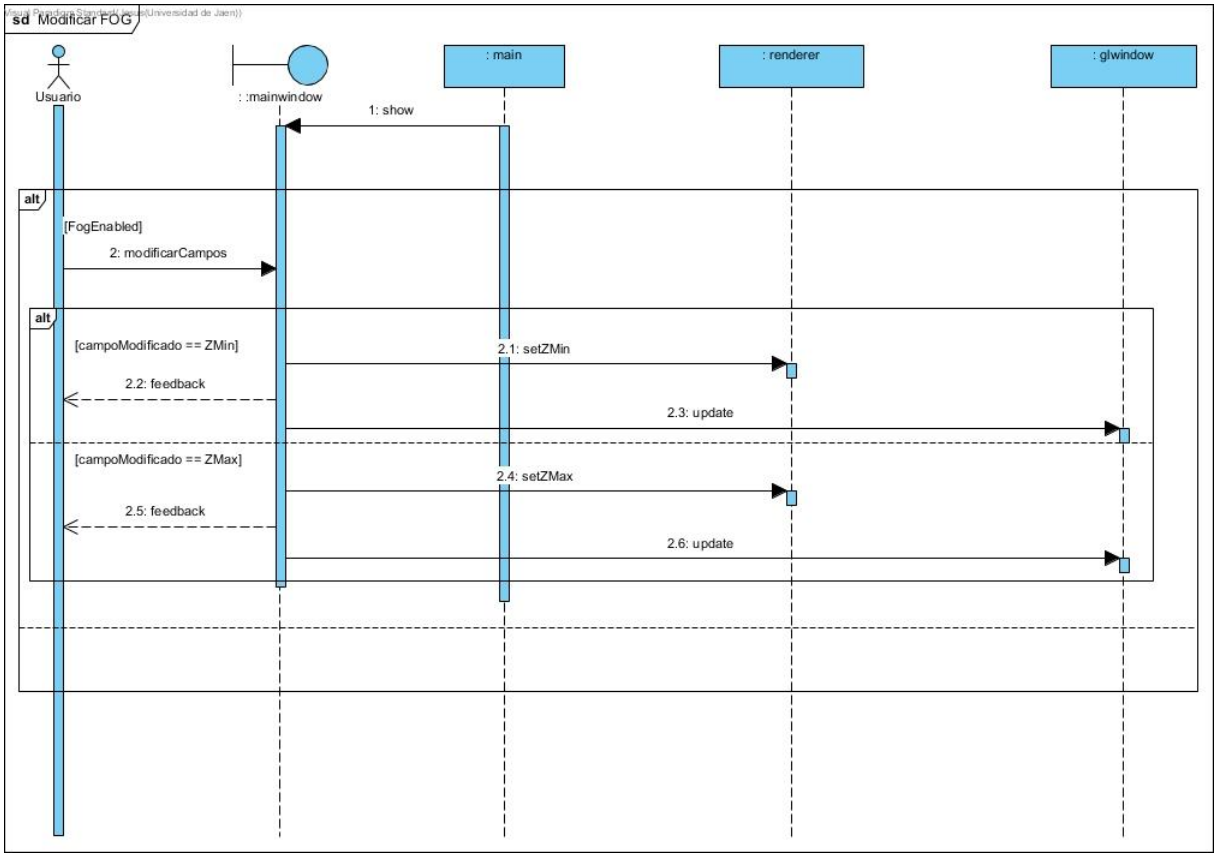


Ilustración 2.25 Diagramas de secuencia: Modificar FOG

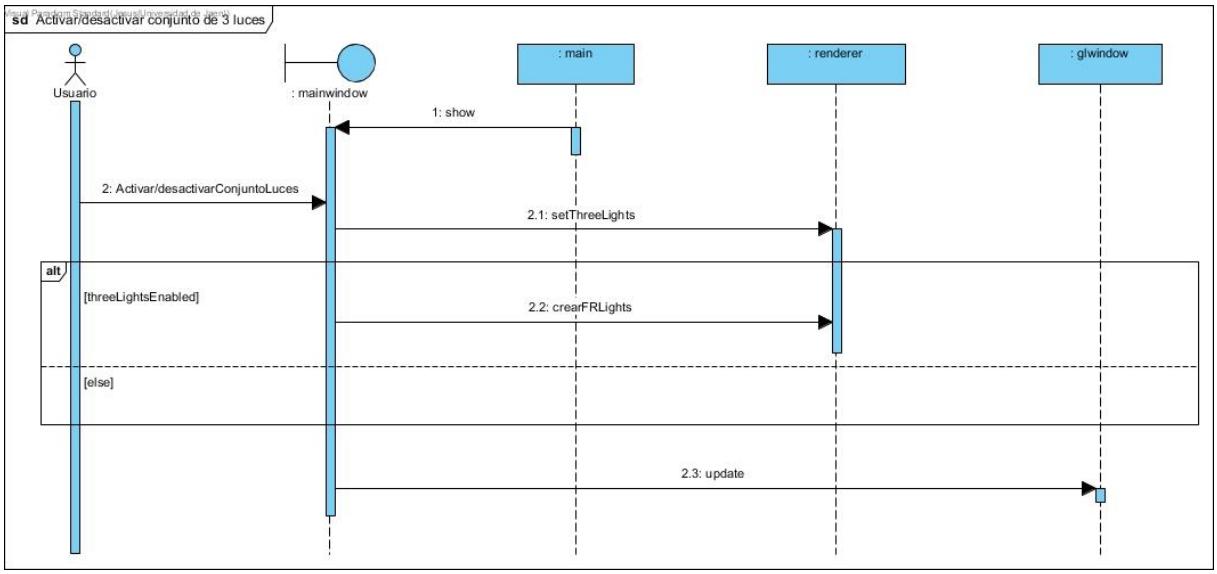


Ilustración 2.26 Diagramas de secuencia: Activar/desactivar conjunto de 3 luces

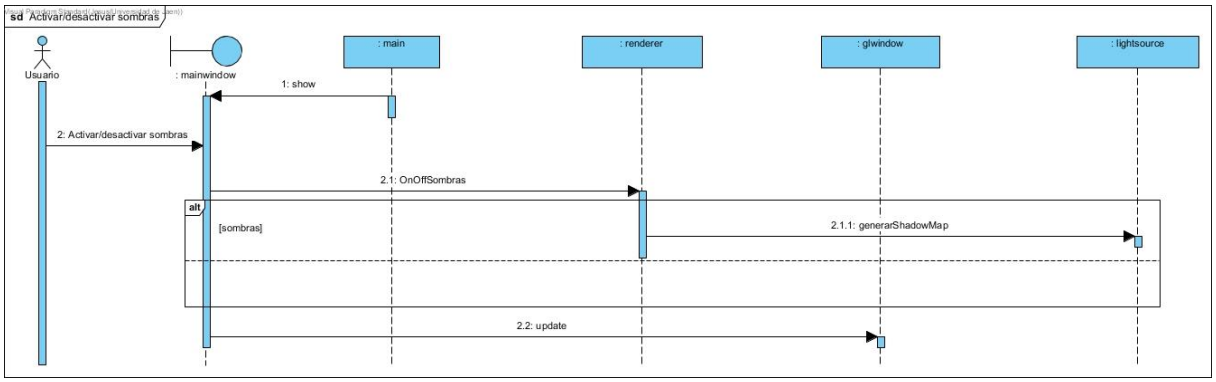


Ilustración 2.27 Diagramas de secuencia: Activar desactivar sombras

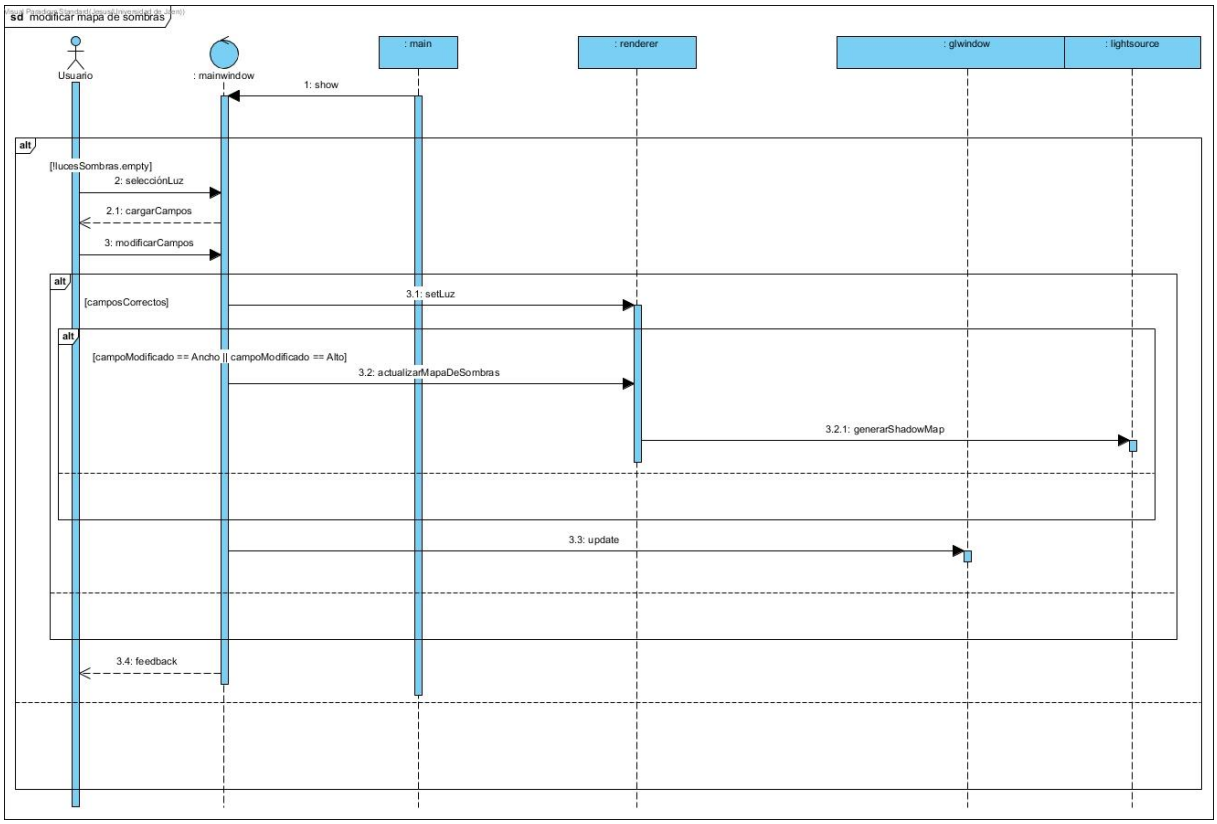


Ilustración 2.28 Diagramas de secuencia: Modificar mapa de sombras

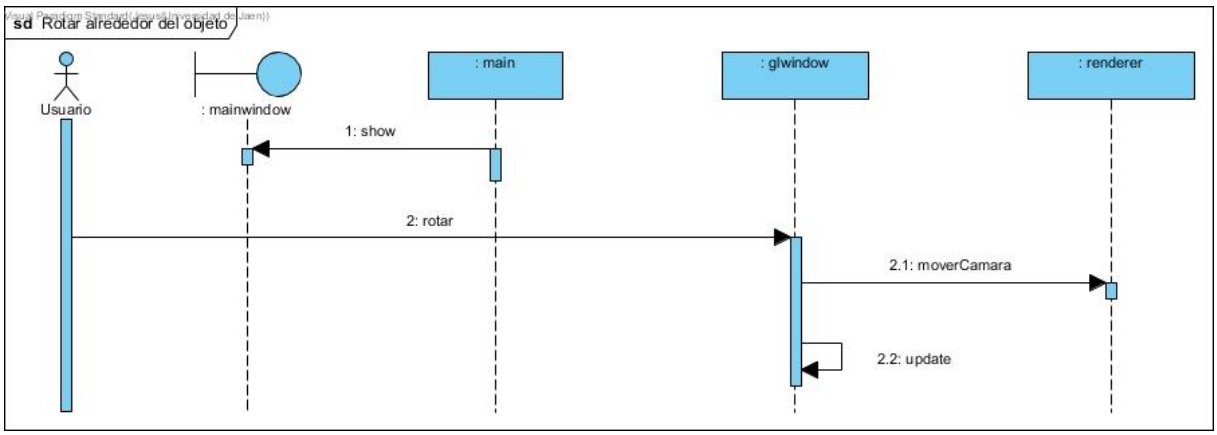


Ilustración 2.29 Diagramas de secuencia: Rotar alrededor del objeto

2.2.5 - Diseño de la interfaz y storyboards

Para este proyecto he decidido realizar varios storyboards agrupando las acciones que puede realizar el usuario. De esta forma queda mucho más claro que debe de hacer el usuario para realizar la tarea deseada.

- Añadir nueva luz:

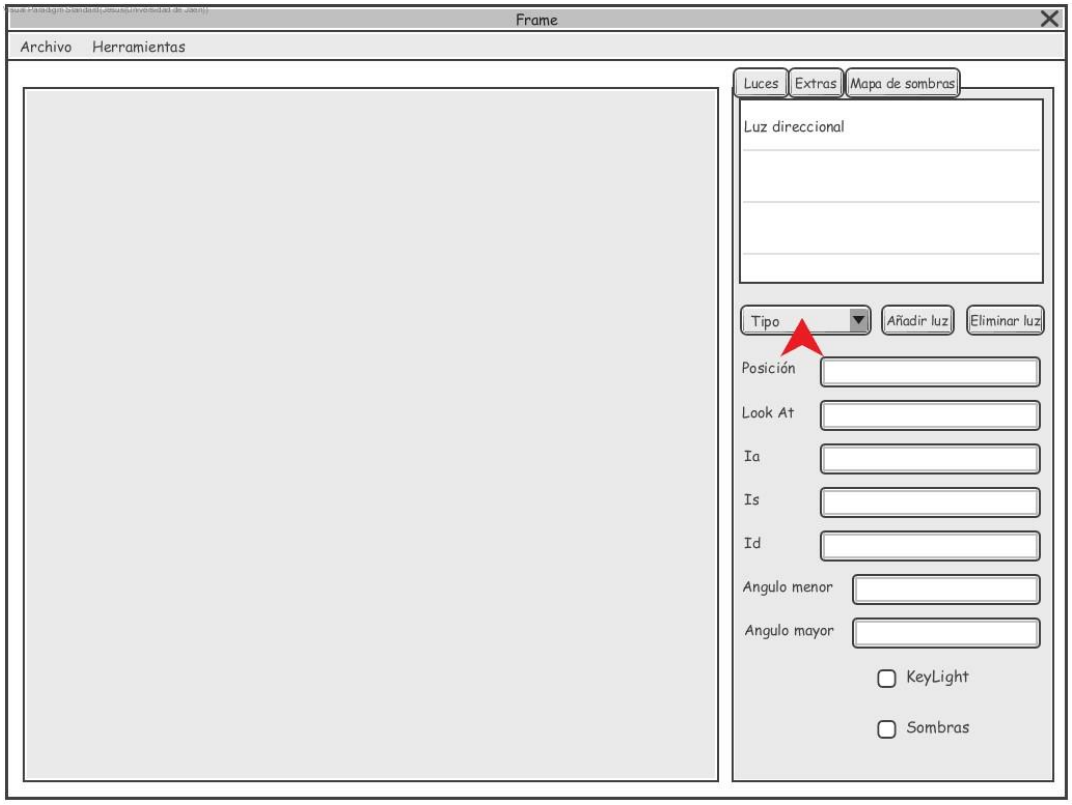


Ilustración 2.30 Storyboards: Añadir luz. Selección de tipo

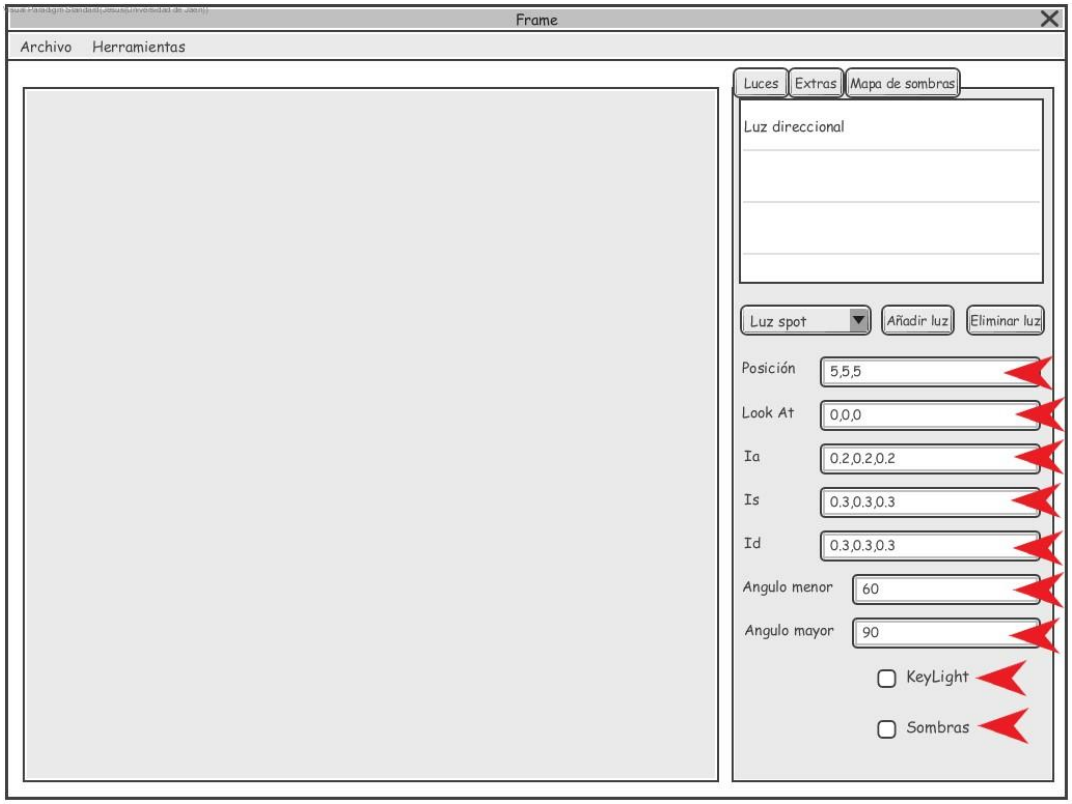


Ilustración 2.31 Storyboards: Añadir luz. Modificación de parámetros

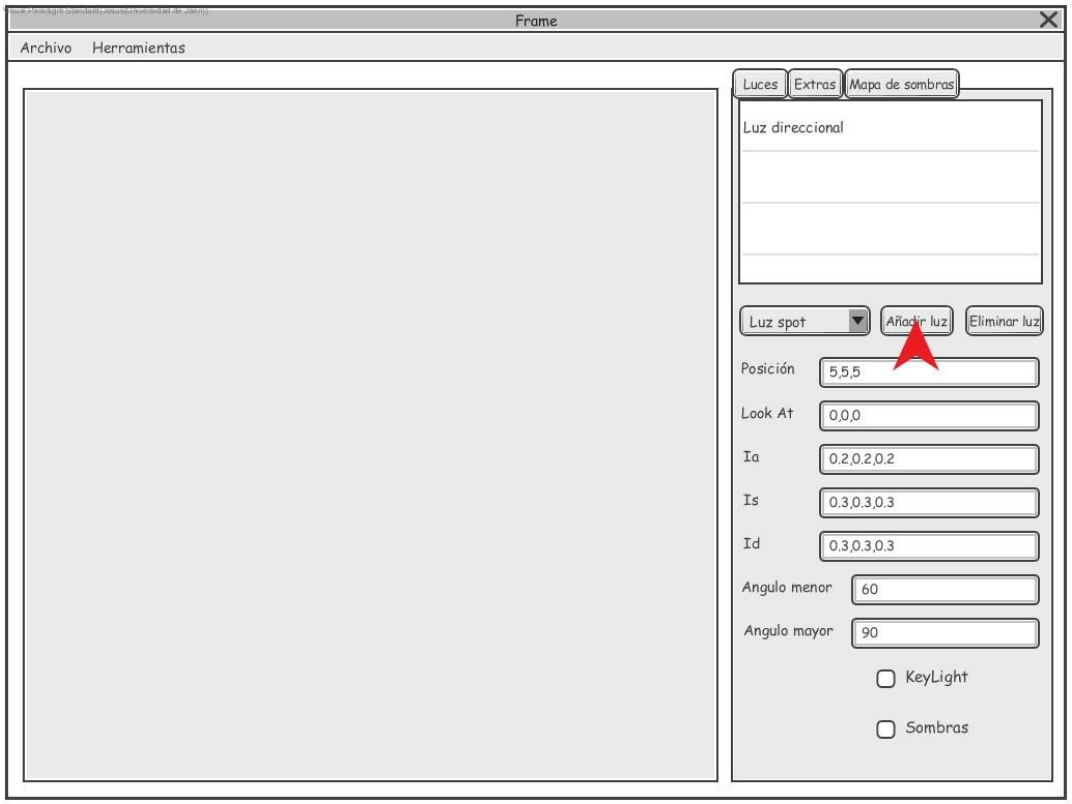


Ilustración 2.32 Storyboards: Añadir luz. Confirmación

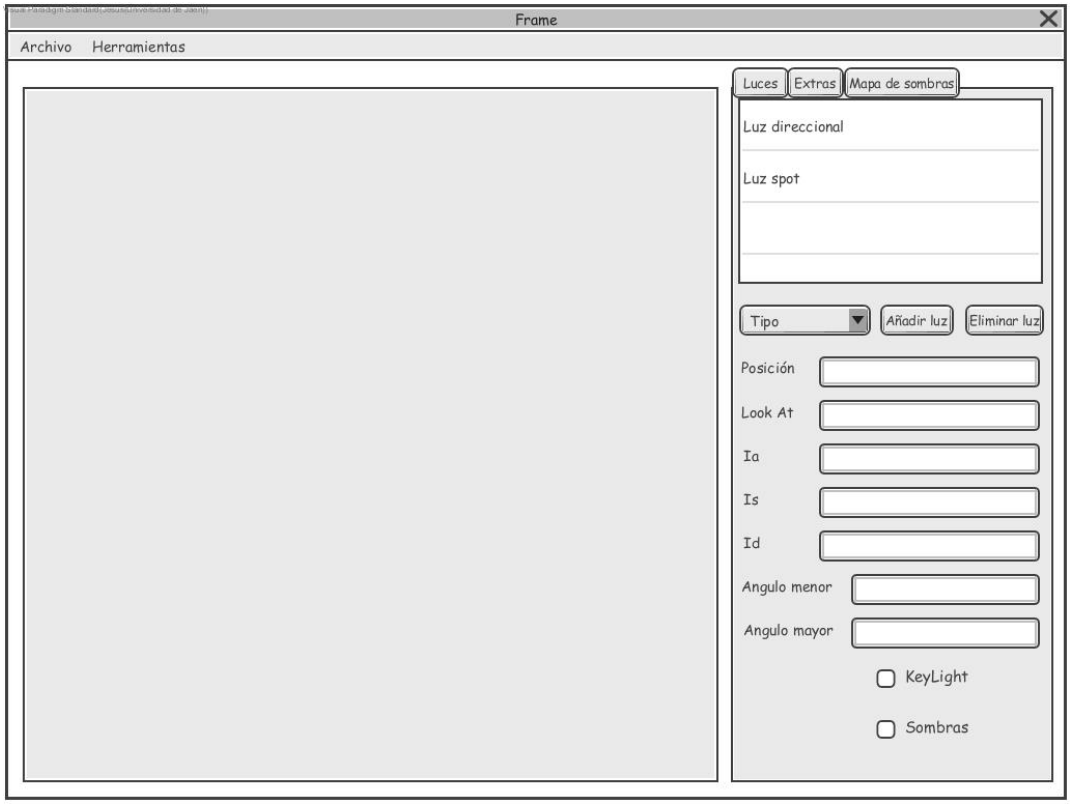


Ilustración 2.33 Storyboards: Añadir luz. Nueva luz en la lista de luces

- Editar luz:

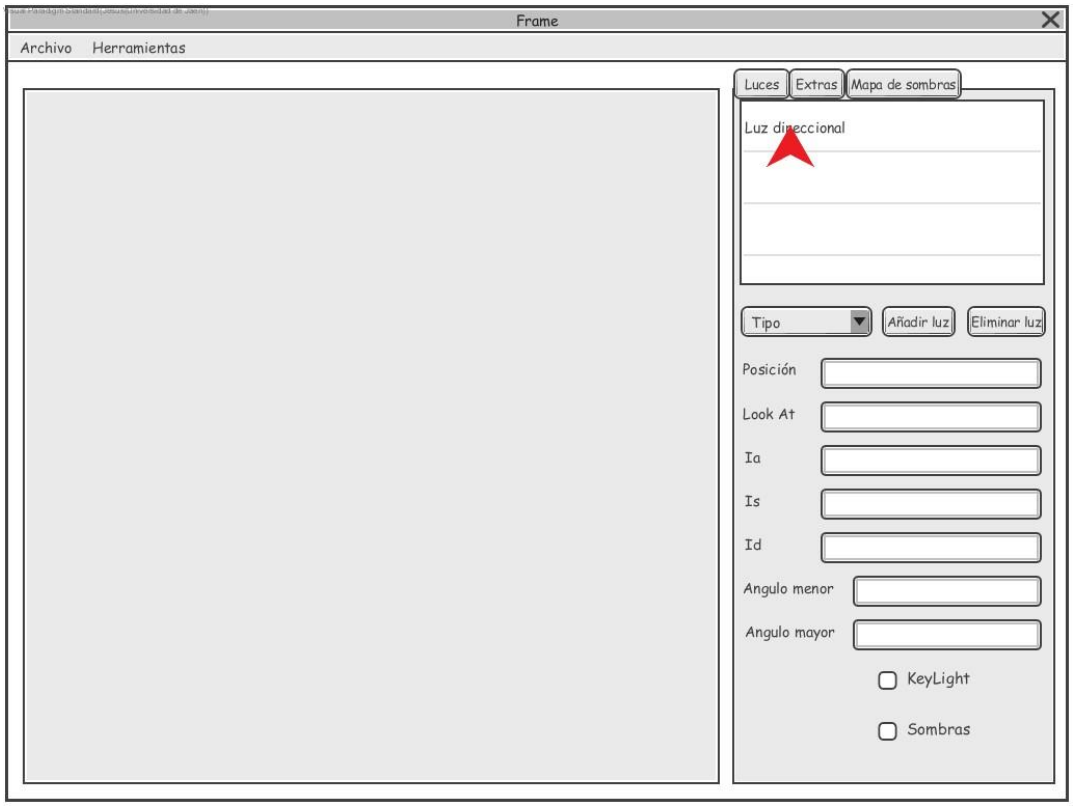


Ilustración 2.34 Storyboards: Editar luz. Selección de la luz

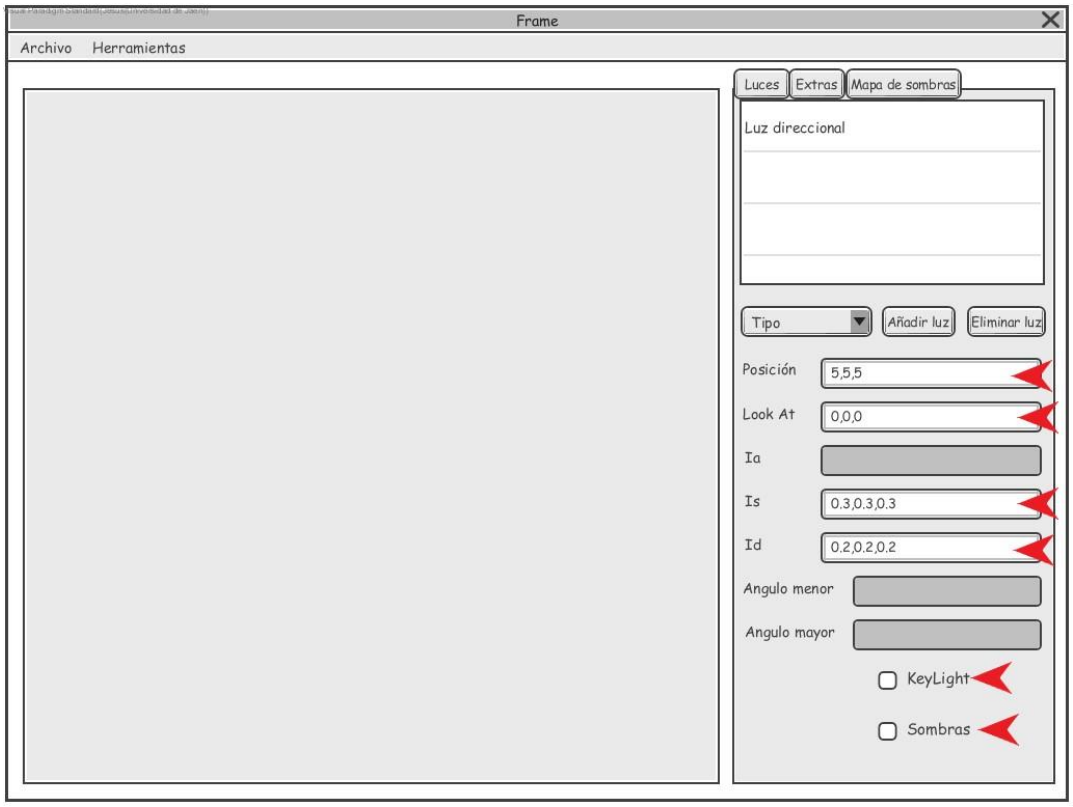


Ilustración 2.35 Storyboards: Editar luz. Modificación de parámetros

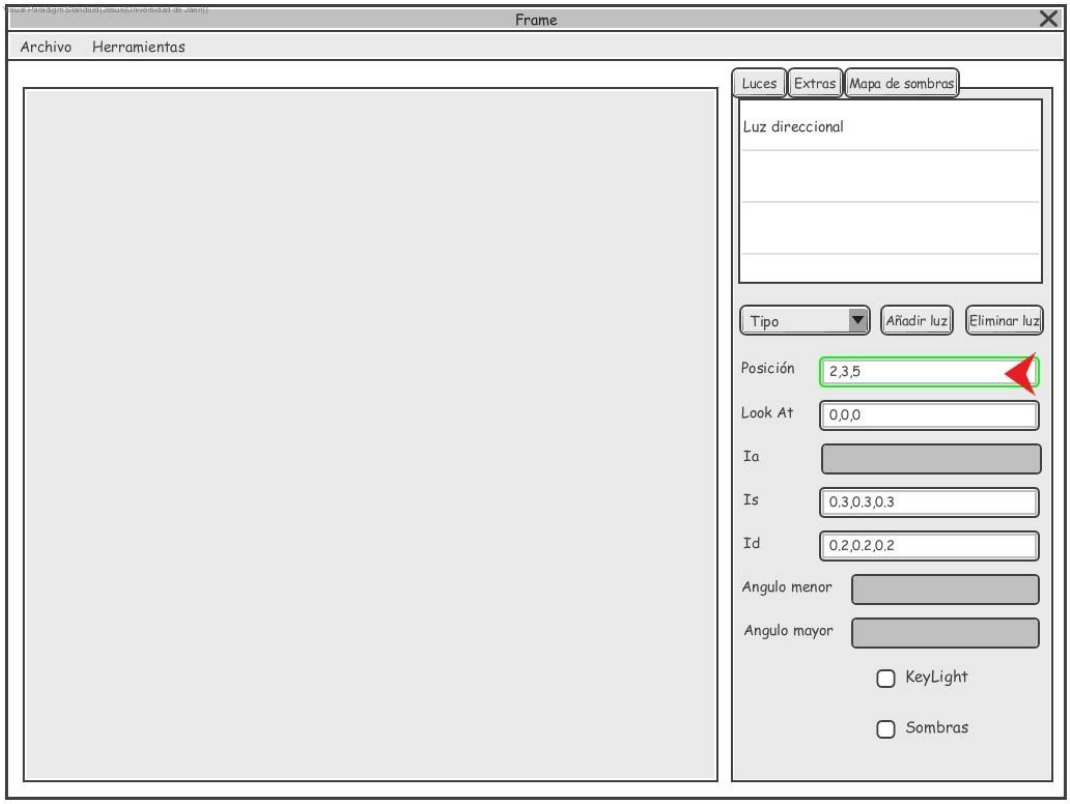


Ilustración 2.36 Storyboards: Editar luz. Parámetros correctos

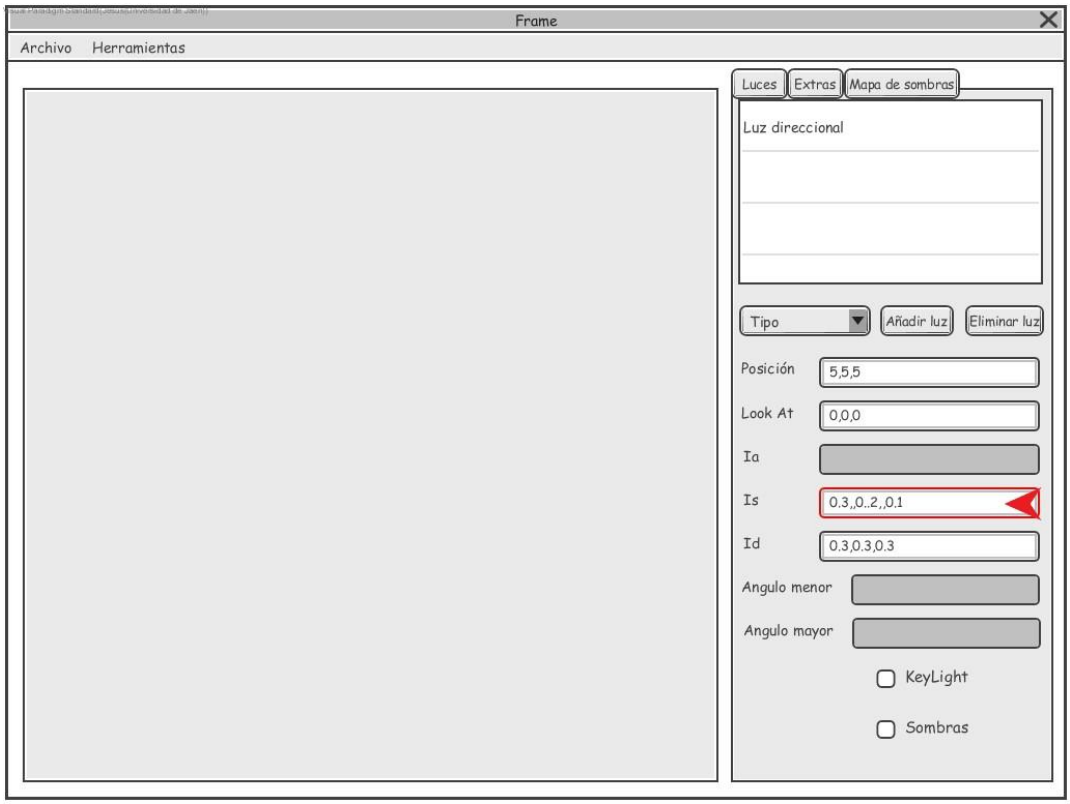


Ilustración 2.37 Storyboards: Editar luz. Parámetros incorrectos

- Eliminar luz:

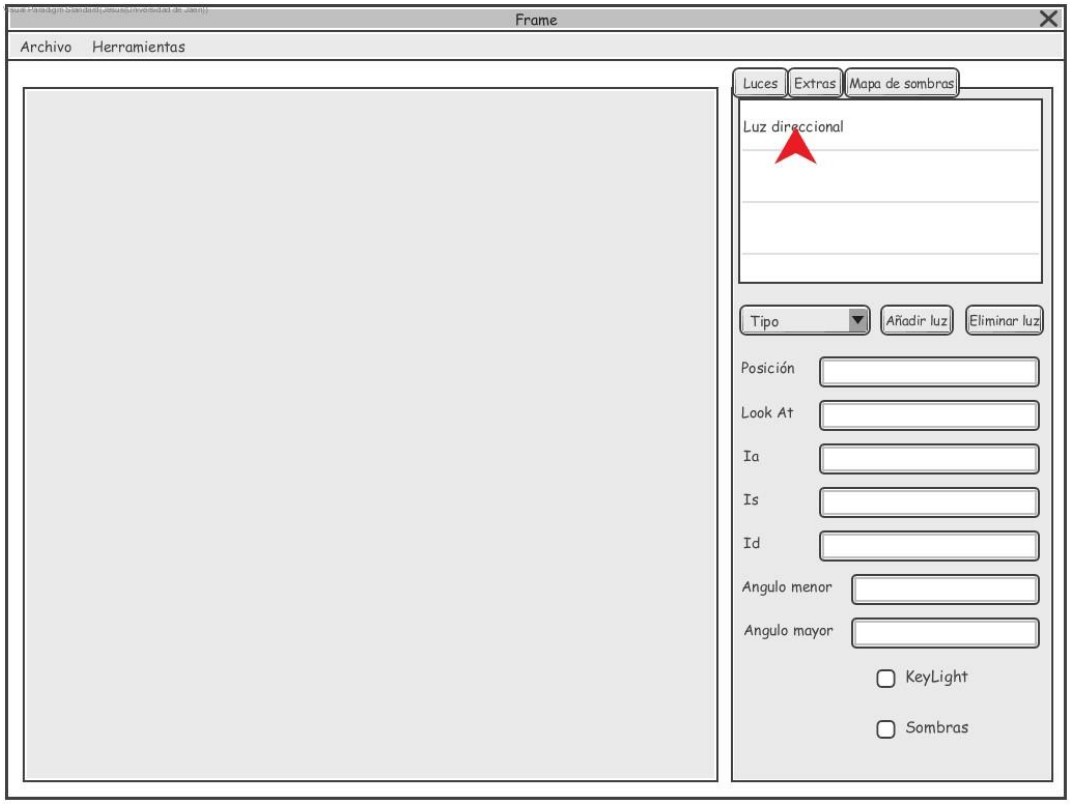


Ilustración 2.38 Storyboards: Eliminar luz. Selección de la luz

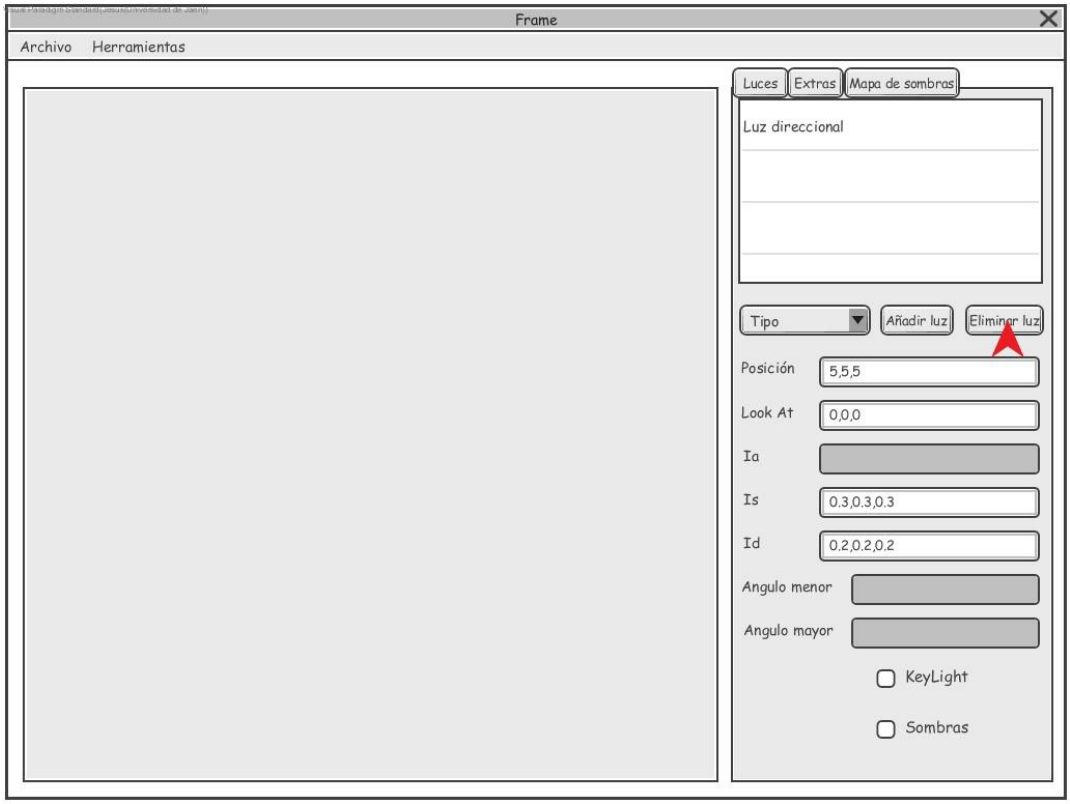


Ilustración 2.39 Storyboards: Eliminar luz. Confirmación

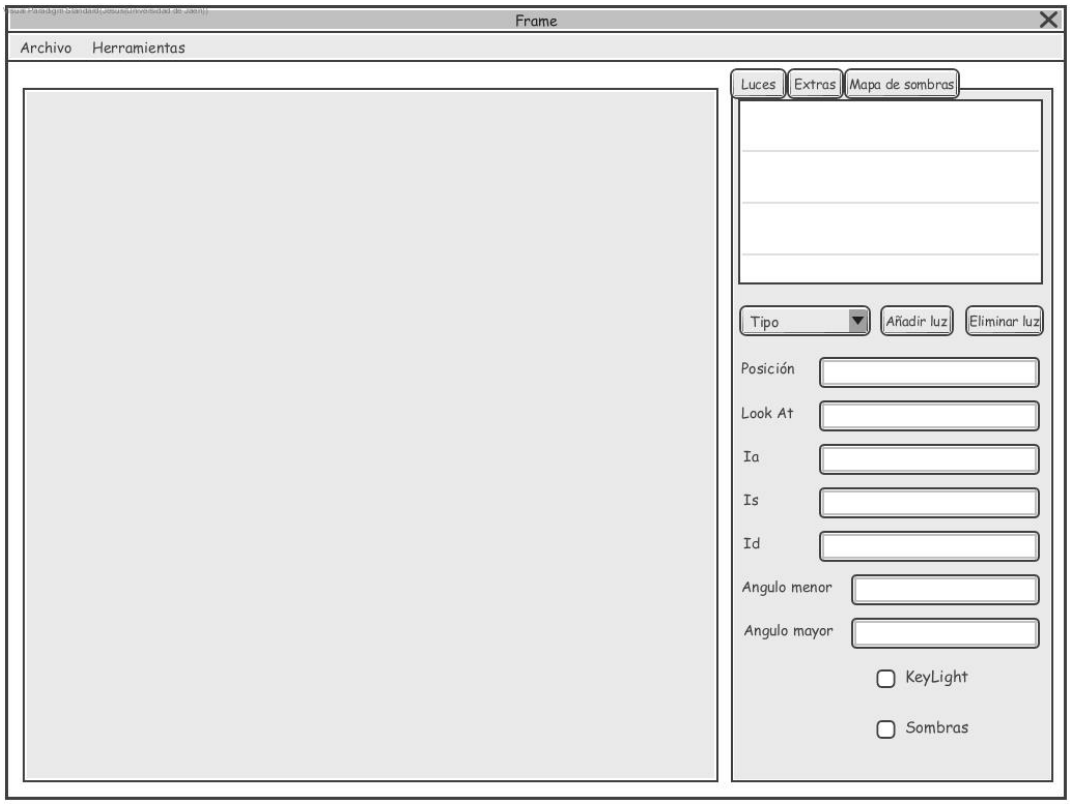


Ilustración 2.40 Storyboards: Eliminar luz. Luz eliminada de la lista de luces

- Cargar configuración:

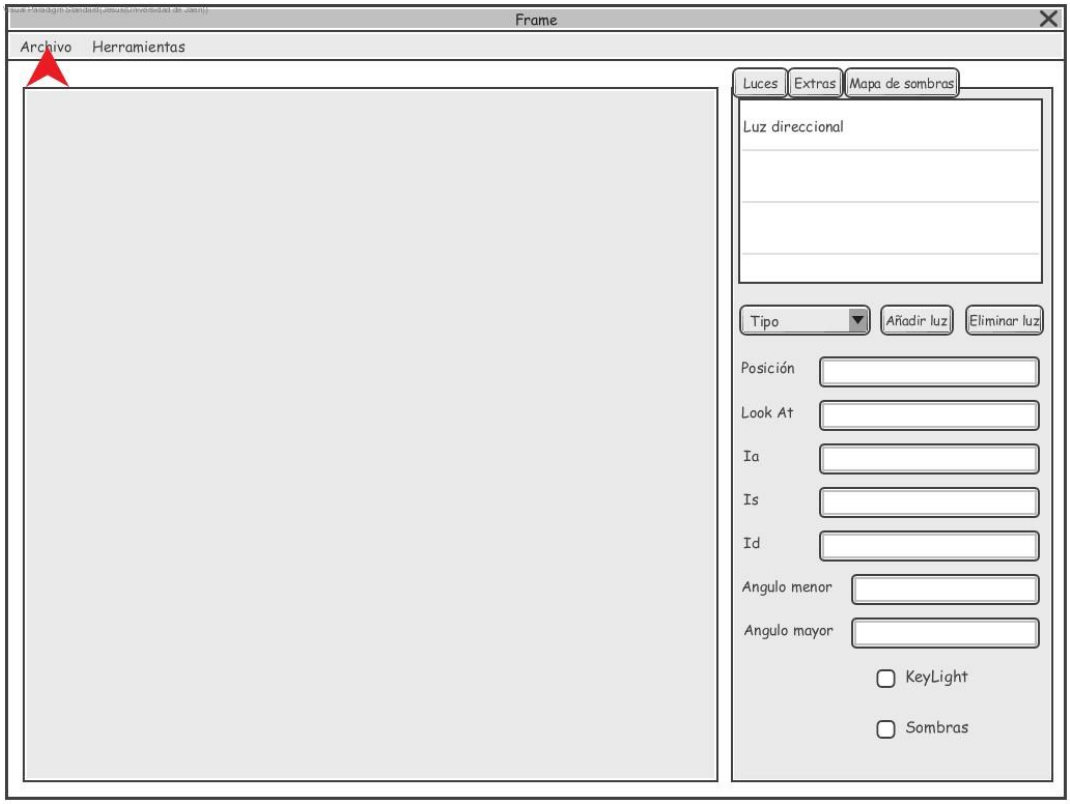


Ilustración 2.41 Storyboards: Cargar configuración. Desplegar menú "Archivo"

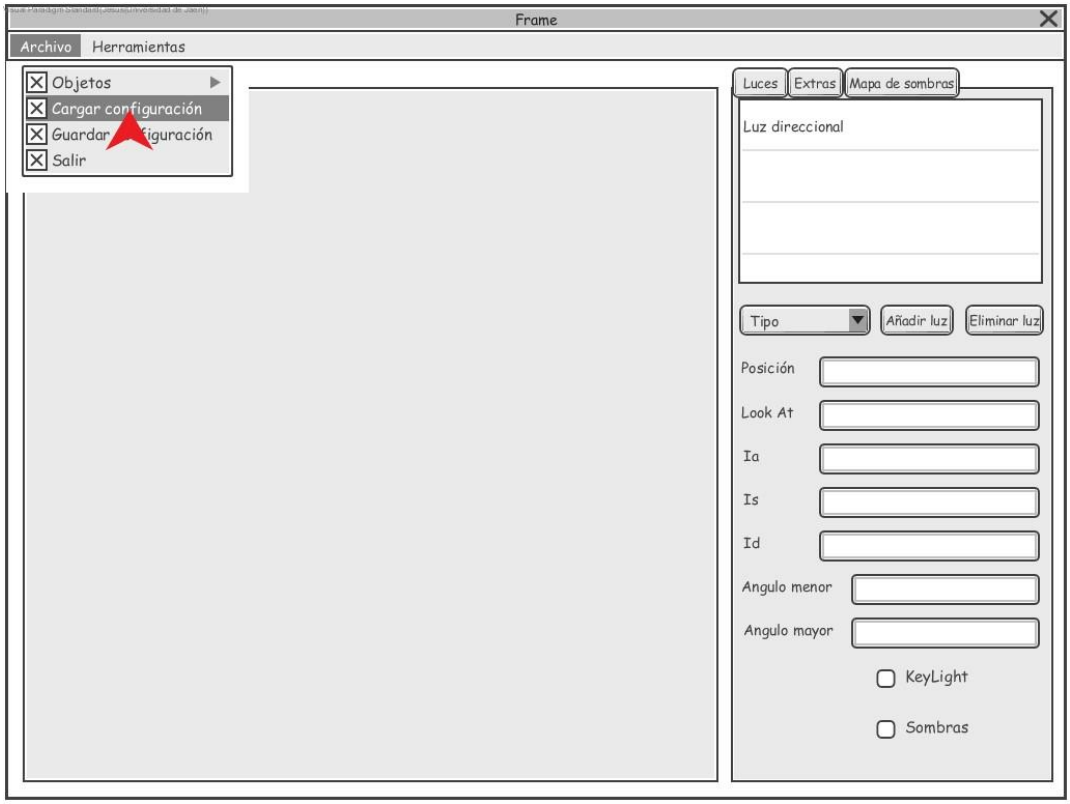


Ilustración 2.42 Storyboards: Cargar configuración. Opción "Cargar configuración"

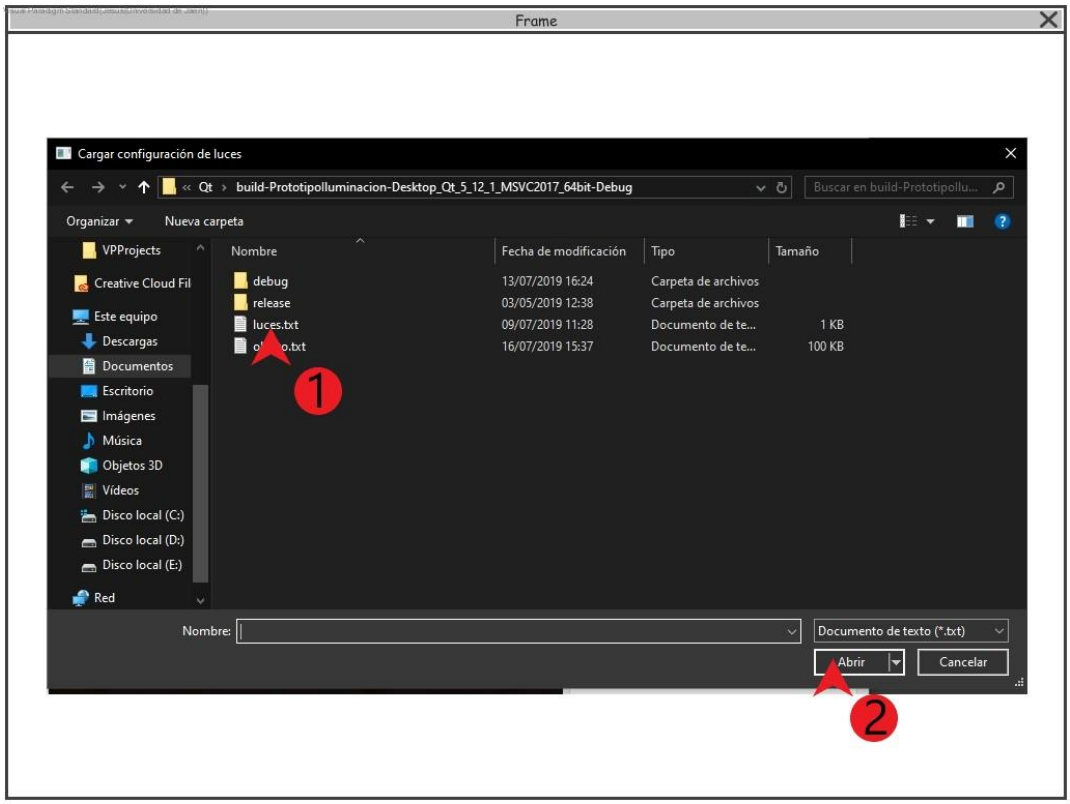


Ilustración 2.43 Storyboards: Cargar configuración. Elección del archivo

- Guardar configuración:

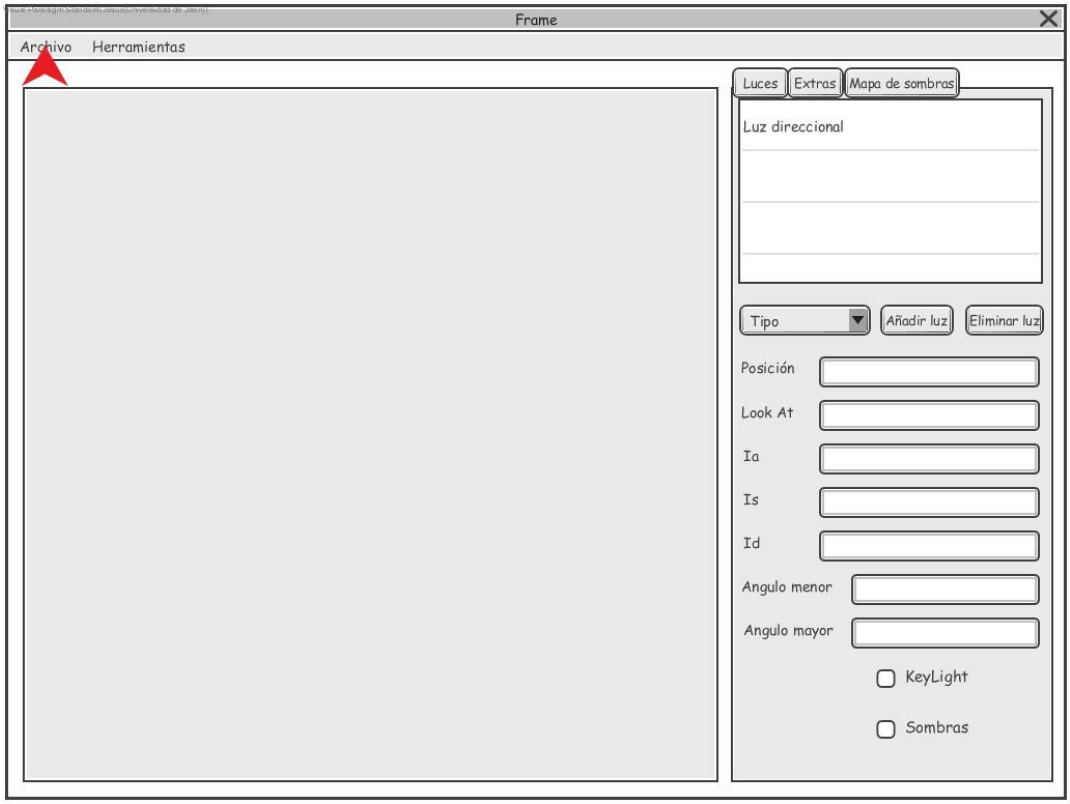


Ilustración 2.44 Storyboards: Guardar configuración. Desplegar menú "Archivo"

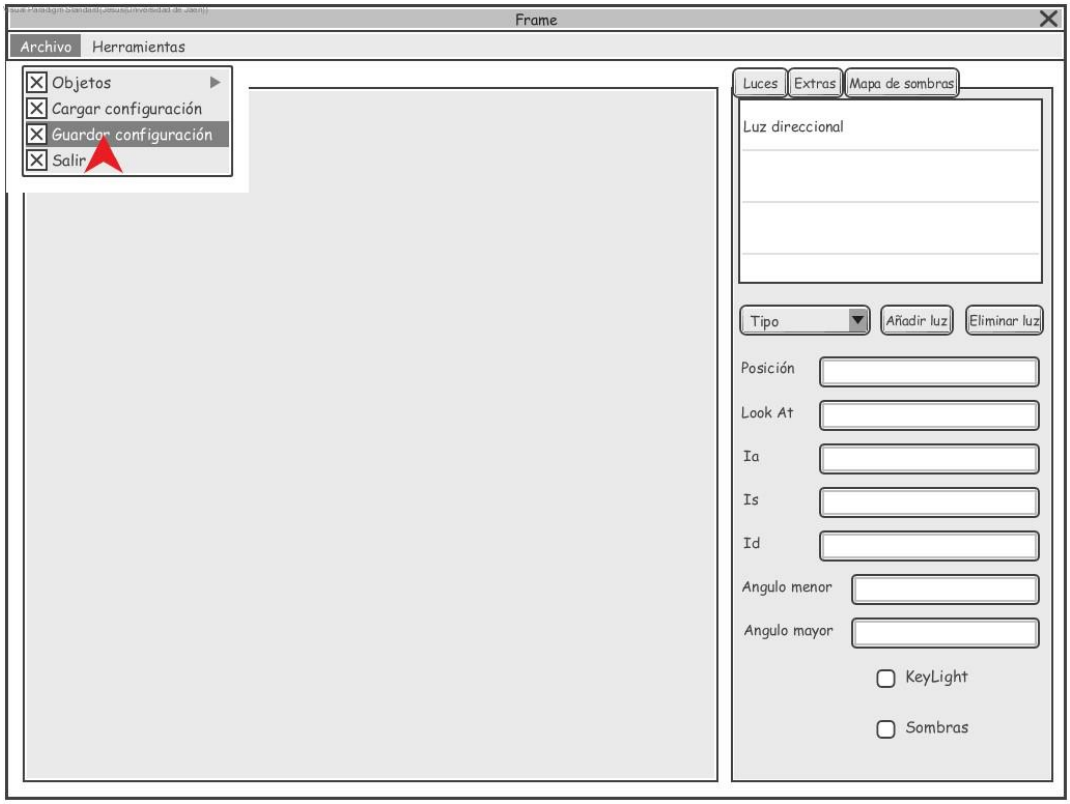


Ilustración 2.45 Storyboards: Guardar configuración. Opción "Guardar configuración"

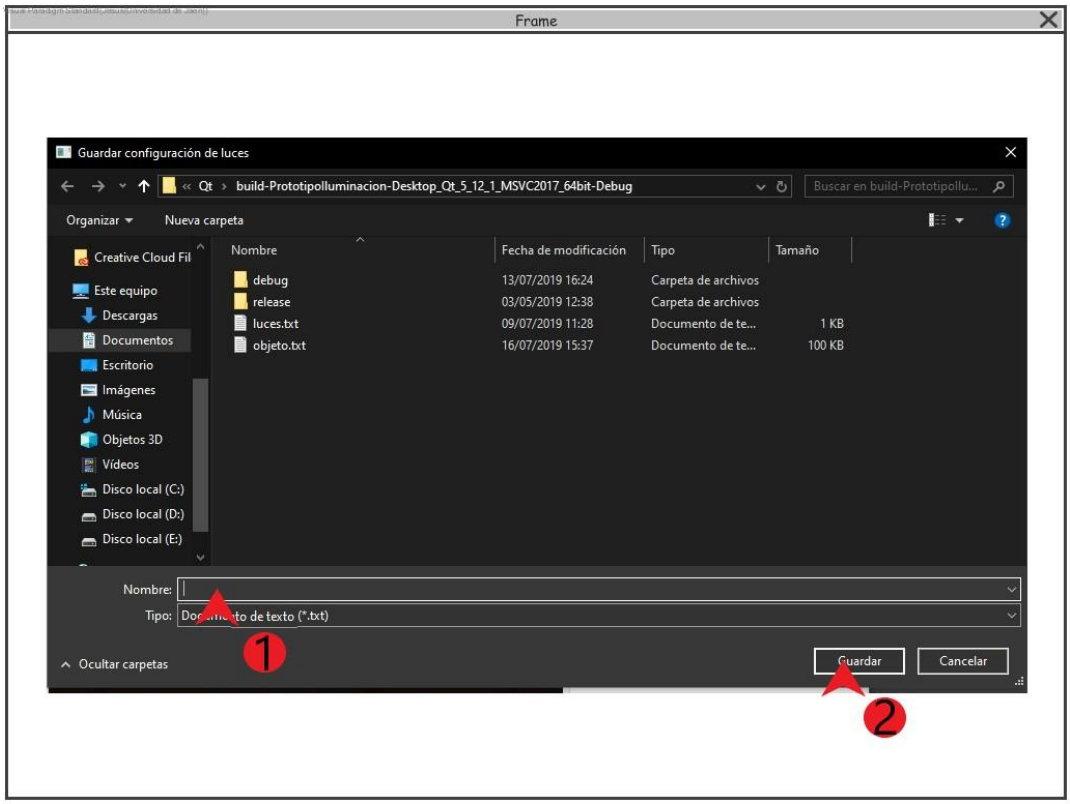


Ilustración 2.46 Storyboards: Guardar configuración. Selección de nombre y confirmación

- Activar/desactivar atenuación:

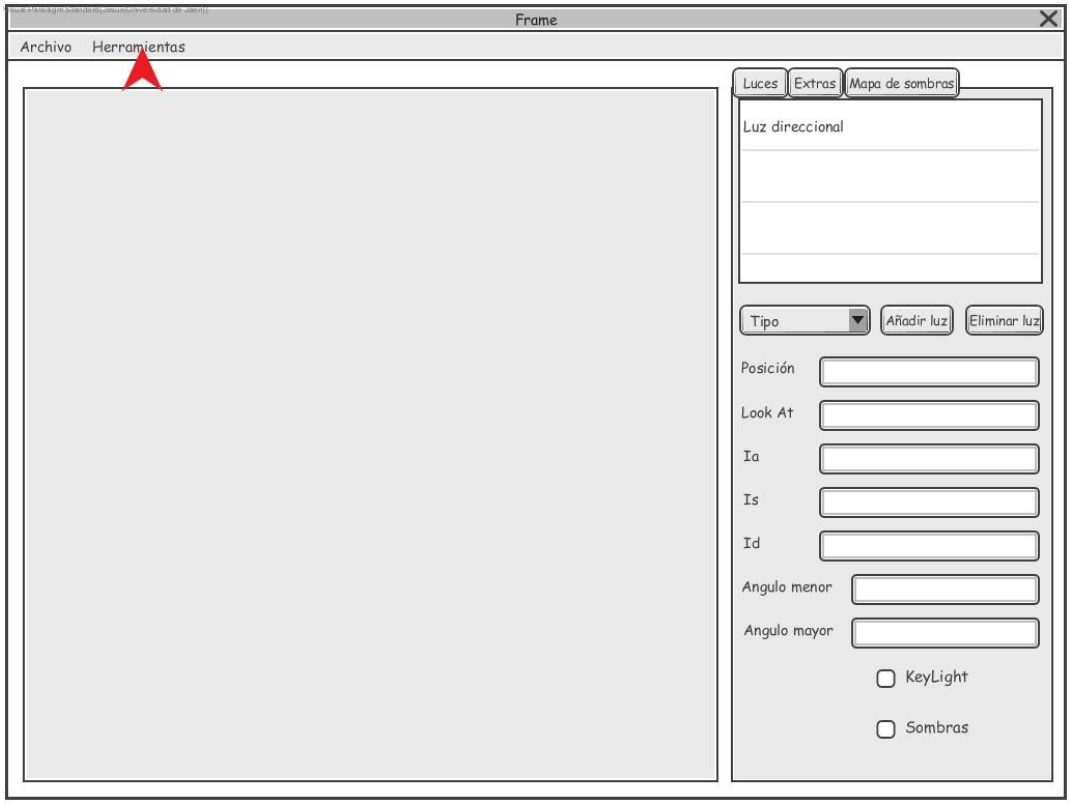


Ilustración 2.47 Storyboards: Activar/desactivar atenuación. Desplegar menú "Herramientas"

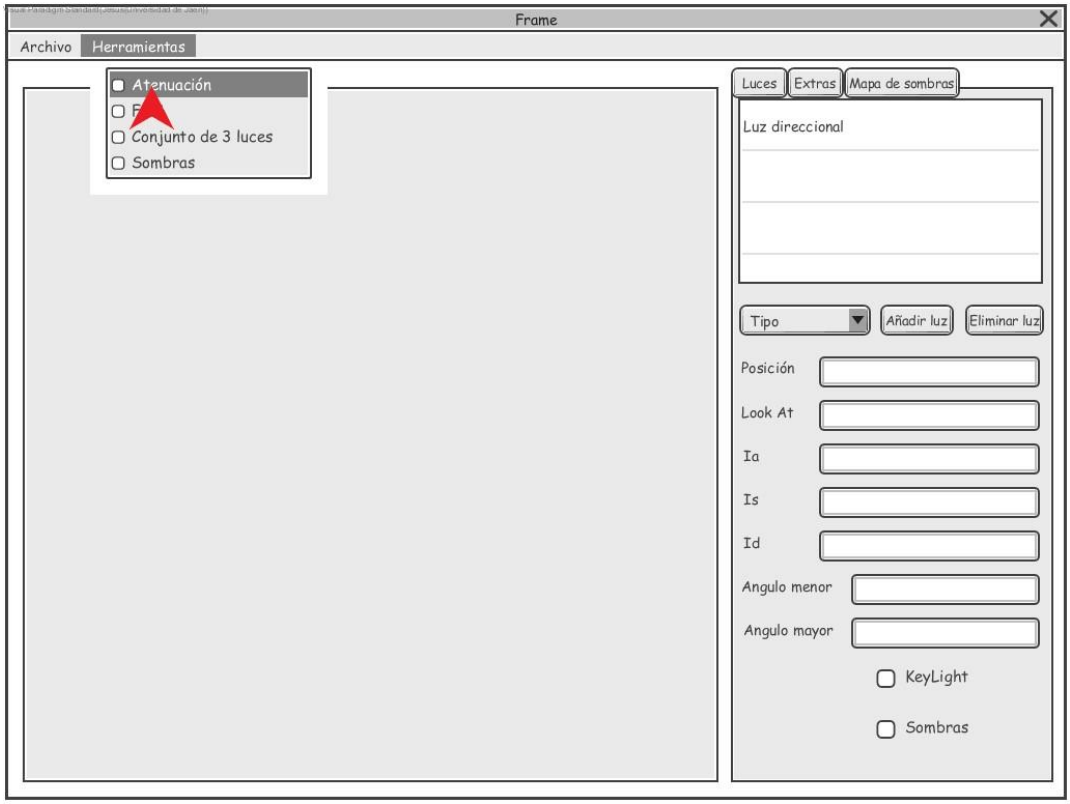


Ilustración 2.48 Storyboards: Activar/desactivar atenuación. Opción "Atenuación"

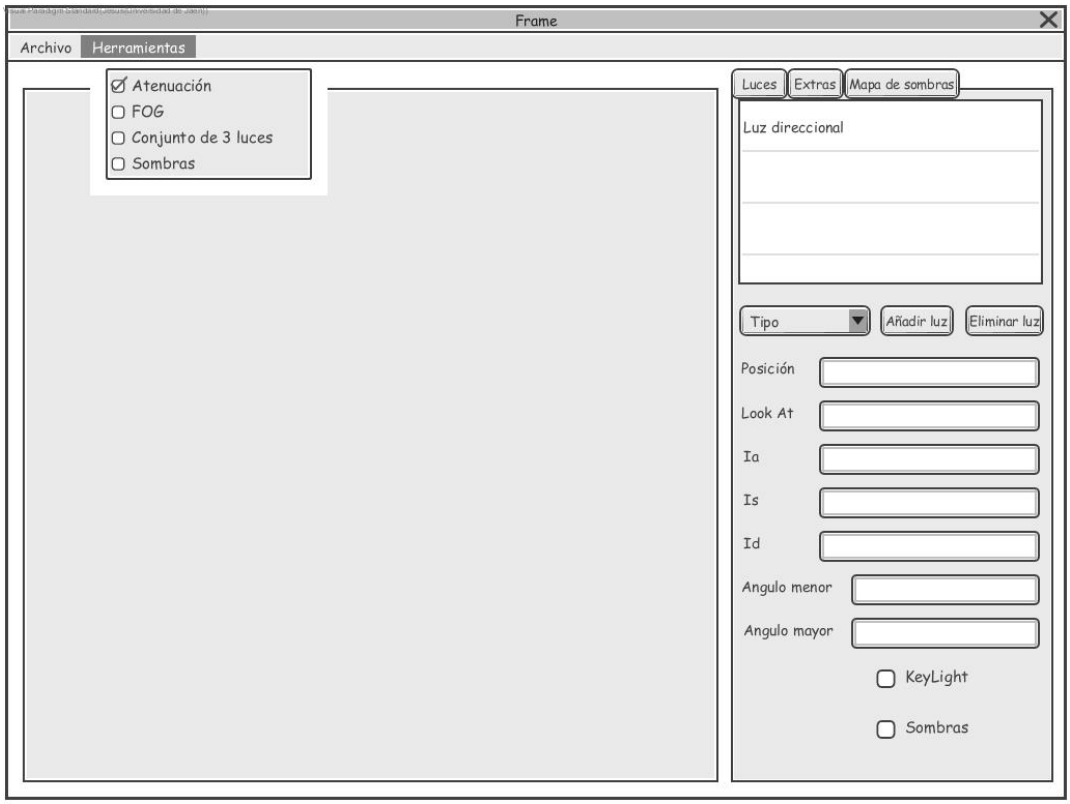


Ilustración 2.49 Storyboards: Activar/desactivar atenuación. Atenuación activa

Ilustración 2.50 Storyboards: Activar/desactivar atenuación. Atenuación sin activar

- Activar/desactivar FOG:

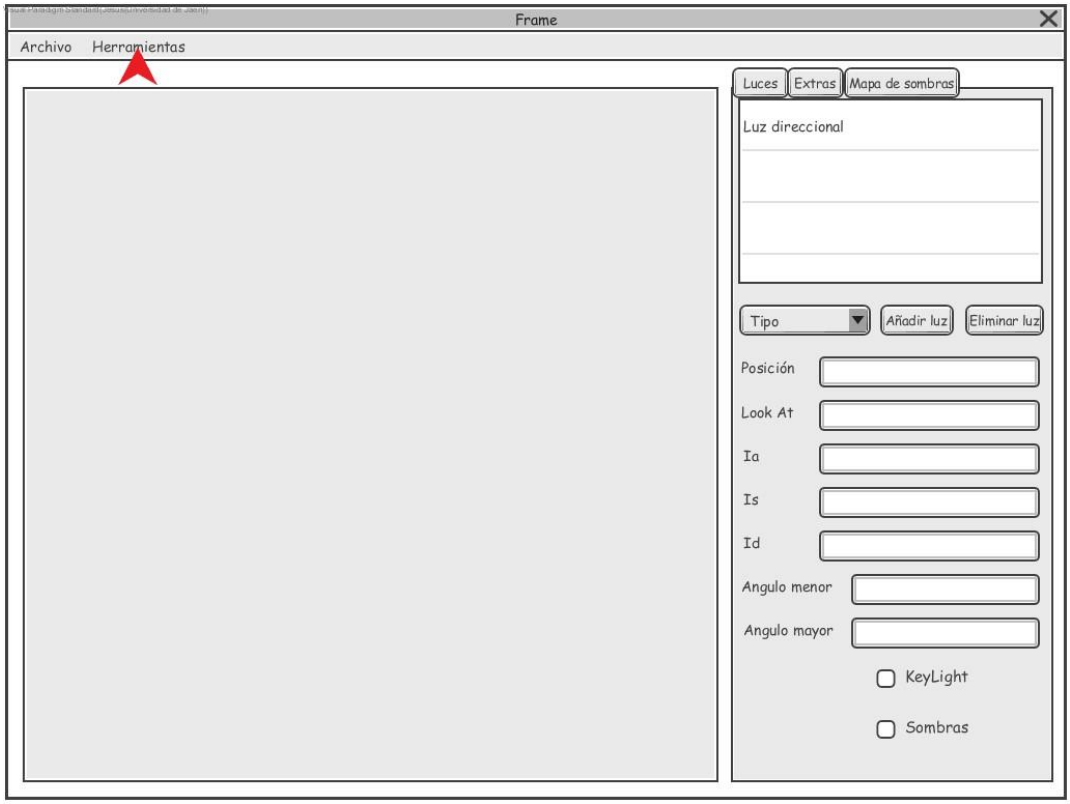


Ilustración 2.51 Storyboards: Activar/desactivar FOG. Desplegar menú "Herramientas"

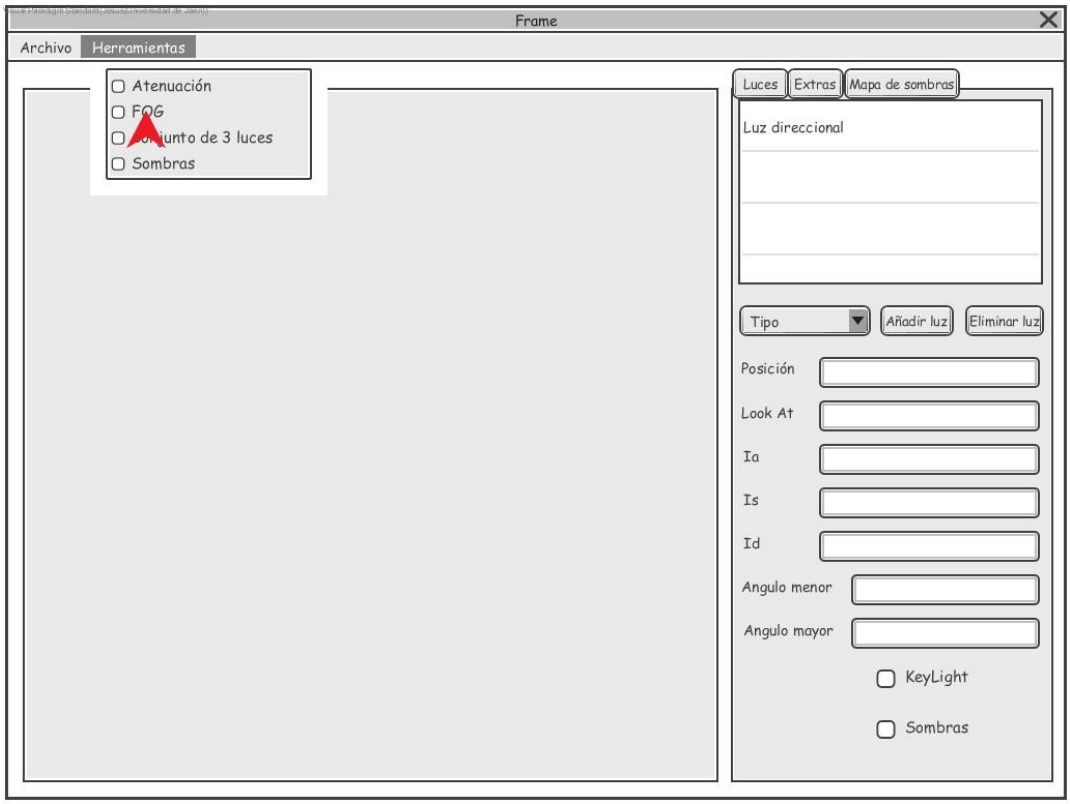


Ilustración 2.52 Storyboards: Activar/desactivar FOG. Opción "FOG"

Ilustración 2.53 Storyboards: Activar/desactivar FOG. FOG activo

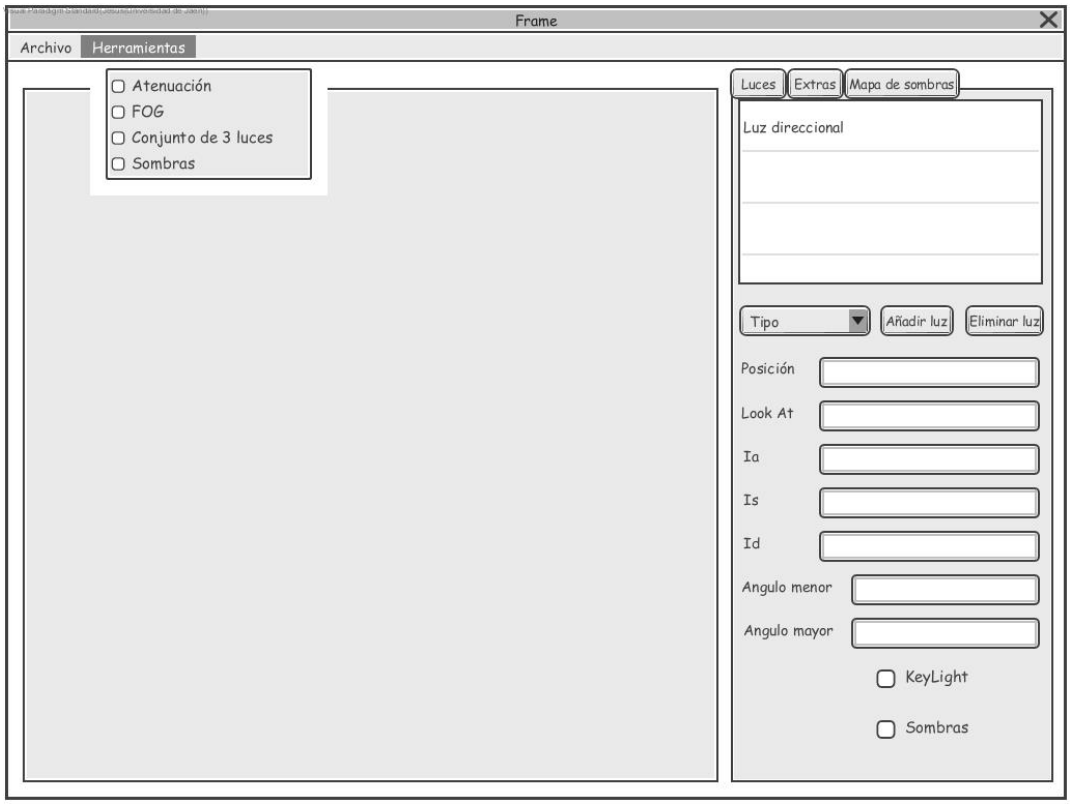


Ilustración 2.54 Storyboards: Activar/desactivar FOG. FOG sin activar

- Activar/desactivar conjunto de 3 luces:

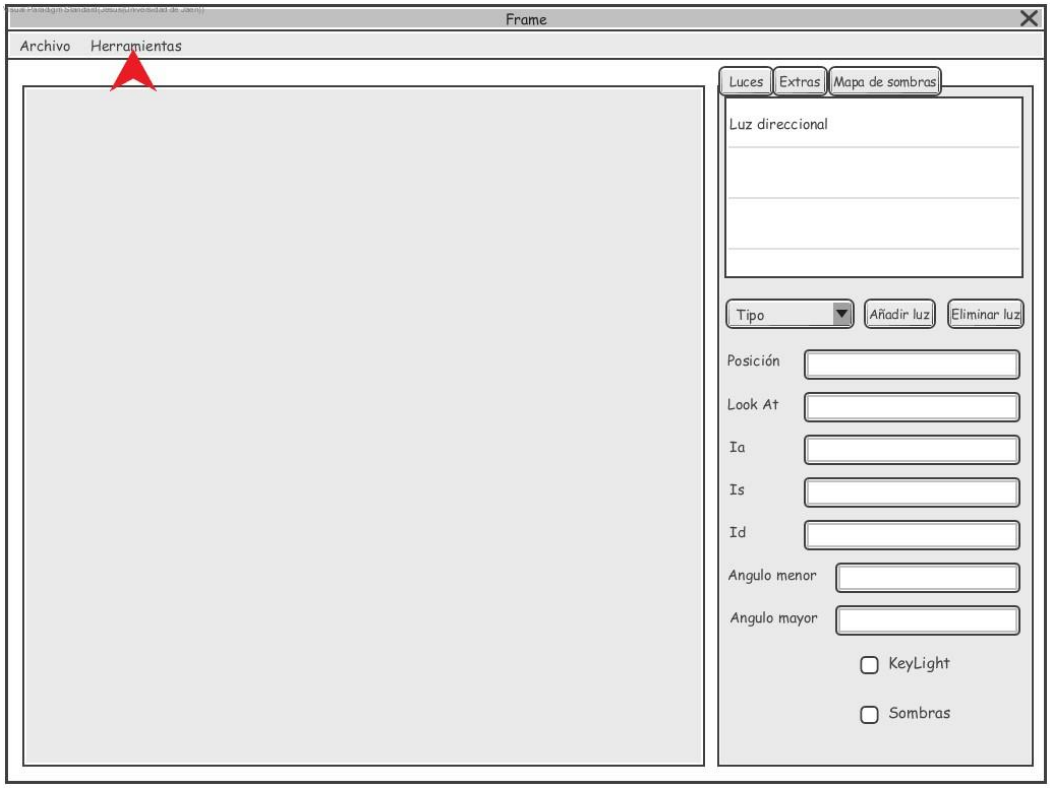


Ilustración 2.55 Storyboards: Activar/desactivar conjunto de 3 luces. Desplegar menú "Herramientas"

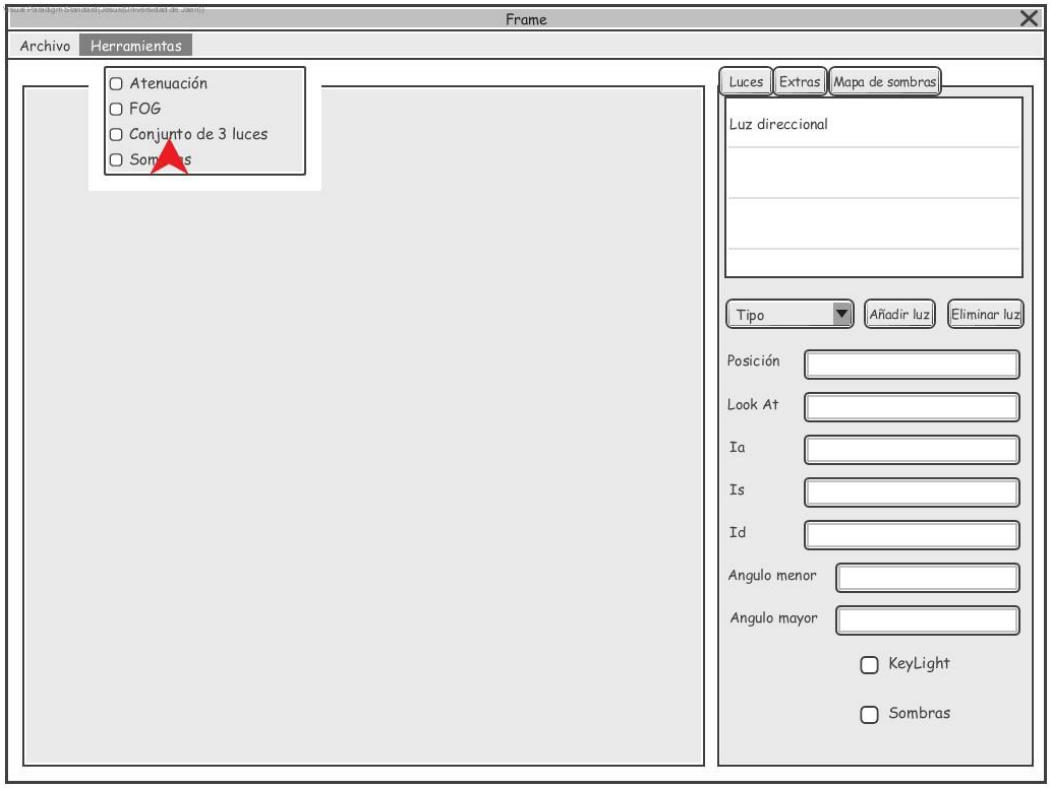


Ilustración 2.56 Storyboards: Activar/desactivar conjunto de 3 luces. Opción "Conjunto de 3 luces"

Ilustración 2.57 Storyboards: Activar/desactivar conjunto de 3 luces. Conjunto de 3 luces activo

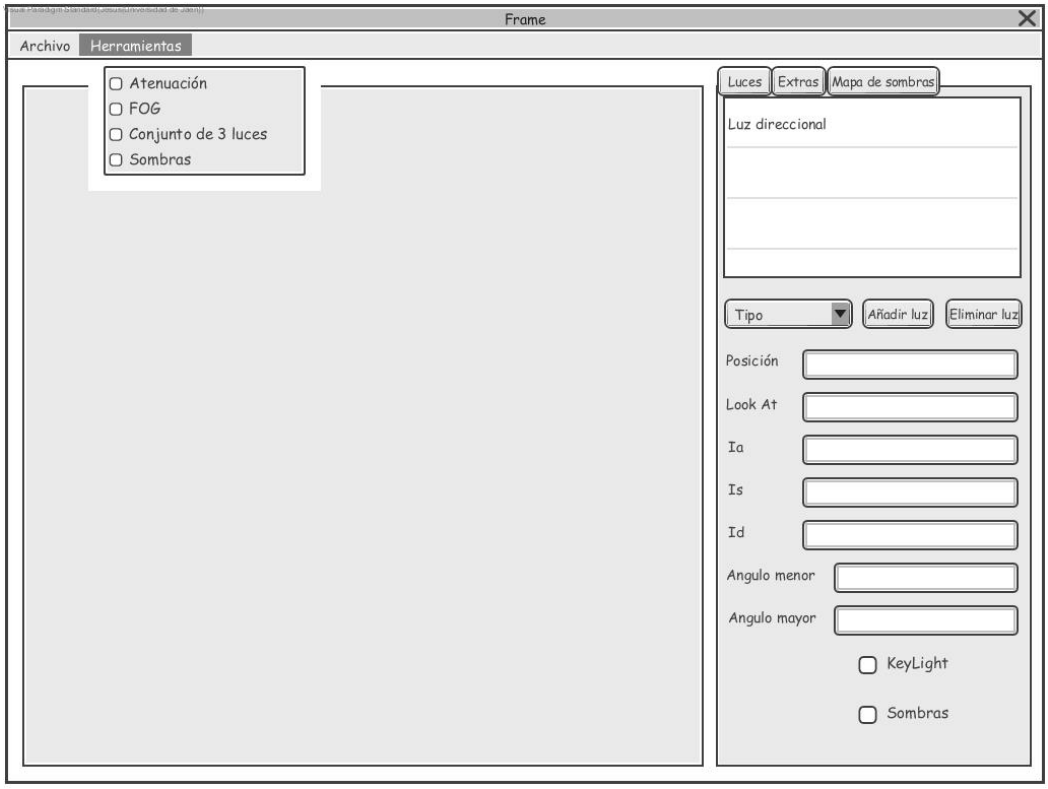


Ilustración 2.58 Storyboards: Activar/desactivar conjunto de 3 luces. Conjunto de 3 luces sin activar

- Activar/desactivar sombras:

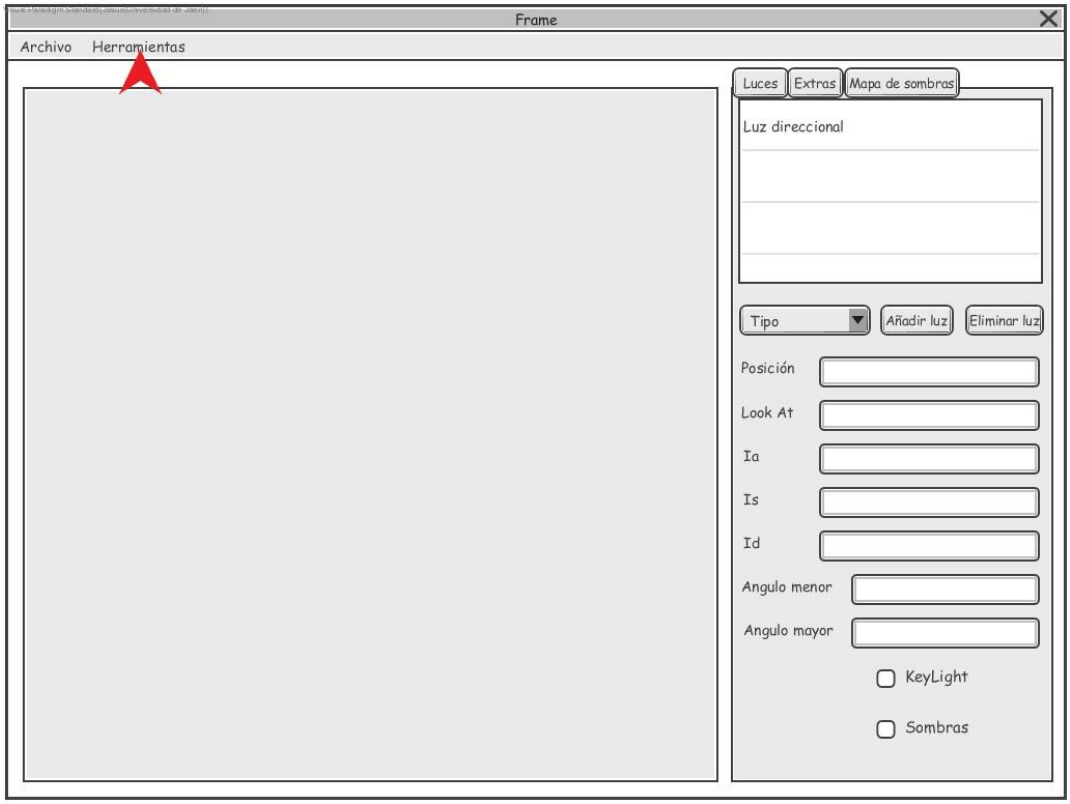


Ilustración 2.59 Storyboards: Activar/desactivar sombras. Desplegar menú "Herramientas"

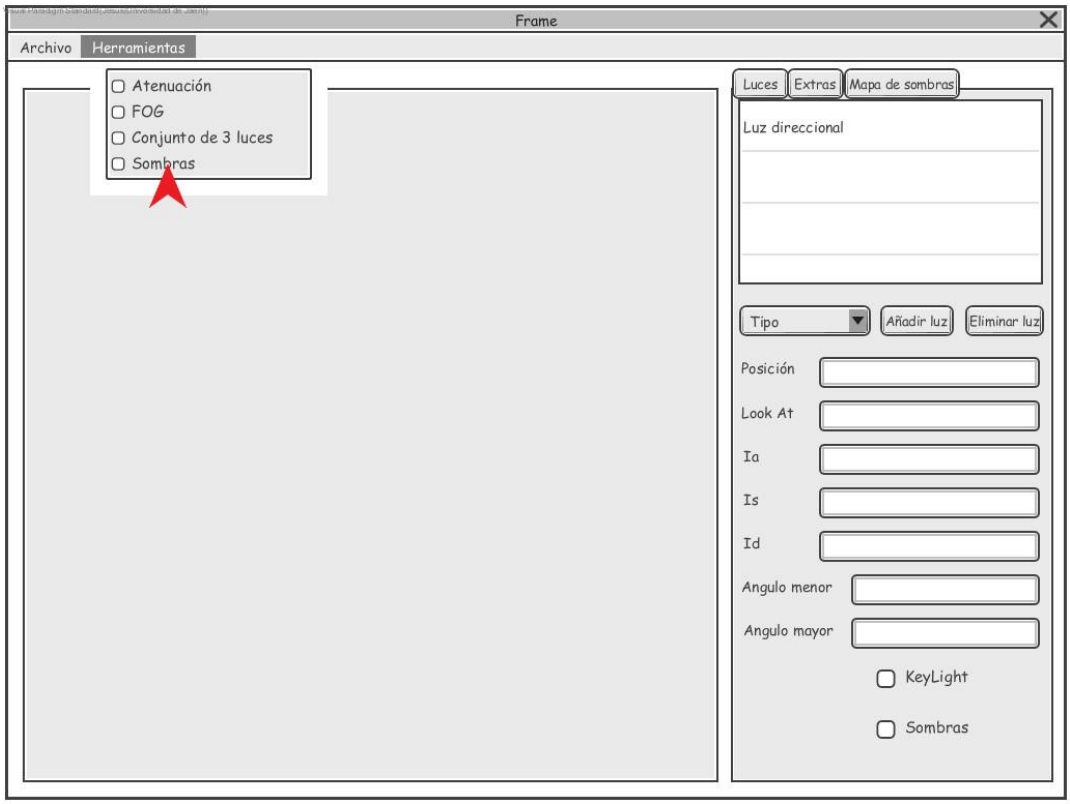


Ilustración 2.60 Storyboards: Activar/desactivar sombras. Opción "Sombras"

Ilustración 2.61 Storyboards: Activar/desactivar sombras. Sombras activadas

Ilustración 2.62 Storyboards: Activar/desactivar sombras. Sombras sin activar

- Editar atenuación:

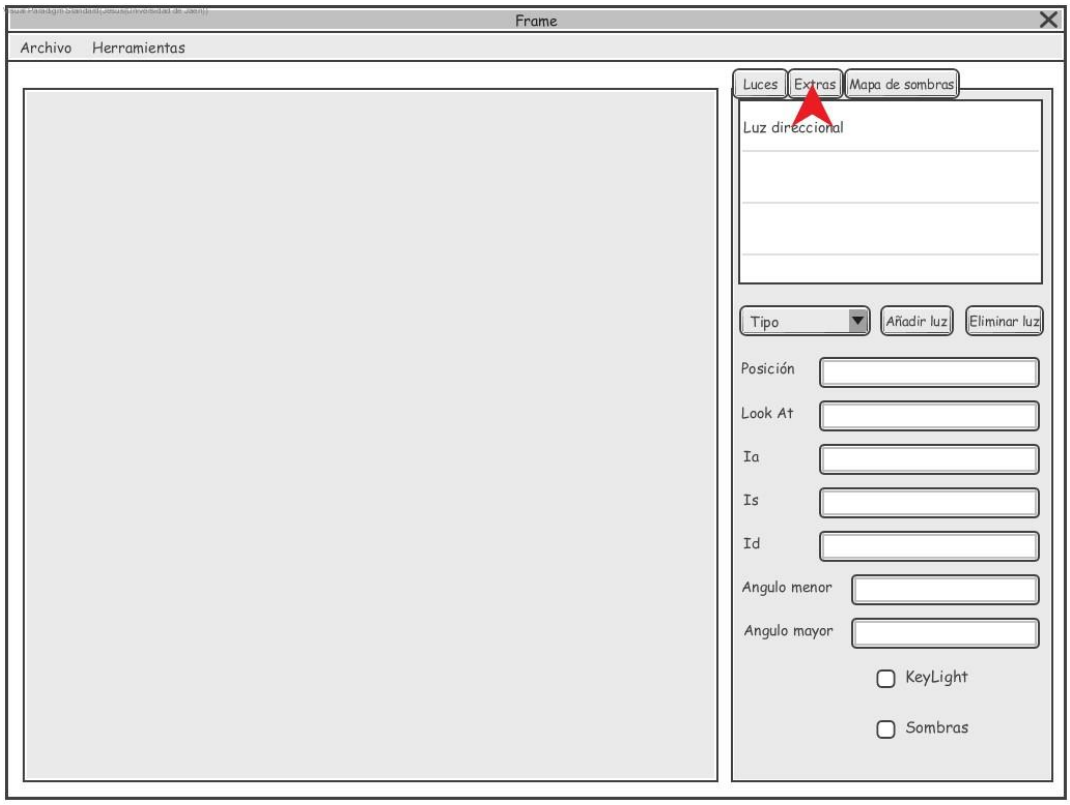


Ilustración 2.63 Storyboards: Editar atenuación. Pestaña "Extras"

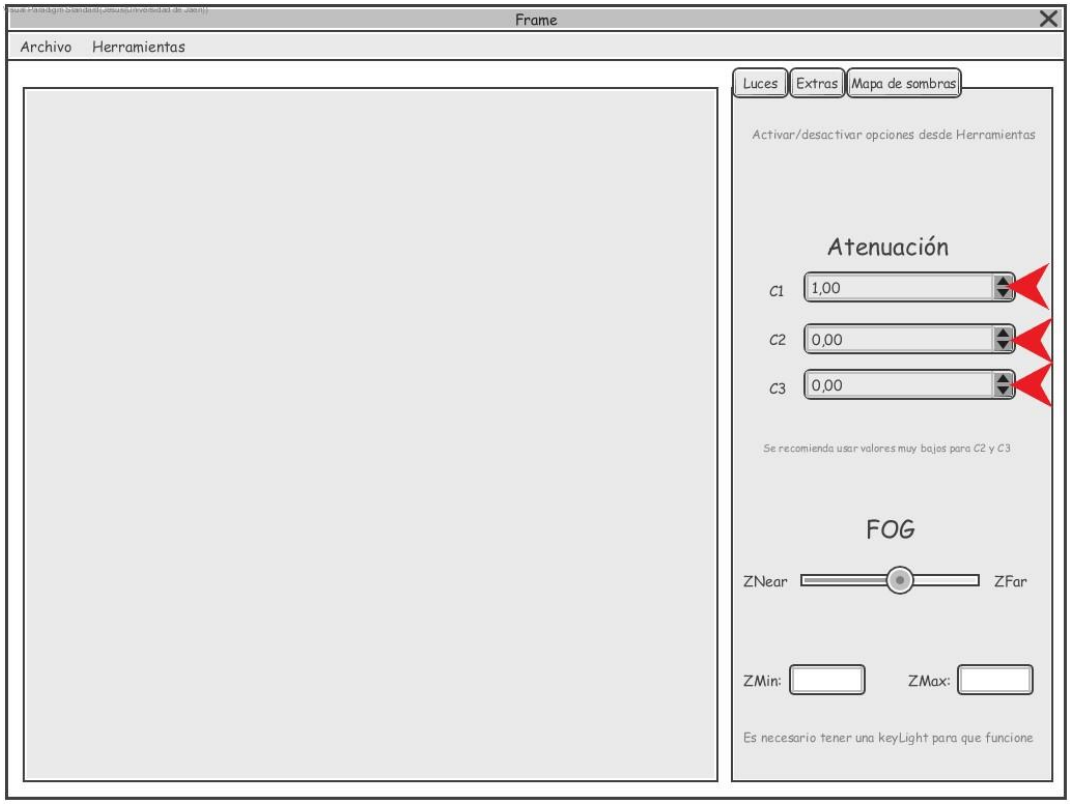


Ilustración 2.64 Storyboards: Editar atenuación. Modificación de los campos

Ilustración 2.65 Storyboards: Editar atenuación. Parámetros correctos

- Editar FOG:

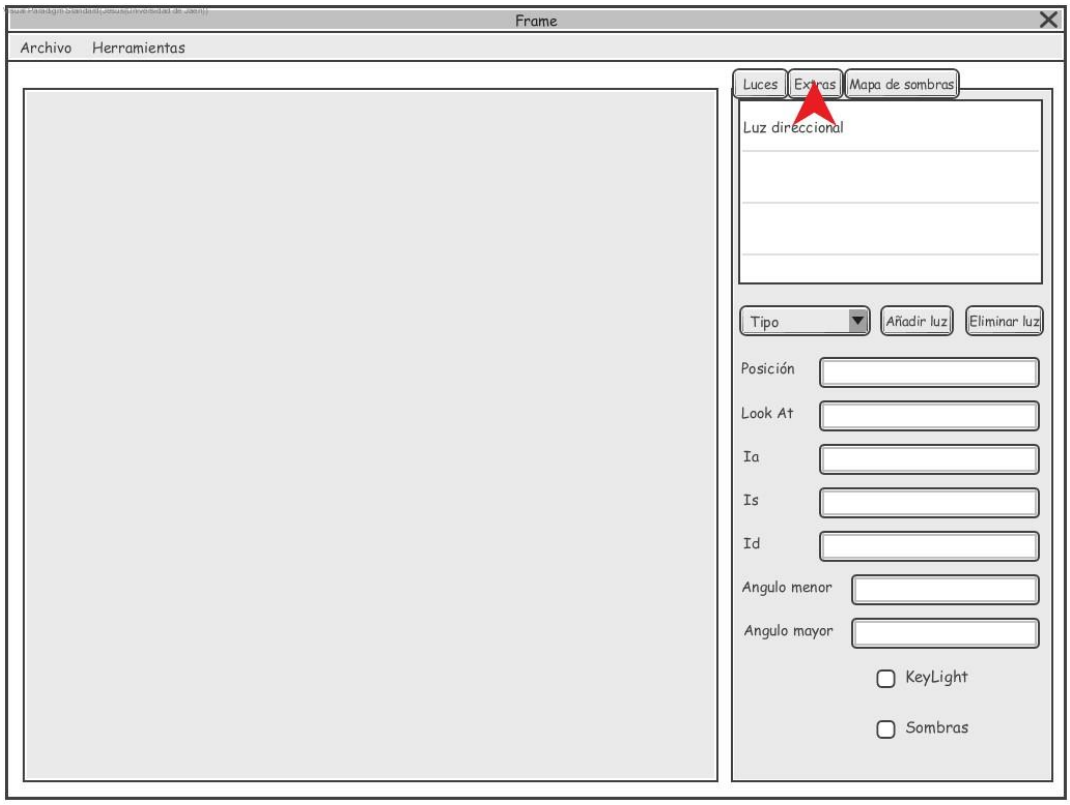


Ilustración 2.66 Storyboards: Editar FOG. Pestaña "Extras"

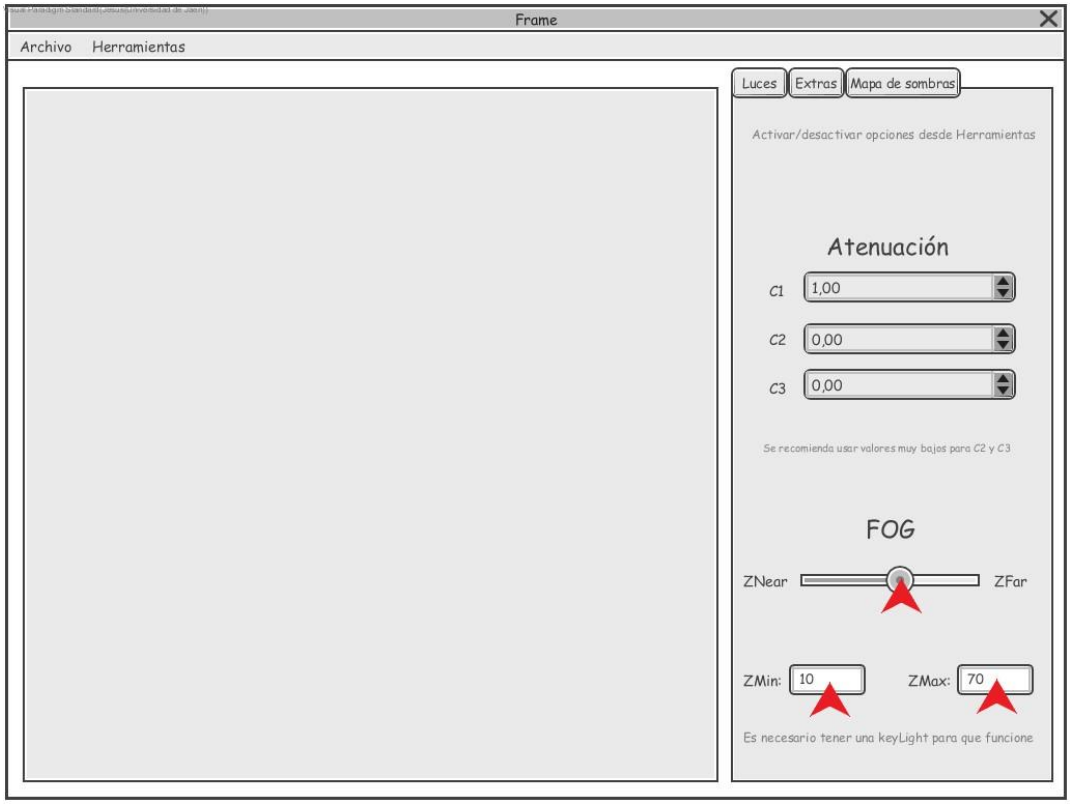


Ilustración 2.67 Storyboards: Editar FOG. Modificar campos

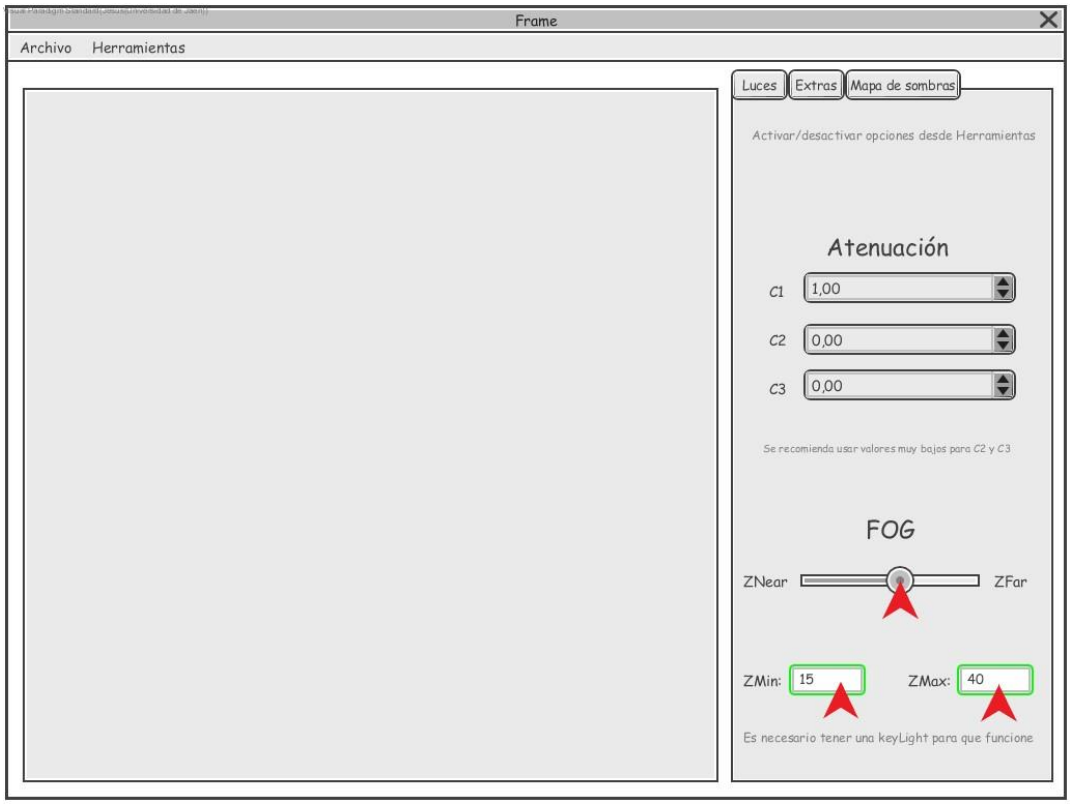


Ilustración 2.68 Storyboards: Editar FOG. Parámetros correctos

- Editar mapa de sombras:

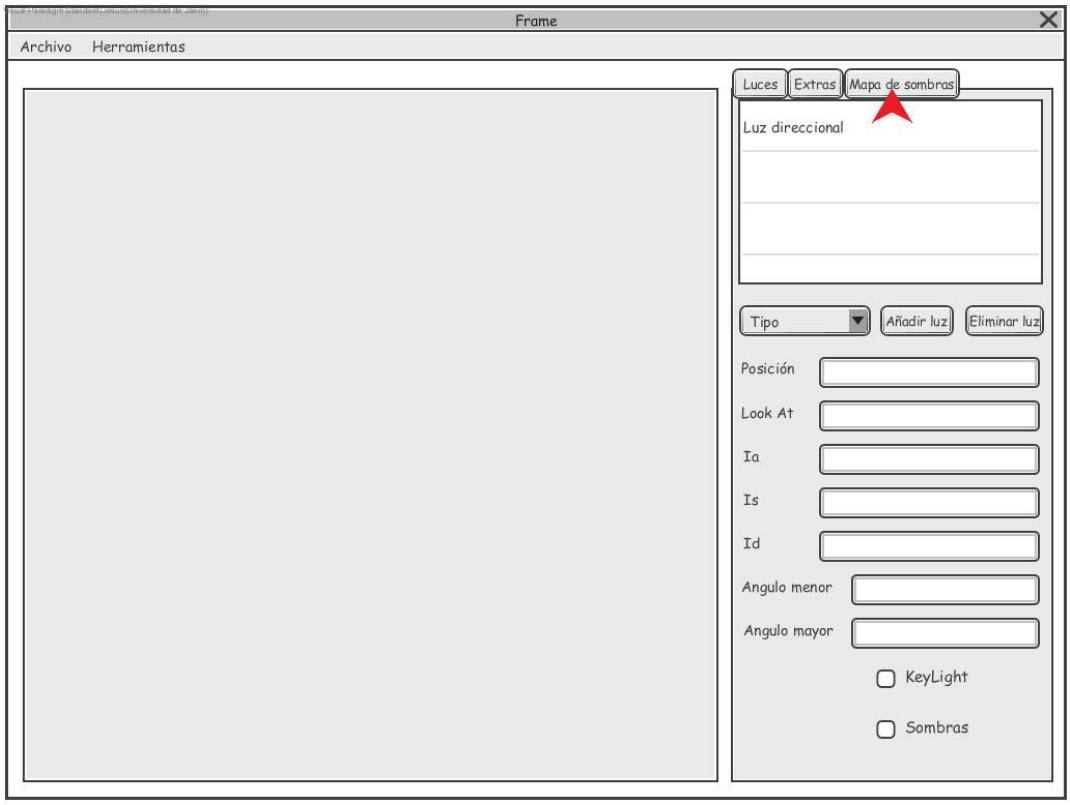


Ilustración 2.69 Storyboards: Editar mapa de sombras. Pestaña "Mapa de sombras"

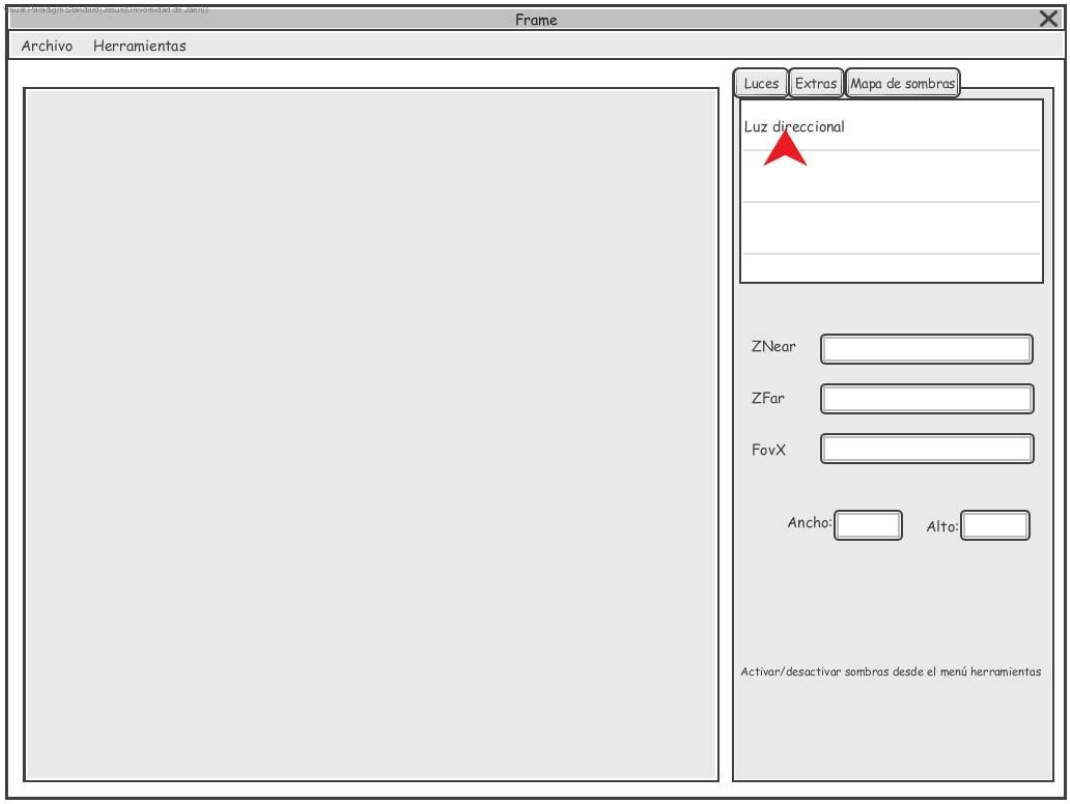


Ilustración 2.70 Storyboards: Editar mapa de sombras. Selección de luz

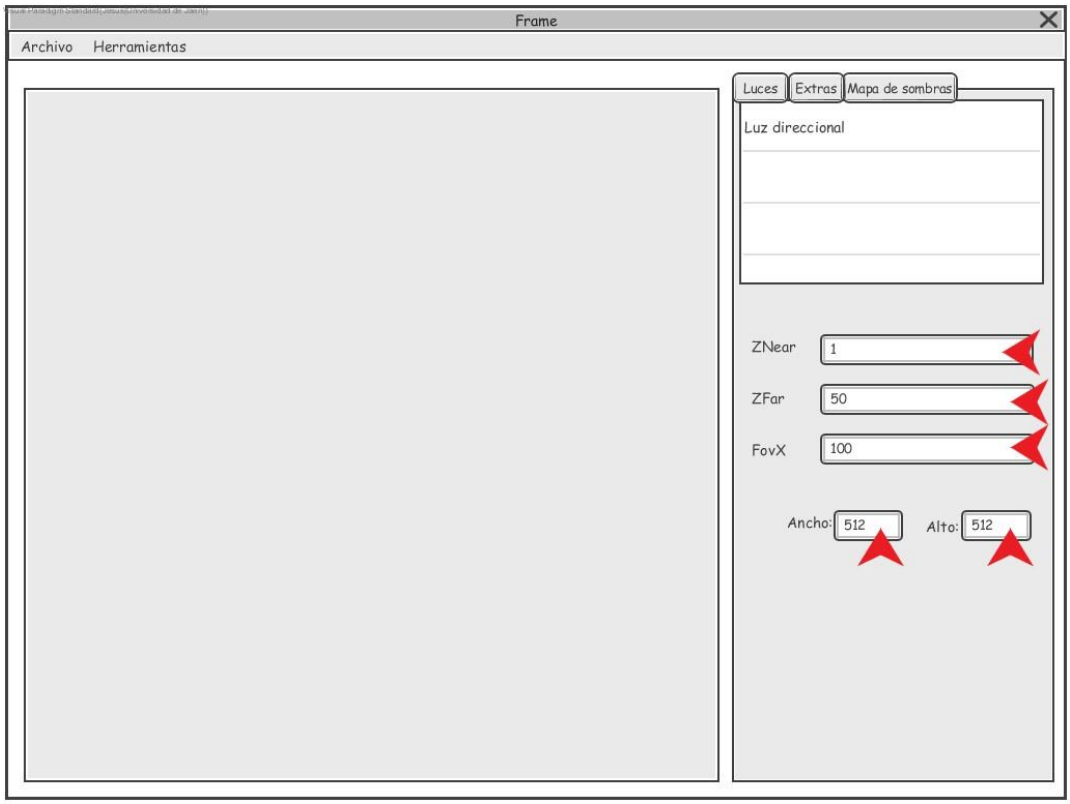


Ilustración 2.71 Storyboards: Editar mapa de sombras. Modificación de campos

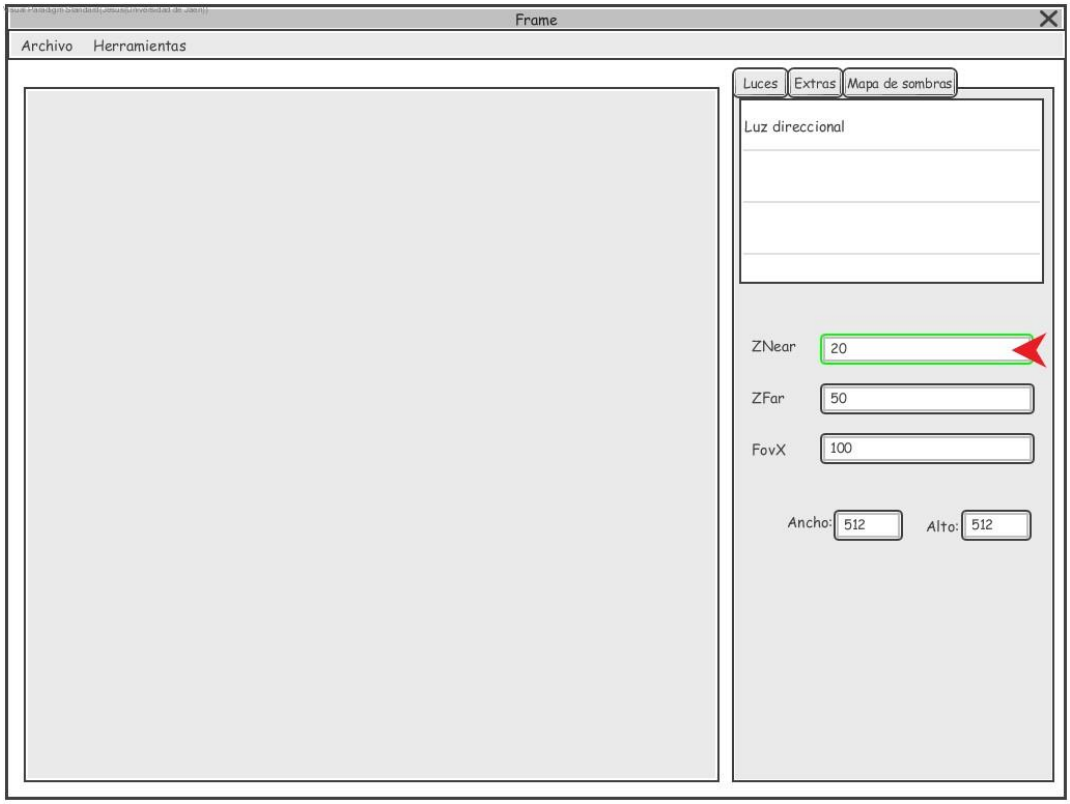


Ilustración 2.72 Storyboards: Editar mapa de sombras. Parámetros correctos

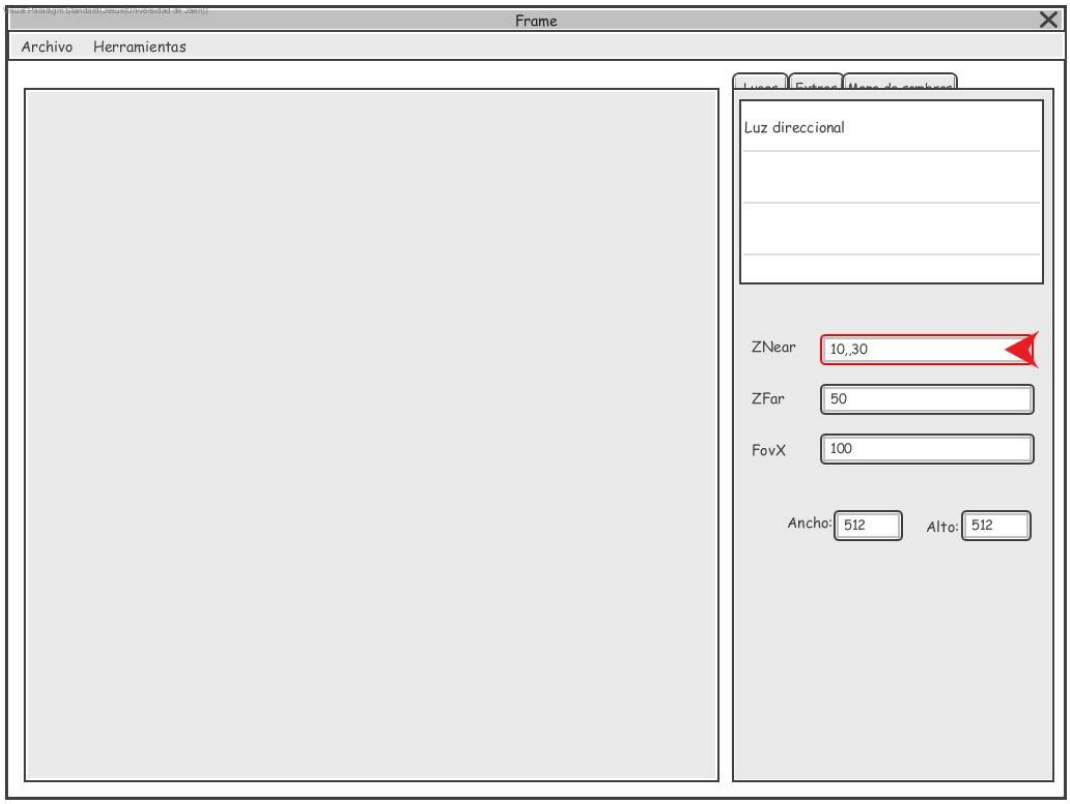


Ilustración 2.73 Storyboards: Editar mapa de sombras. Parámetros incorrectos

- Cambiar objeto:

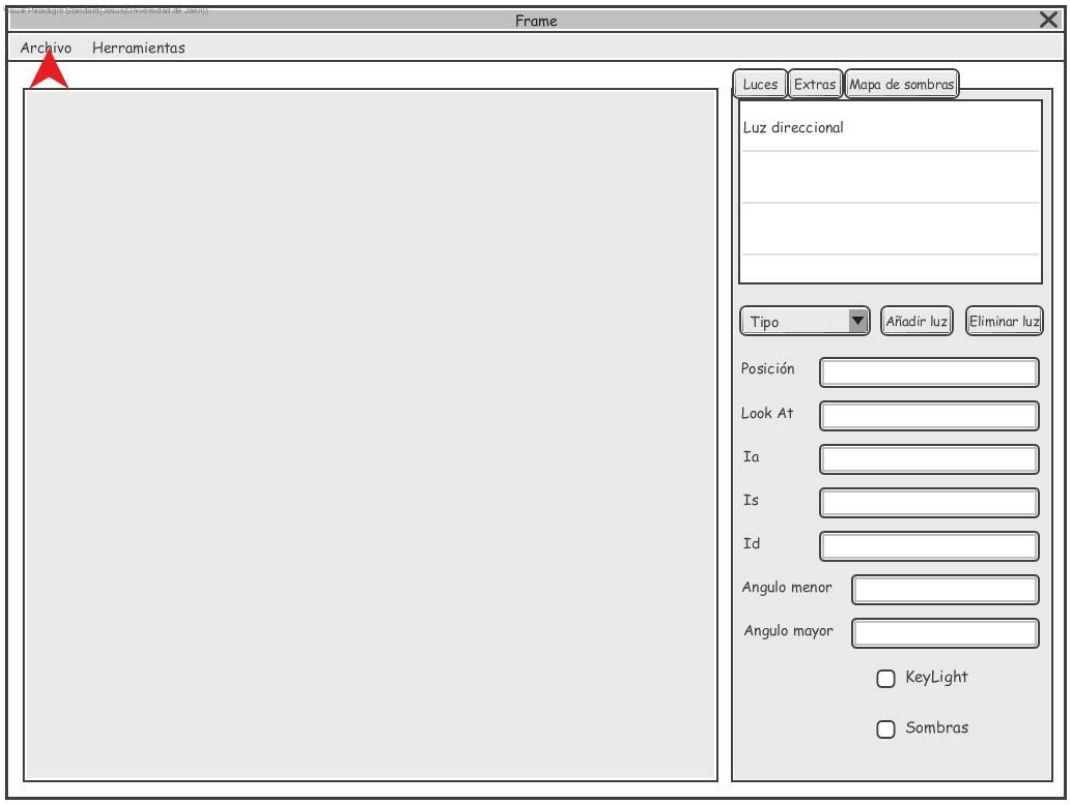


Ilustración 2.74 Storyboards: Cambiar objeto. Desplegar menú "Archivo"

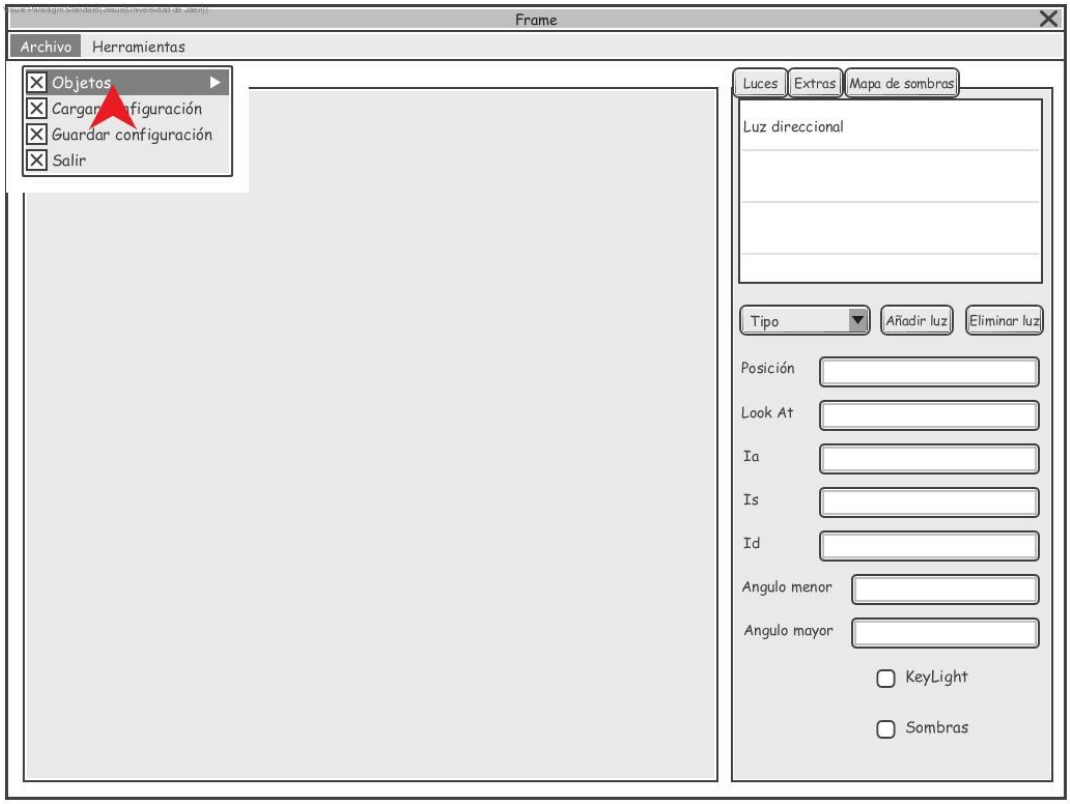


Ilustración 2.75 Storyboards: Cambiar objeto. Selección submenú "Objetos"

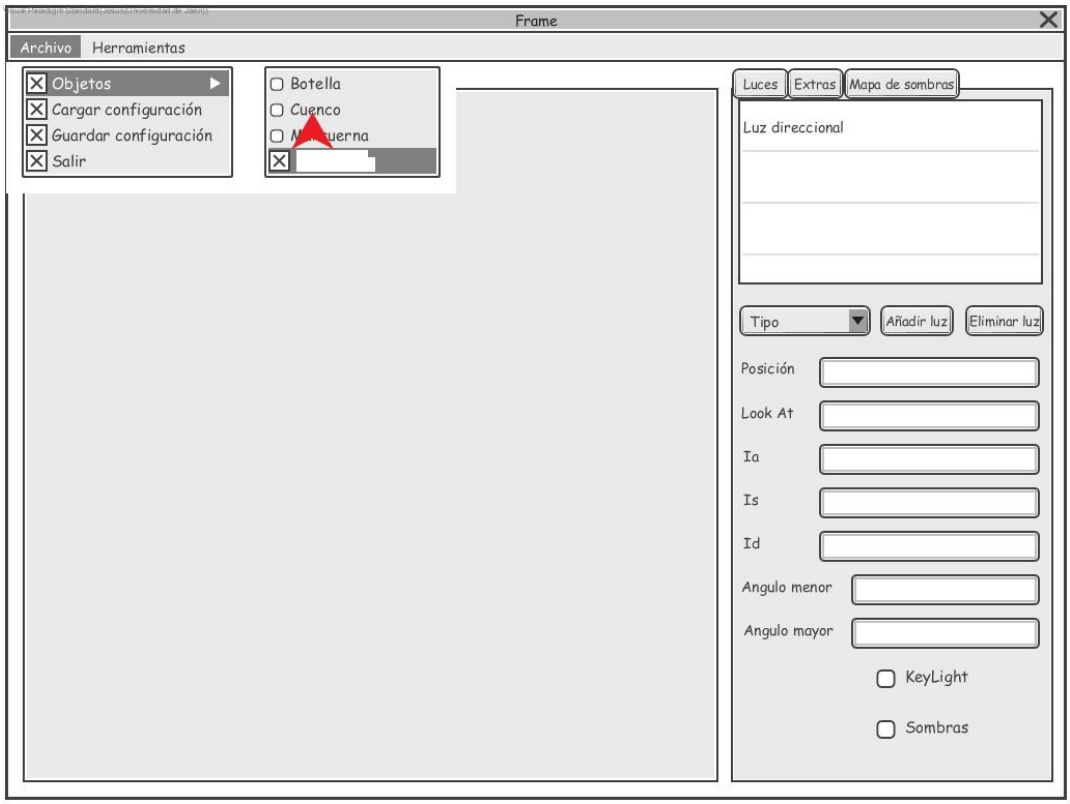


Ilustración 2.76 Storyboards: Cambiar objeto. Selección objeto

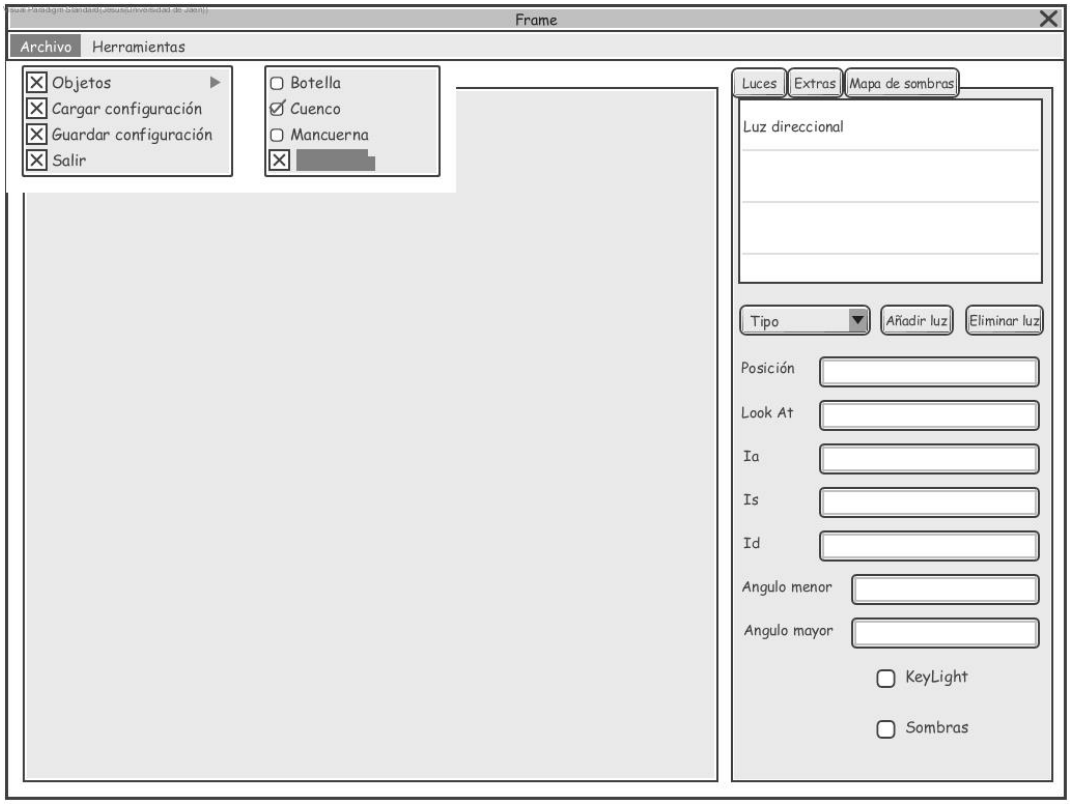


Ilustración 2.77 Storyboards: Cambiar objeto. Objeto seleccionado

- Mover cámara:

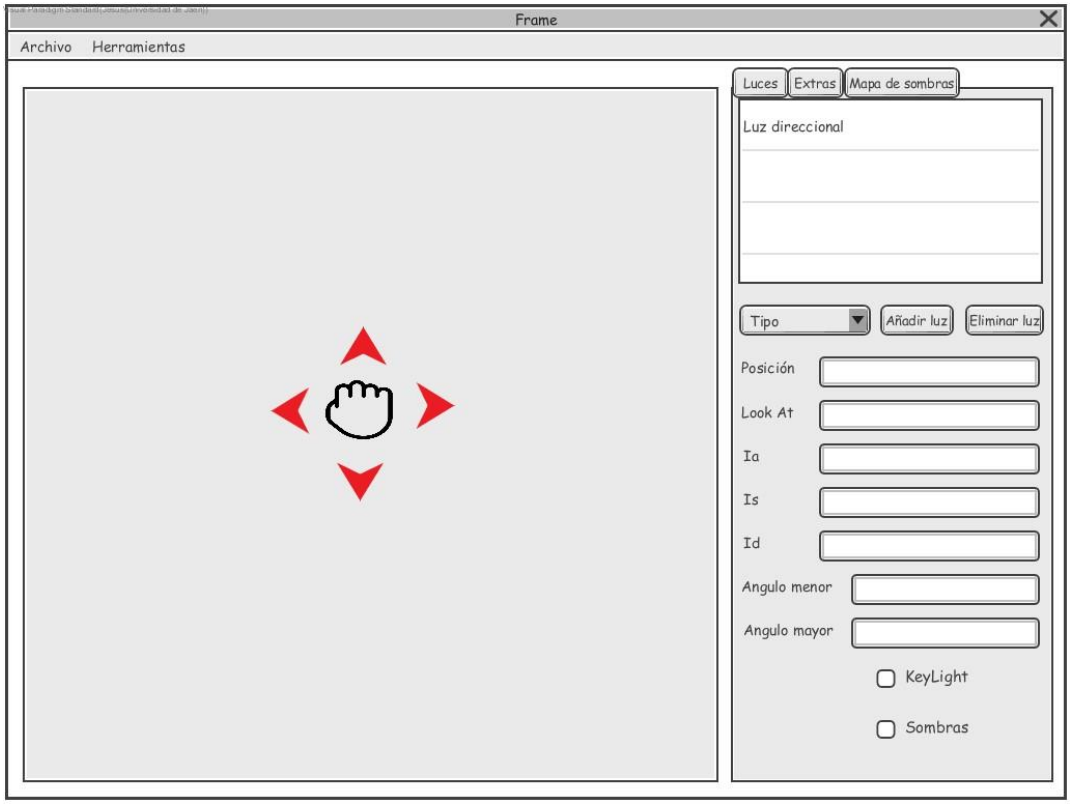


Ilustración 2.78 Storyboards: Mover cámara. Click y arrastre en escena

- Salir de la aplicación:

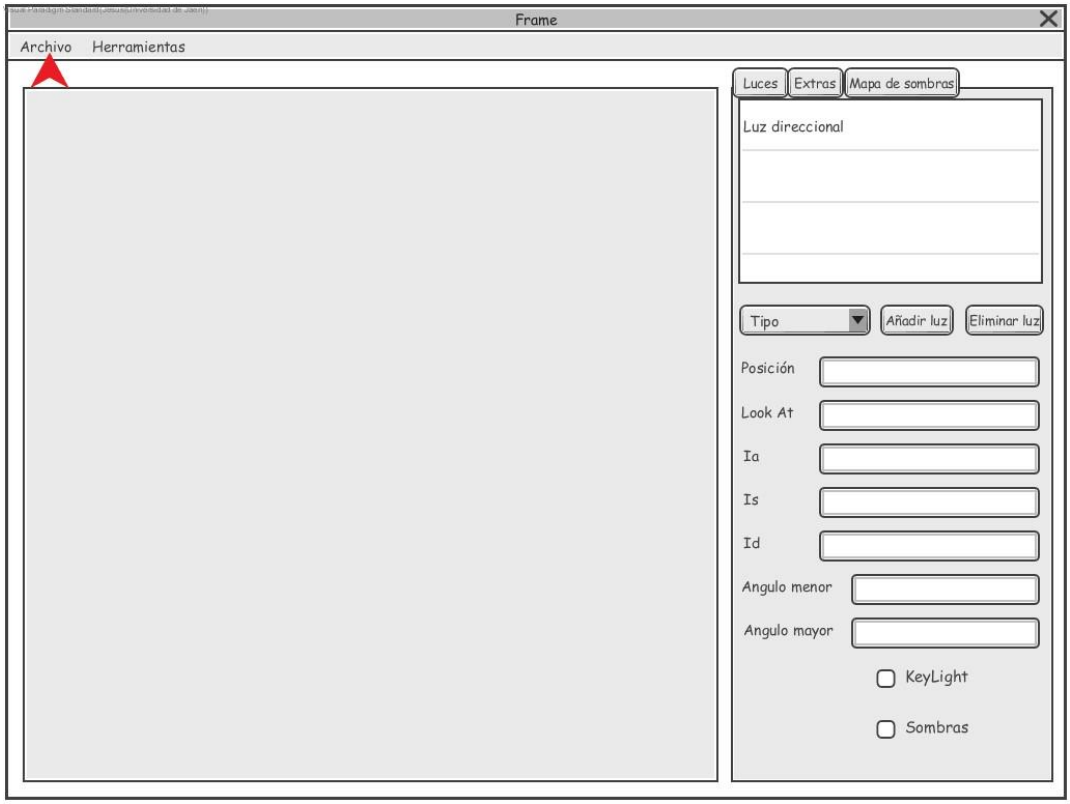


Ilustración 2.79 Storyboards: Salir de la aplicación. Opción 1: Desplegar menú "Archivo".

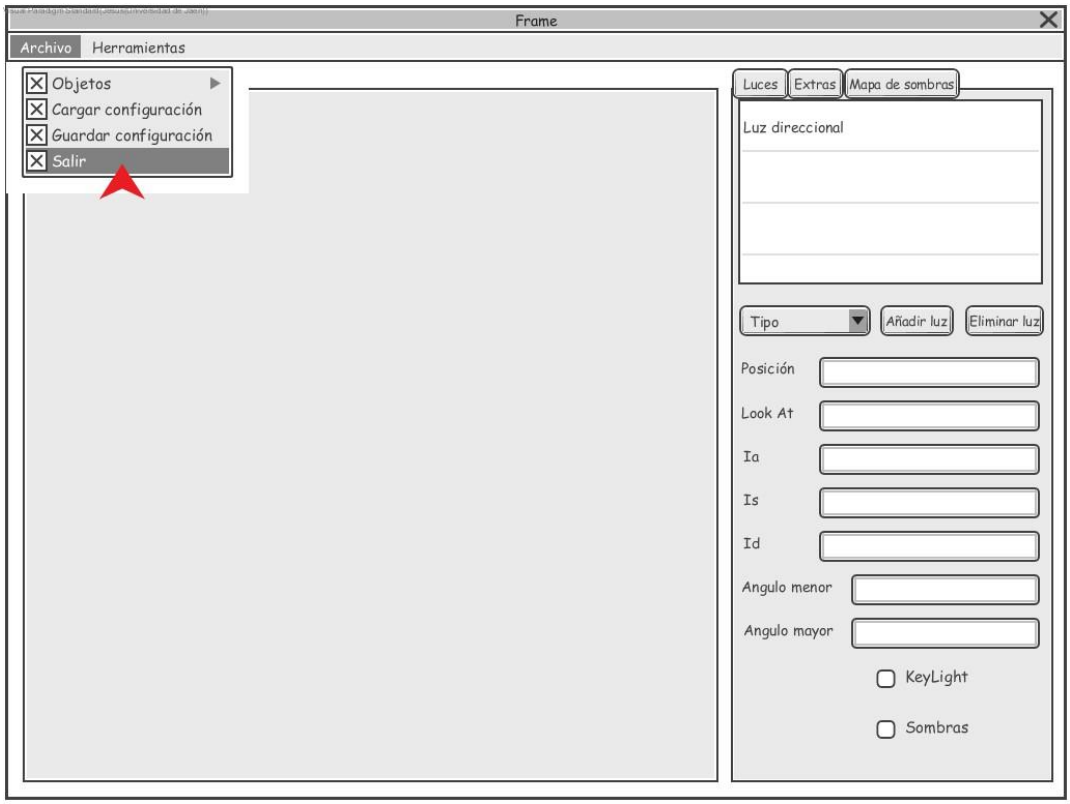


Ilustración 2.80 Storyboards: Salir de la aplicación. Opción 1: Click en salir.

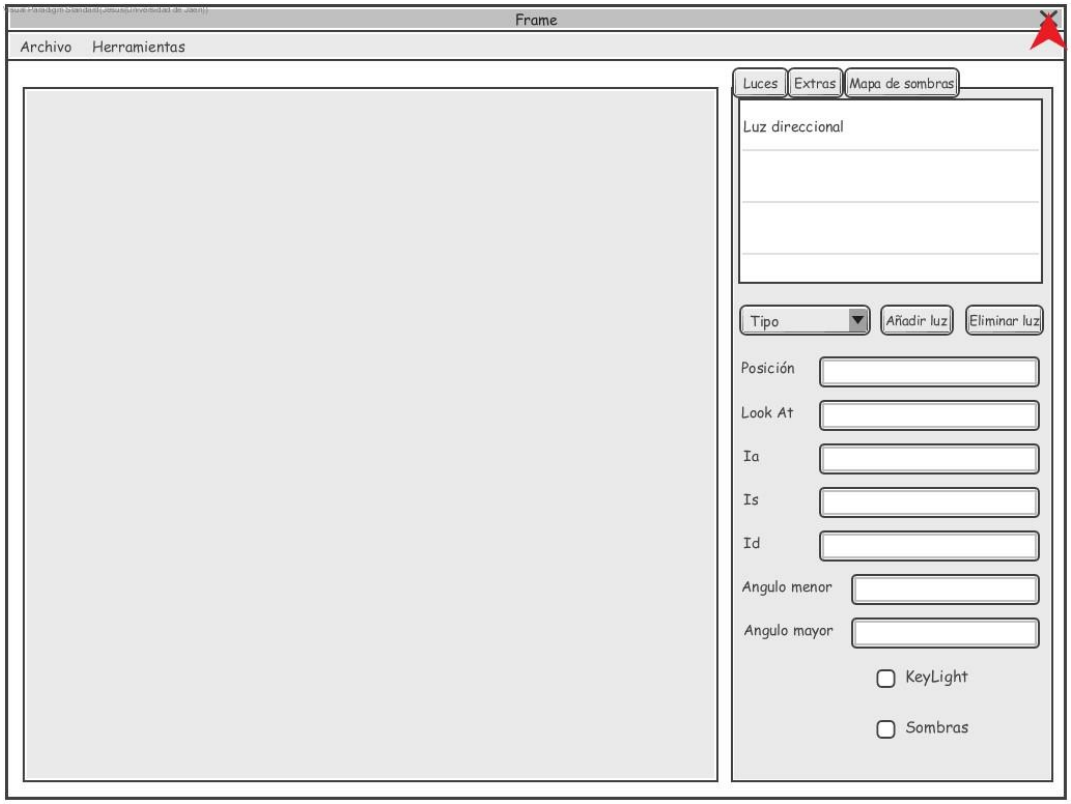


Ilustración 2.81 Storyboards: Salir de la aplicación. Opción 2: Click en la "X".

2.3 - Implementación

La implementación de este proyecto se ha llevado a cabo de la siguiente forma:

En primer lugar, se empezó con la implementación de la ventana de OpenGL que se encuentra en la clase "glwindow". Para poder crear y gestionar una ventana de OpenGL en la plataforma de desarrollo utilizada (Qt) es necesario tener una clase que herede de QOpenGLWidget. Además, para hacer uso de las funciones de OpenGL, Qt también proporciona distintas clases que contienen las funciones de OpenGL organizadas en función de la versión de OpenGL a utilizar, en nuestro caso OpenGL Core 4.1, por lo que nuestra clase "glwindow" también deberá de heredar de QOpenGLFunctions_4_1_Core.

Antes de comenzar a definir los distintos métodos que nos permitirán lanzar la ventana de OpenGL, se comenzó a implementar la clase "glwindow" para hacerla un singleton. Para ello, la clase debe de cumplir una serie de requisitos:

- Su constructor ha de ser privado, para evitar que se puedan crear otras instancias de la misma clase.

```
private:
    glwindow(QWidget *parent = nullptr);
```

Ilustración 2.82 Implementación: Clase glwindow. Constructor glwindow

- Debe de tener un atributo de tipo “static” privado, donde se almacenará la única instancia de nuestra clase creada.

```
static glwindow *instance;
```

Ilustración 2.83 Implementación: Clase glwindow. Instancia glwindow

- Debe tener un método público, también de tipo “static”, que nos permita acceder a dicha instancia si está creada. De no estar creada, este método se encargará de crearla y asignarla a su correspondiente atributo.

```
public:
    static glwindow *getInstance();
```

Ilustración 2.84 Implementación: Clase glwindow. Método static getInstance

Una vez que ya tenemos nuestra clase definida como un singleton, ya podemos proceder a implementar los distintos métodos necesarios para lanzar la ventana de OpenGL.

El primer método necesario es “initializeGL” y es un método heredado de la clase QOpenGLWidget. Este es el primer método al que se llama nada más crear una instancia de la clase “glwindow” y en el cual hay que inicializar todo lo necesario para lanzar la ventana. En nuestro caso debemos de inicializar las funciones de OpenGL Core 4.1 para poder utilizarlas, definir el color de fondo de la ventana, activar el test de profundidad y definir la función de profundidad que se va a utilizar. El test de profundidad nos sirve para decirle a OpenGL que tenga en cuenta la profundidad de los datos a la hora de dibujar. La función que lleva asociada este test nos sirve para definir que píxeles son los que hay que dibujar en pantalla, en nuestro caso, los píxeles que tengan valores de profundidad menores o iguales que los almacenados en el Depth buffer.

```
void glwindow::initializeGL()
{
	//Inicializar todo lo necesario para el renderizado en la ventana
	initializeOpenGLFunctions();
	glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

	// - Le decimos a OpenGL que tenga en cuenta la profundidad a la hora de dibujar.
	// No tiene por qué ejecutarse en cada paso por el ciclo de eventos.
	glEnable(GL_DEPTH_TEST);
	glDepthFunc(GL_LEQUAL);
}
```

Ilustración 2.85 Implementación: Clase glwindow. Método initializeGL

Los siguientes métodos a definir son también heredados de QOpenGLWidget y consisten en resizeGl y paintGL.

El primero es llamado automáticamente cada vez que redimensionamos la ventana de OpenGL y nos servirá para, posteriormente, poder definir nuestro viewport y la matriz de proyección de la cámara.

```
void glwindow::resizeGL(int w, int h)
{
    renderer::getInstance()->redimensionarAreaDibujo(w,h);
    window_refresh();
}
```

Ilustración 2.86 Implementación: Clase glwindow. Método resizeGL

El segundo método es el último método al que se llama en este proceso y consiste en el método que se encarga de dibujar en nuestra ventana.

```
void glwindow::paintGL()
{
	// Draw the scene:
	glClear(GL_COLOR_BUFFER_BIT);
	renderer::getInstance()->dibujar();

}
```

Ilustración 2.87 Implementación: Clase glwindow. Método paintGL

Después, implementamos un método que se encarga de recibir los eventos de ratón que posteriormente nos permitirá mover la cámara. Este método nos lo

proporciona también la clase QOpenGLWidget por lo cual sólo tenemos que sobrecargarlo.

```
bool eventFilter(QObject *obj, QEvent *event) override;
```

Ilustración 2.88 Implementación: Clase glwindow. Método eventFilter

Finalmente, implementamos un método que se encarga de refrescar la escena cuando se produzca cualquier cambio que pueda afectar al dibujado de la misma y de avisar a nuestra clase "renderer" que vamos a definir posteriormente que la escena va a ser refrescada. Para el refresco de la escena se hace uso de una función de QOpenGLWidget llamada "update" que se encarga del refrescado de la misma.

```
void glwindow::window_refresh()
{
    renderer::getInstance()->refreshCallback();
    update();
}
```

Ilustración 2.89 Implementación: Clase glwindow. Método window_refresh

Una vez hemos definido la clase "glwindow" pasamos a definir la clase "renderer". Esta clase va a funcionar como un gestor de nuestra venta de visualización y se va a encargar de todo lo relacionado con el proceso de renderizado. Al igual que "glwindow" esta clase va a funcionar como un singleton, por lo cual hemos tenido que definir los atributos y métodos necesarios para ello.

```
private:
    // - En un singleton el constructor es privado. Esto impide que se puedan construir nuevos renderers aparte del singleton
    renderer();
```

Ilustración 2.90 Implementación: Clase renderer. Constructor renderer

```
// - Este es el singleton, la única instancia de la clase renderer  que se tiene en la aplicación.
static renderer *instance;
```

Ilustración 2.91 Implementación: Clase renderer. Instancia renderer

```
//- Este método de clase permite acceder al singleton. Cada vez que se necesite llamar al renderer se hará a través de este método.
static renderer *getInstance();
```

Ilustración 2.92 Implementación: Clase renderer. Método static getInstance

El primer método que vamos a definir en esta clase será el método “prepareOpenGL, este será el primer método al que llamaremos de la instancia de “renderer” y será el encargado de preparar todo lo necesario para dibujar la escena. Se encargará de cargar y compilar todos los shaders, crear las primeras luces por defecto de la aplicación y activar algunas funciones de OpenGL necesarias como son glPrimitiveRestart y glBlend.

glPrimitiveRestart nos permite definir un valor que, al ser pasado con el conjunto de índices de nuestra geometría a la GPU, le permite a la GPU saber que ese es el último vértice y de esa geometría a generar y debe de empezar otra.

glBlend nos permite activar la mezcla de los valores de color que provienen de los fragmentos, con los valores de color que hay en el color buffer.

Los siguientes métodos a definir son los métodos que serán llamados desde “glwindow” para avisar de la redimensión de la ventana o del refresco de la misma. Estos métodos son “redimensionarAreaDibujo” y “refresh_callback”.

El primero será el encargado de cambiar el viewport y la relación de aspecto de nuestra ventana, así como avisar a la cámara que la relación de aspecto ha cambiado para que pueda recalcular su matriz de proyección.

```
void renderer::redimensionarAreaDibujo(int width, int height)
{
    glViewport(0, 0, width, height);
    this->width = static_cast<float>(width);
    this->height = static_cast<float>(height);
    camaras[camaraActiva].cambiarRelacionAspecto(width, height);
}
```

Ilustración 2.93 Implementación: Clase renderer. Método redimensionarAreaDibujo

El segundo se encarga de limpiar los buffers de color y profundidad para volver a pintar de cero la escena.

```
// - Método que se encarga de redibujar el área OpenGL. Será llamado desde el callback  void window_refresh().
void renderer::refreshCallback()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    camaras[camaraActiva].cambiarRelacionAspecto(width, height);
}
```

Ilustración 2.94 Implementación: Clase renderer. Método refreshCallback

El siguiente método a definir dentro de “renderer” es el método que se encarga de dibujar la escena. Este método se llama “dibujar” y es el último método al que se llama en todo el proceso de dibujado y que se encarga de volver a dibujar toda la escena. Aunque todas las opciones de este método no se llegan a definir completamente hasta que se termina la implementación de algunas clases como “lightsource” y “revolutionobject”, para evitar estar dando salto entre clases voy a explicar directamente el resultado final del método.

Al comienzo del dibujado se comprueba si el usuario tiene activadas las sombras y, de ser así, para cada una de las luces en escena que producen sombras se comprueba si es necesario recalcular su mapa de sombras. El recálculo del mapa de sombras de una luz se produce cuando se modifica algún parámetro que puede afectar a este mapa de sombras como puede ser el ancho y alto del mapa de sombras, la posición o lookAt de la luz, los planos zNear y zFar de la luz o el fogX de la luz. Por lo tanto, en esta parte, para aquellas luces que necesitan un recálculo de su mapa de sombras se recalcula y se almacena en la luz correspondiente.

Para el recálculo del mapa de sombras de una luz hay que seguir una serie de pasos:

En primer lugar, hay que recalcular las matrices de visión y proyección de la luz con las nuevas modificaciones.

```
l->calcularVPMatrix();
```

Ilustración 2.95 Implementación: Clase renderer. Calcular matrices de visión y proyección de la luz

En segundo lugar, hay que decirle a OpenGL que no va a usar el framebuffer principal (el que pinta en pantalla) sino que va a usar otro framebuffer para realizar los cálculos. Este FBO se encuentra almacenado en cada luz y consiste en un FBO que solo va a almacenar valores de profundidad, pues son los valores que necesitamos para hacer nuestro shadow map.

```
glBindFramebuffer(GL_FRAMEBUFFER, l->getShadowFBO2());
glActiveTexture(GL_TEXTURE2);
glBindTexture(GL_TEXTURE_2D, l->getDepthText());
```

Ilustración 2.96 Implementación: Clase renderer. Activar FBO y textura

Después hay que decirle a OpenGL en qué textura se van a guardar esos valores de profundidad y modificar los parámetros de OpenGL necesarios para realizar el cálculo del shadow map en este FBO. Estos parámetros son:

- Limpiar el buffer de profundidad del nuevo FBO.

```
glClear(GL_DEPTH_BUFFER_BIT);
```

Ilustración 2.97 Implementación: Clase renderer. Limpiar buffer de profundidad

- Cambiar el viewport con los valores de ancho y alto del mapa de sombras que se va a generar.

```
glViewport(0, 0, l->getShadowMapWidth(), l->getShadowMapHeight());
```

Ilustración 2.98 Implementación: Clase renderer. Cambiar viewport

- Cambiar la función del test de profundidad a GL_LESS para que sólo tome valores menores a los almacenados en el Depth buffer.

```
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
```

Ilustración 2.99 Implementación: Clase renderer. Test de profundidad

- Activar la funcionalidad de GL_CULL_FACE. Esta funcionalidad nos permite decirle a OpenGL que caras de nuestro objeto queremos que ignore al dibujar.
- Asignar la cara delantera (GL_FRONT) como la cara que debe ignorar OpenGL a la hora de pintar. Esto se hace para evitar el shadow acné a la hora de generar el mapa de sombras. El shadow acné no desaparece, pero queda dentro del objeto por lo que no es visible y produce un mejor resultado.

```
glEnable(GL_CULL_FACE);
glCullFace(GL_FRONT);
```

Ilustración 2.100 Implementación: Clase renderer. Activar cull face y asignar cara frontal.

Después de realizar todos estos cambios, se pinta la escena utilizando un shader específico para la generación del mapa de sombras que explicaré posteriormente.

```
for( auto o : objetosEscena)
{


        QMatrix4x4 lightMVP = vpMatrixLuz * o.second->getModelMatrix();
        shaderGenerarSombras->use();
        shaderGenerarSombras->setUniform("mvpMatrixLuz", lightMVP);
        o.second->drawAsTriangles(shaderGenerarSombras,modelMatrix,l->getVisionMatrixLuz(),l->getProyectionMatrixLuz());

    |

}
```

Ilustración 2.101 Implementación: Clase renderer. Dibujar objetos

Una vez se han recalculado los mapas de sombras necesarios si el usuario tenía activadas las sombras, pasamos al dibujado normal de la escena. Para poder dibujar en nuestra ventana de OpenGL debemos de dibujar sobre el framebuffer principal de la aplicación, este framebuffer está activo por defecto, pero, si hemos realizado el recálculo de los mapas de sombras, lo hemos cambiado por otro FBO, y, por lo tanto, debemos de asegurarnos de volverlo a enlazar para que todas las órdenes que vengan ahora vayan dirigidas a este framebuffer.

```
glBindFramebuffer(GL_FRAMEBUFFER, glwindow::getInstance()->defaultFramebufferObject());
```

Ilustración 2.102 Implementación: Clase renderer. Enlazar framebuffer principal

Además, si hemos recalculado el mapa de sombras, hemos tenido que cambiar algunos parámetros de OpenGL que hay que volver a cambiar para el dibujado normal de la escena:

- Asignar un color de fondo a la escena.

```
glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
```

Ilustración 2.103 Implementación: Clase renderer. Definir color de fondo

- Activar el multisampling. Esto nos permite activar el antialiasing.

```
glEnable(GL_MULTISAMPLE);
```

Ilustración 2.104 Implementación: Clase renderer. Activar multisample

- Limpiar los buffers de color y profundidad para dibujar desde cero la escena.

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

Ilustración 2.105 Implementación: Clase renderer. Limpiar buffers de color y profundidad

- Cambiar la GL_CULL_FACE a GL_BACK para que ahora ignore las caras traseras del objeto a la hora del dibujado.

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```

Ilustración 2.106 Implementación: Clase renderer. Activar cull face y asignar caras traseras

- Cambiar el viewport al viewport de nuestra ventana.

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

Ilustración 2.107 Implementación: Clase renderer. Cambiar viewport

- Cambiar la función de comparación del depth test a la que teníamos al principio: GL_LEQUAL.

```
glDepthFunc(GL_LEQUAL);
```

Ilustración 2.108 Implementación: Clase renderer. Función de profundidad

Una vez hecho esto ya podemos empezar a dibujar nuestra escena. Para ello, tenemos que ir dibujando uno a uno los objetos que aparecerán en nuestra escena con cada aportación que reciben de las luces de la misma. Pero, si el usuario tiene activas las sombras, en esta parte sólo se aplicarán las luces que no produzcan sombras.

En primer lugar, para la primera luz de todas, debemos de definir una función de mezcla como la siguiente:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Ilustración 2.109 Implementación: Clase renderer. Función de blend

Esta función de mezcla hace que los nuevos colores se mezclen con el fondo del framebuffer. El fondo del framebuffer es lo que nosotros hemos definido como glClearColor.

Para el resto de luces la función de mezcla que hay que utilizar es:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

Ilustración 2.110 Implementación: Clase renderer. Función de blend

Esta función de mezcla hace que los nuevos colores se sumen con el contenido del framebuffer.

Posteriormente se enlaza el shader necesario para dibujar la escena de forma normal con la luz que toque en ese momento, y se le pasan los parámetros necesarios para dibujar la escena tal y como el usuario quiere. Además, se le indica a la luz que se va a dibujar que pase sus parámetros necesarios al shader y se dibuja el objeto.

En esta parte hay que destacar que la luz ambiental no utiliza un shader específico como el resto de luces, sino que tiene una aportación al resto de luces y es por ello por lo que siempre se aplica y no es tomada como una luz independiente. De hecho, se recomienda que sólo haya una única luz de este tipo en la escena y que debe de tener una intensidad muy débil.

```
shaderProgram* shader = shadersLuzTexturas.find(tipo)->second;
luz->setVisionMatrix(visionMatrix);
luz->aplicar(shader);
shader->use();
if(atenuacionEnabled)
{
    shader->setUniform("atenuacionEnabled",1);
    shader->setUniform("c1",c1);
    shader->setUniform("c2",c2);
    shader->setUniform("c3",c3);
}else
{
    shader->setUniform("atenuacionEnabled",0);
}
if(fogEnabled && luz->getKeyLight())
{
    shader->setUniform("fogEnabled",1);
    shader->setUniform("zMin",this->zMin);
    shader->setUniform("zMax", this->zMax);

}else
{
    shader->setUniform("fogEnabled",0);
}
luzAmbiental->aplicar(shader);
objeto.second->drawAsTriangles(shader, modelMatrix, visionMatrix, proyectionMatrix);
```

Ilustración 2.111 Implementación: Clase renderer. Aplicar uniforms

Tras el dibujado normal de la escena, puede que sea necesario realizar una segunda pasada de dibujo si el usuario tiene activas las sombras. En ese caso, hay luces que no se han aplicado en el paso anterior puesto que están definidas como luces que producen sombras y, por lo tanto, hay que usar un shader especial que explicaré posteriormente que se encarga de utilizar el mapa de sombras para calcular las sombras que producirían dichas luces.

El proceso a seguir es el mismo, se coge el shader asociado a cada luz y se le pasan los parámetros necesarios a dicho shader y, después, se dibuja la escena con ese shader. Aquí, uno de los parámetros que hay que pasar al shader es el mapa de sombras que tengamos calculado para la luz que se va a aplicar.

Finalmente, para terminar con este método "dibujar", si el usuario tiene activado el uso de las luces fil y rim, su dibujado se realiza en esta última pasada de dibujo. Estas luces también usan un shader distinto al resto y que explicaré posteriormente ya que estas luces no deben de producir brillo y, en el caso de la rim light, tiene un color distinto que no es obtenido del objeto.

El último método a implementar es el método que se encarga de crear las luces fill y rim ("crearFRLights").

Para crear la fill light lo que hay que conseguir es el vector reflejado del vector dirección de la key light con respecto a la normal. Esto se puede hacer mediante la implementación de la función reflect. Después, para calcular la posición de la luz tan solo hay que sumar a ese vector el lookAt de la cámara.

Para la rim light, hay que reflejar la dirección de la luz con el eje Y positivo y, al vector resultante, se le suma también el lookAt de la cámara.

La dirección de las nuevas luces serán sus respectivos vectores de reflexión invertidos y su intensidad será el 40% de la intensidad de la key light.

$$vreflect(x, y, z) = I - 2 * dot(N, I) * N$$

$$Siendo\ I\ el\ vector\ incidente\ y\ N\ el\ vector\ normal\ que\ debe\ estar\ normalizado$$

Ahora vamos a pasar a implementar la clase "camera" que es la clase que se encarga de crear una cámara virtual desde la cuál veré mi escena.

Para definir correctamente una cámara necesitamos su posición, el punto al que mira la cámara, sus planos zNear y zFar, su fovY y la relación de aspecto de la

misma. En nuestro caso los planos zNear y zFar los hemos dejado fijado por defecto para todas las cámaras, pero el resto de parámetros han de ser pasados el constructor de “camera” para poder crear una cámara virtual. Además, el fovY (ángulo vertical) suele ser más difícil de dar para los usuarios ya que lo que normalmente se utiliza es el ángulo horizontal (fovX), es por ello por lo que se decide pasar al constructor de cámara el fovX y, a partir de la siguiente fórmula, calcular el fovY.

$$fovY = 2 * \tan^{-1}(\frac{\tan(\frac{fovX}{2})}{aspect})$$

```
camera::camera(QVector3D position, QVector3D lookAt, float fovX,float width,float height)
{
    this->proyectionMatrix = QMatrix4x4();
    this->visionMatrix = QMatrix4x4();

    this->position = position;
    this->lookAt = lookAt;
    this->width = width;
    this->height = height;
    this->fovX = fovX;
    this->fovY = 2.0f * atan(tan(qDegreesToRadians((fovX / 2.0f ) ) )/ ( width / height ) );
    this->znear = 1.0f;
    this->zfar = 100.0f;

    calcularVectores();
    calcularMatrices();

}
```

Ilustración 2.112 Implementación: Clase camera. Constructor camera

Después de tener todos los parámetros de la cámara es necesario calcular los vectores de la misma para poder moverla luego correctamente por la escena. Estos vectores son los vectores $\hat{n}$,$\hat{u}$,$\hat{v}$.

- El vector $\hat{n}$ se calcula como la posición a la que mira la cámara menos el punto al que mira.
- El vector $\hat{u}$ se calcula como el producto vectorial del eje Y positivo con $\hat{n}$. En el caso en el cual el eje Y positivo sea colineal con $\hat{n}$ se utilizará el eje Z y, en el caso en el cual el eje Y negativo sea colineal con $\hat{n}$ se utilizará el eje -Z.
- Por último, el eje $\hat{v}$ se calcula como el producto vectorial de $\hat{n}$ x $\hat{u}$.

```
void camera::calcularVectores()
{
    //Cálculo de los vectores de la cámara
    this->n = position - lookAt;
    this->n.normalize();
    if (distanciaEpsilon(QVector3D(0, 1, 0), n))
    {
        this->u = QVector3D::crossProduct(QVector3D(0, 0, 1), n);
        this->u.normalize();
    }
    else if (distanciaEpsilon(QVector3D(0, -1, 0), n))
    {
        this->u = QVector3D::crossProduct(QVector3D(0, 0, -1), n);
        this->u.normalize();
    }
    else
    {
        this->u = QVector3D::crossProduct(QVector3D(0, 1, 0), n);
        this->u.normalize();
    }
    this->v = QVector3D::crossProduct(this->n, this->u);
    this->v.normalize();

}
```

Ilustración 2.113 Implementación: Clase camera. Función calcularVectores

Tras esto, calcularemos las matrices de visión y proyección de la cámara. Estas matrices son necesarias pasarlas al vertex shader para proyectar nuestros puntos en el volumen de visión de la cámara. Nos ayudaremos unas funciones que nos proporciona Qt para calcular estas matrices. Para la matriz de visión utilizaremos la función “lookAt” y para la matriz de proyección en perspectiva utilizaremos la función “perspective”.

```
void camera::calcularMatrices()
{
    this->proyectionMatrix.setToIdentity();
    this->visionMatrix.setToIdentity();

    this->visionMatrix.lookAt(this->position, this->lookAt, this->v);
    this->proyectionMatrix.perspective(qRadiansToDegrees(this->fovY), (this->width / this->height), this->znear, this->zfar);
}
```

Ilustración 2.114 Implementación: Clase camera. Función calcularMatrices

Con esto ya tenemos todo lo necesario para poder tener una cámara virtual en la escena desde la cual ver nuestros objetos.

Por último, vamos a implementar un método que se encargue de actualizar los valores de la cámara cuando la relación de aspecto de nuestra ventana cambie y los métodos que nos permitan mover la cámara alrededor del objeto.

El método para actualizar los valores de la cámara se llama "cambiarRelacionAspecto" y se encarga de volver a construir la matriz de proyección, que es la matriz que se ve afectada con el cambio de la relación de aspecto.

```
void camera::cambiarRelacionAspecto(float width, float height)
{
    this->width = width;
    this->height = height;
    this->fovY = 2.0f * atan(tan(qDegreesToRadians((fovX / 2.0f ) ) )/ ( width / height ) );
    this->proyectionMatrix.setToIdentity();
    this->proyectionMatrix.perspective(qRadiansToDegrees(this->fovY), (width / height), 1.0f, 20.0f);
}
```

Ilustración 2.115 Implementación: Clase camera. Función cambiarRelacionAspecto

Los métodos para el movimiento alrededor del objeto son "orbit_vertical" y "orbit_horizontal".

El orbit_horizontal realiza su movimiento rotando el lookAt de la cámara alrededor del eje Y.

```
void camera::orbit_horizontal(float angulo)
{
    QMatrix4x4 transformacion = QMatrix4x4();
    transformacion.translate(-this->lookAt);
    transformacion.rotate(angulo, QVector3D(0, 1, 0));
    transformacion.translate(this->lookAt);

    QVector4D aux;

    aux = QVector4D(this->position, 1) * transformacion;
    this->position = aux.toVector3D();

    aux = QVector4D(this->u, 0) * transformacion;
    this->u = aux.toVector3D();

    aux = QVector4D(this->v, 0) * transformacion;
    this->v = aux.toVector3D();

    aux = QVector4D(this->n, 0) * transformacion;
    this->n = aux.toVector3D();

    calcularMatrices();
}
```

Ilustración 2.116 Implementación: Clase camera. Función orbit_horizontal

En cambio, el orbit_vertical realiza su movimiento rotando el lookAt alrededor del eje $\hat{u}$ de la cámara.

```
void camera::orbit_vertical(float angulo)
{
    QMatrix4x4 transformacion = QMatrix4x4();
    transformacion.translate(-this->lookAt);
    transformacion.rotate(angulo, this->u);
    transformacion.translate(this->lookAt);

    if ( !distanciaEpsilon(QVector3D(0,1,0), QVector3D(QVector4D(this->n, 0) * transformacion)) && !distanciaEpsilon(QVector3D(0, -1, 0), QVector3D(QVector4D(this->n, 0) * transformacion)))
    {

        QVector4D aux;

        aux = QVector4D(this->position, 1) * transformacion;
        this->position = aux.toVector3D();

        aux = QVector4D(this->u, 0) * transformacion;
        this->u = aux.toVector3D();

        aux = QVector4D(this->v, 0) * transformacion;
        this->v = aux.toVector3D();

        aux = QVector4D(this->n, 0) * transformacion;
        this->n = aux.toVector3D();

        calcularMatrices();
    }
}
```

Ilustración 2.117 Implementación: Clase camera. Función orbit_vertical

Ambos movimientos serán realizados cuando el gestor de eventos de la clase “glwindow” detecte un click y arrastre del ratón por parte del usuario sobre la ventana.

Ahora vamos a pasar a definir la implementación de las clases que se encargan de la creación de los objetos junto con su geometría y topología. Estas clases son “element3d”, “revolutionobject” y “plane”.

Empezamos por “element3d”, esta clase contiene los métodos y atributos necesarios para dibujar cualquier objeto 3D.

En lo que respecta a los atributos, todo objeto necesita de una matriz de modelado, que consiste en la matriz que sitúa al objeto en el mundo, y de una textura que lleva asociada. Para la gestión de las texturas implementaremos una clase denominada “textura” que definiremos posteriormente.

```
QMatrix4x4 modelMatrix;
```

Ilustración 2.118 Implementación: Clase element3d. Atributo modelMatrix

```
texture* texturaActual;
```

Ilustración 2.119 Implementación: Clase element3d. Atributo texturaActual

En cuanto a los métodos, esta clase contiene un método virtual para dibujar la escena utilizando mallas de triángulos y que será sobrecargado por “revolutionobject” y “plane” para dibujarse.

```
virtual void drawAsTriangles(shaderProgram* shader, QMatrix4x4 modelMatrix, QMatrix4x4 visionMatrix, QMatrix4x4 proyectionMatrix) = 0;
```

Ilustración 2.120 Implementación: Clase element3d. Función drawAsTriangles

Antes de pasar a implementar las clases “revolutionobject” y “plane” es necesario implementar la clase “subdivisionprofile”. Esta clase parte de una serie de puntos de un perfil con coordenadas (x,y) y aplica un número de subdivisiones al mismo para conseguir suavizar las zonas con curvas. Además, se encarga de comprobar que el perfil cumple con una serie de requisitos para ser un perfil válido como son:

- Debe de haber más de un punto en el perfil.
- De haber 2 puntos en el perfil, al menos uno ha de tener una coordenada x > 0.
- Si hay más de 2 puntos en el perfil, puede haber como máximo 2 puntos con coordenada x = 0.

El pseudocódigo del algoritmo de subdivisión utilizado es el siguiente:

```
Entrada: Una aproximación poligonal P a una curva con n
         puntos
Salida:  Una poligonal subdividida P'

Añadir p0 a P'
Para cada i   1<=i<=n-2
    Añadir (pi-1+pi)/2 a P'
    Añadir (3*pi/4)+(pi-1/8)+(pi+1/8) a P'
Fin para
Añadir (pn-2+pn-1)/2 a P'
Añadir pn-1 a P'
```

Ilustración 2.121 Implementación: Clase subdivisionprofile. Algoritmo subdivisión

Ahora pasamos a implementar la clase “revolutionobject”. Esta clase contendrá toda la información necesaria para dibujar un objeto de revolución. Hereda tanto de “element3d” para sobrecargar el método de dibujado, como de QOpenGLFunctions_4_1_Core para poder usar las funciones de OpenGL.

Partiendo de un perfil de subdivisión válido, esta clase se encarga de aplicar las técnicas de revolución para calcular las posiciones, normales, tangentes y

coordenadas de textura de cada punto del objeto. Todo esto se hace en el constructor del objeto de revolución.

```
revolutionobject(QVector<QVector2D> points, unsigned int subdivisions, unsigned int slices);
```

Ilustración 2.122 Implementación: Clase revolutionobject. Constructor revolutionobject

El cálculo de las posiciones de los puntos se realiza siguiendo el siguiente pseudocódigo:

```
Entrada: Un array P con los puntos pi de la poligonal de
         control (generatriz)
         Un entero S indicando el número de cortes
         longitudinales
Salida:  Un array V con los puntos que forman la geometría
         de la superficie de revolución

delta = 360 / S
Para cada punto pi de la generatriz
    Para cada corte longitudinal s=0, s<=S
        Calcular el ángulo a = s * delta
        Calcular p' revolucionando pi alrededor del eje Y
                  un ángulo a
        Añadir p' al array de vértices V
    Fin para
Fin para
```

16

Ilustración 2.123 Implementación: Clase revolutionobject. Algoritmo de revolución

Para el cálculo de las normales revolucionadas se aplica el mismo algoritmo que para los puntos, pero, primero hay que calcular la normal en cada punto del perfil y, para ello, hay que seguir una serie de pasos:

- Primero hay que calcular el vector entrante a nuestro punto (de haberlo, en el caso del primer punto del perfil no existe).

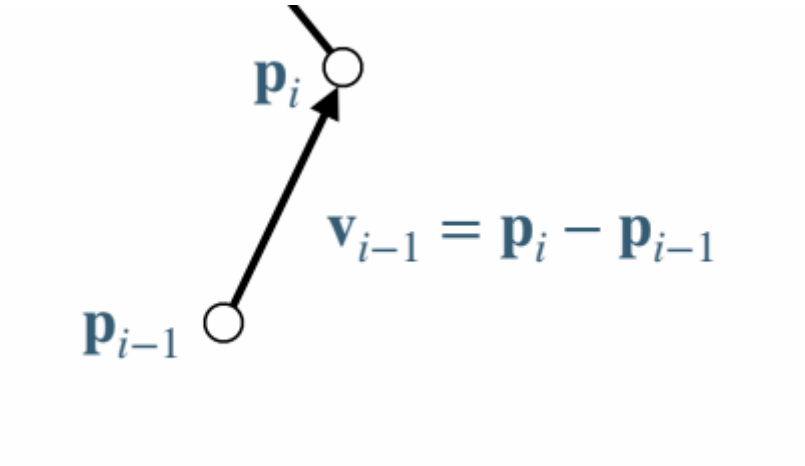


Ilustración 2.124 Implementación: Clase revolutionobject. Vector entrante

- Después se calcula el vector saliente a nuestro punto (de haberlo, en el caso del último punto del perfil no existe).

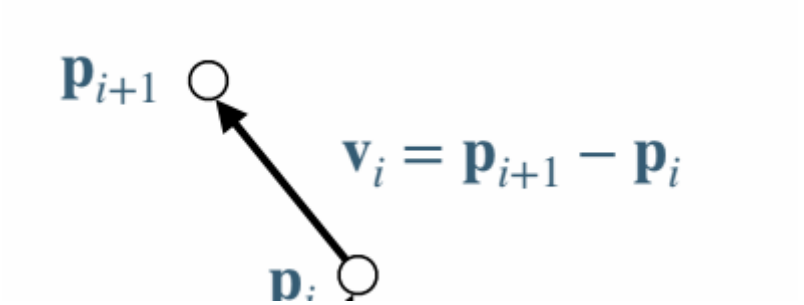


Ilustración 2.125 Implementación: Clase revolutionobject. Vector saliente

- Posteriormente estos 2 vectores se rotan -90 grados para que apunten de forma perpendicular hacia la parte externa del objeto.

$$\mathbf{R}(\alpha) = \begin{bmatrix} cos\ \alpha & -sin\ \alpha & 0 \\ sin\ \alpha & cos\ \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\hat{\mathbf{ns}}_i = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_{x_i} \\ v_{y_i} \\ 0 \end{bmatrix} = \begin{bmatrix} v_{y_i} \\ -v_{x_i} \\ 0 \end{bmatrix}$$

Ilustración 2.126 Implementación: Clase revolutionobject. Rotación de vectores

- Por último, se hace la media de estos 2 vectores y esa será la normal en el punto. En el caso del primer y último punto este paso se obvia porque no existen 2 vectores para hacer la media. En ese caso se coge ese vector rotado como el vector normal.

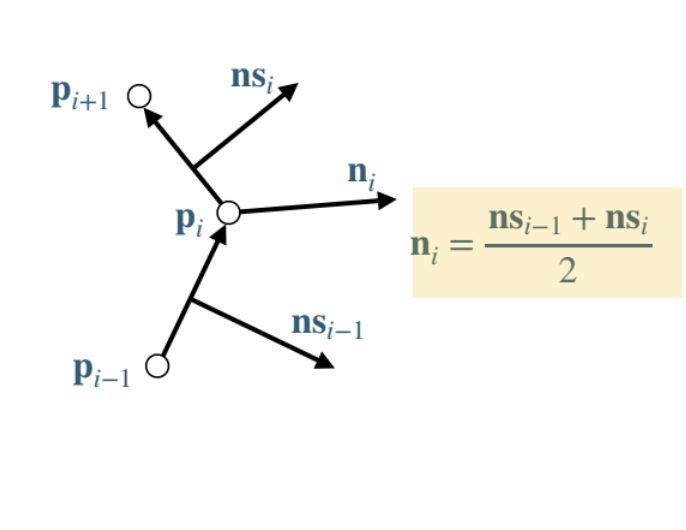


Ilustración 2.127 Implementación: Clase revolutionobject. Cálculo de la normal

En esta parte hay que destacar que la normal de la tapa inferior es siempre (0,-1,0) y la normal de la tapa superior es (0,1,0). Las normales que se revolucionan son las pertenecientes al cuerpo.

Para revolucionar las tangentes, en el caso del cuerpo, se revolucionan a la vez que los puntos, pero su fórmula es:

$$\hat{\mathbf{t}} = (0{,}0, -1)$$

$$\alpha = s\Delta$$

$$\Delta = \frac{2\pi}{S}$$

$$\hat{\mathbf{t}}(s) = \begin{bmatrix} cos\ \alpha & 0 & sin\ \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -sin\ \alpha & 1 & cos\ \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} -sin\ \alpha \\ 0 \\ -cos\ \alpha \\ 0 \end{bmatrix}$$

$$\hat{\mathbf{t}}(s) = (-sin(s\Delta), 0, -cos(s\Delta)) \quad 0 \le s \le S$$

Ilustración 2.128 Implementación: Clase relovutionobject. Revolución de tangentes

En el caso de existir la tapa superior la tangente se calcula como el producto vectorial de la normal en el punto con el eje Z positivo.

En el caso de existir tapa inferior la tangente se calcula como el producto vectorial de la normal en el punto con el eje Z negativo.

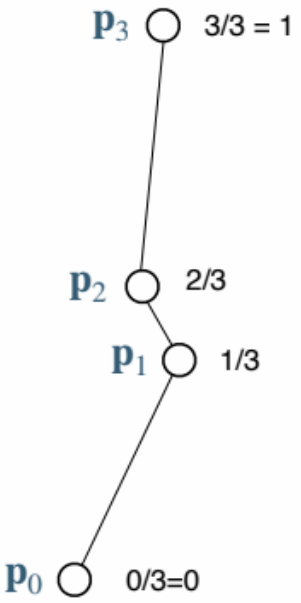
Lo último que hay que calcular son las coordenadas de textura. Estas coordenadas tienen 2 dimensiones (u,v).

Para el caso del cuerpo, la coordenada u se calcula:

$$u(s) = s\frac{1}{S} \qquad 0 \le s \le S$$

Ilustración 2.129 Implementación: Clase revolutionobject. Coordenada u de textura

Y la coordenada v:



$$v(i) = i\frac{1}{n-1} \qquad 0 \le i < n$$

Ilustración 2.130 Implementación: Clase revolutionobject. Coordenada v de textura

En el caso de la tapa superior, las coordenadas u y v de textura se calculan:

$$textura(u,v) = (\frac{\cos a}{2} + 0.5, \frac{\sin a}{2} + 0.5)$$

Excepto en el punto central que las coordenadas son (0.5,0.5)

En el caso de la tapa inferior, las coordenadas u y v de textura se calculan:

$$textura(u,v) = (\frac{\cos a}{2} + 0.5, \frac{-\sin a}{2} + 0.5)$$

Excepto en el punto central que las coordenadas son (0.5,0.5).

El motivo de aplicar esta fórmula es que las coordenadas de textura van en el rango [0,1] pero, las funciones seno y coseno van en el rango [-1,1] por lo que tengo que aplicar la división por 2 y sumarle 0.5 para meterla en mi rango [0,1].

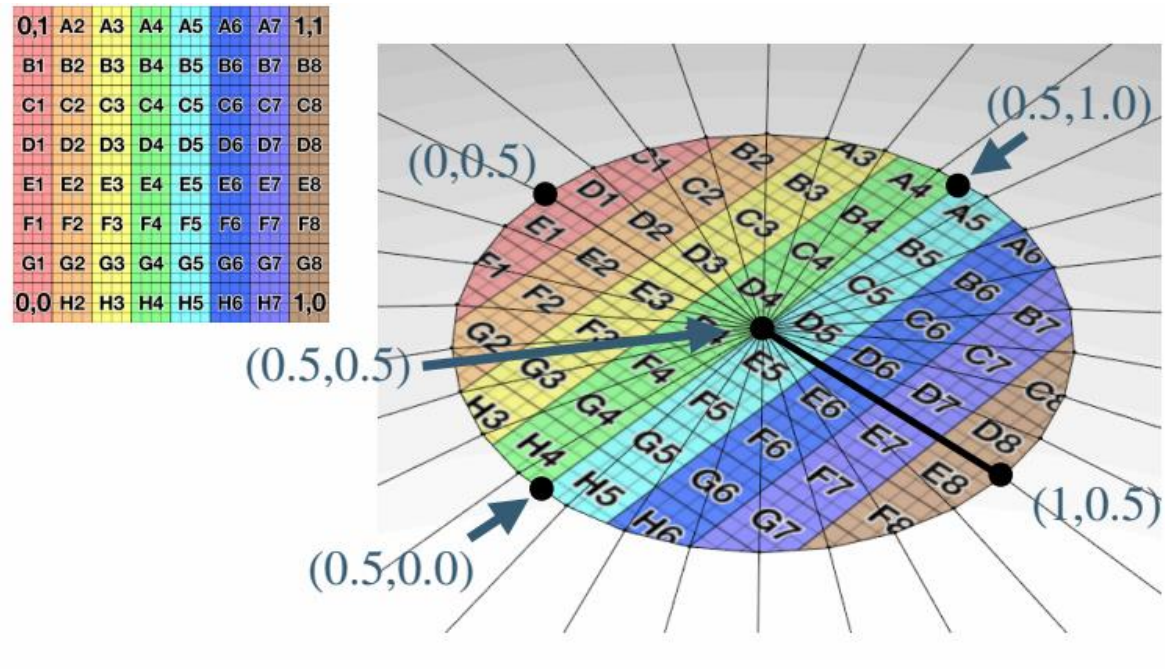


Ilustración 2.131 Implementación: Clase revolutionobject. Coordenadas de textura

Ya tenemos toda la geometría creada y almacenada. Ahora lo que necesitamos es calcular la topología y ya tendríamos todo lo necesario para poder dibujar nuestro objeto.

Para la topología del cuerpo debemos de aplicar el siguiente pseudocódigo:

```
Entrada: Un entero S indicando el número de cortes
         longitudinales o slices y un entero n indicando
         el número de puntos que tiene la poligonal
         generatriz
Salida:  Un array de enteros I con los índices que
         conforman la topología de la malla

Para cada corte longitudinal s=0, s<S
    Para cada elemento de la generatriz i=0, i<n
        Añadir (i*(S+1))+s al array de índices I
        Añadir (i*(S+1))+(s+1) al array de índices I
    Fin para
    Añadir 0xFFFF al array de índices I
Fin para
```

Ilustración 2.132 Implementación: Clase revolutionobject. Algoritmo topología

Para la topología de las tapas, en el caso de la tapa superior, se da primero el índice del último punto y, después, se van dando los índices de los puntos en sentido antihorario.

En el caso de la tapa inferior, se da primero el índice del último punto y, después, se van dando los índices de los puntos en sentido horario.

Una vez que ya tenemos la geometría y la topología de nuestro objeto, hay que pasar todos los datos a sus correspondientes VAO, VBO e IBO. Esto se hace en las funciones “rellenarVBO” y “rellenarIBO” que son llamadas al final del constructor de “revolutionobject”.

Como nuestro objeto puede contener hasta 3 mallas distintas, podremos tener hasta tres VAO, cada uno con la información de una malla. Para la gestión de los VAO deberemos de implementar una clase denominada “vao” que posteriormente definiré.

Los pasos para rellenar los VBO con la información necesaria son:

- Enlazar el VAO de la malla a la que queremos pasar la información.
- Enlazar el VBO de los datos que queremos pasar.
- Activar un vertex attribute array (location).
- Pasar la información al VBO.

Estos pasos habría que repetirnos hasta rellenar todos los VBO que contiene cada VAO. En nuestro caso existen 3 VBO: vboPosNorm, que almacena información entrelazada de posiciones y normales, vboTangentes, que almacena información de las tangentes y vboTexturas, que almacena información de las texturas.

```
this->vaoCuerpo->enlazarVAO();
this->vaoCuerpo->enlazarVBO(POSNORM);

// - Aquí se indica que uno de los elementos del array entrelazado va asociado con el layout (location=0) en el shader, en concreto la posición
glEnableVertexAttribArray(0);
// - Aquí se describen las características del puntero que permite a la GPU acceder a las posiciones
glVertexAttribPointer(0, sizeof(QVector3D) / sizeof(GLfloat), GL_FLOAT, GL_FALSE, sizeof(PosNorm), ((GLubyte *)NULL + (0)));

// - Como es un array entrelazado, hay que repetir el proceso para los demás elementos, en este caso para la normal
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, sizeof(QVector3D) / sizeof(GLfloat), GL_FLOAT, GL_FALSE, sizeof(PosNorm), ((GLubyte *)NULL + (sizeof(QVector3D))));
glBufferData(GL_ARRAY_BUFFER, posNormCuerpo.size() * sizeof(PosNorm), posNormCuerpo.data(), GL_STATIC_DRAW);

this->vaoCuerpo->enlazarVBO(TANGENTES);
// - Aquí se indica que las tangentes van a ir en el layout(location = 2) del shader
glEnableVertexAttribArray(2);
// - Aquí se describen las características del puntero que permite a la GPU acceder a las tangentes
glVertexAttribPointer(2, sizeof(QVector3D) / sizeof(GLfloat), GL_FLOAT, GL_FALSE, sizeof(QVector3D), ((GLubyte *)NULL + (0)));
glBufferData(GL_ARRAY_BUFFER, tangentesCuerpo.size() * sizeof(QVector3D), tangentesCuerpo.data(), GL_STATIC_DRAW);

this->vaoCuerpo->enlazarVBO(TEXTURAS);
// - Aquí se indica que las coordenadas de texturas van a ir en el layout(location = 3) del shader
glEnableVertexAttribArray(3);
// - Aquí se describen las características del puntero que permite a la GPU acceder a las coordenadas de textura
glVertexAttribPointer(3, sizeof(QVector2D) / sizeof(GLfloat), GL_FLOAT, GL_FALSE, sizeof(QVector2D), ((GLubyte *)NULL + (0)));
glBufferData(GL_ARRAY_BUFFER, texturasCuerpo.size() * sizeof(QVector2D), texturasCuerpo.data(), GL_STATIC_DRAW);
```

Ilustración 2.133 Implementación: Clase revolutionobject. Rellenar VBOs

El fragmento de código pertenece a la malla del cuerpo, en el caso de haber tapa superior y tapa inferior se haría de la misma forma, pero habría que enlazar sus respectivos VAO y VBO.

```
this->vaoTapaI->enlazarVAO();
this->vaoTapaI->enlazarVBO(POSNORM);
```

Ilustración 2.134 Implementación: Clase revolutionobject. VBO tapa inferior

```
this->vaoTapaS->enlazarVAO();
this->vaoTapaS->enlazarVBO(POSNORM);
```

Ilustración 2.135 Implementación: Clase revolutionobject. VBO tapa superior

Por último, los pasos para rellenar los IBO son:

- Enlazar el VAO de la malla a la que queremos pasarle la información.

```
this->vaoCuerpo->enlazarVAO();
```

Ilustración 2.136 Implementación: Clase revolutionobject. Enlazar VAO cuerpo

- Enlazar el IBO correspondiente.

```
this->vaoCuerpo->enlazarIBO(MALLATRIANGULOS);
```

Ilustración 2.137 Implementación: Clase revolutionobject. Enlazar IBO cuerpo

- Introducir la información de topología en el IBO.

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER, indicesCuerpoNM.size() * sizeof(GLuint), indicesCuerpoNM.data(), GL_STATIC_DRAW);
```

Ilustración 2.138 Implementación: Clase revolutionobject. Dibujar

El fragmento de código pertenece a la malla del cuerpo, en el caso de haber tapa superior y tapa inferior se haría de la misma forma, pero habría que enlazar sus respectivos VAO e IBO.

```
this->vaoTapaI->enlazarVAO();
```

Ilustración 2.139 Implementación: Clase revolutionobject. Enlazar VAO tapa inferior

```
this->vaoTapaI->enlazarIBO(NUBEPUNTOS);
```

Ilustración 2.140 Implementación: Clase revolutionobject. Enlazar IBO tapa inferior

```
this->vaoTapaS->enlazarVAO();
```

Ilustración 2.141 Implementación: Clase revolutionobject. Enlazar IBO tapa superior

```
this->vaoTapaS->enlazarIBO(MALLATRIANGULOS);
```

Ilustración 2.142 Implementación: Clase revolutionobject. Enlazar IBO tapa superior

Para finalizar, paso a explicar el método sobrecargado de “drawAsTriangles” que es el encargado de dibujar el objeto.

Este método recibe un shader, una matriz de modelado, y las matrices de visión y proyección de la cámara.

Los pasos para dibujar el objeto son:

- Activar el shader.

```
shader->use();
```

Ilustración 2.143 Implementación: Clase revolutionobject. Activar shader

- Pasar los uniforms necesarios.

```
QMatrix4x4 nuevaModelMatrix = modelMatrix * this->modelMatrix;
QMatrix4x4 modelVisionProymatrix = proyectionMatrix * visionMatrix*nuevaModelMatrix;
shader->setUniform("mvpMatrix", modelVisionProymatrix);
shader->setUniform("mModelView", visionMatrix*nuevaModelMatrix);
```

Ilustración 2.144 Implementación: Clase revolutionobject. Paro de uniforms

- Aplicar la textura.

```
this->texturaActual->aplicar(shader);
```

Ilustración 2.145 Implementación: Clase revolutionobject. Aplicar textura

- Enlazar el VAO a dibujar (en el caso de tener varios VAO para dibujar, se repetiría desde este paso para cada VAO).

```
this->vaoCuerpo->enlazarVAO();
```

Ilustración 2.146 Implementación: Clase revolutionobject. Enlazar VAO cuerpo

- Enlazar la topología con el IBO a utilizar.

```
this->vaoCuerpo->enlazarIBO(MALLATRIANGULOS);
```

Ilustración 2.147 Implementación: Clase revolutionobject. Enlazar IBO cuerpo

- Dibujar la malla pasando el tamaño del array de índices que se va a utilizar.

```
glDrawElements(GL_TRIANGLE_STRIP, indicesCuerpoNM.size(), GL_UNSIGNED_INT, NULL);
```

Ilustración 2.148 Implementación: Clase revolutionobject. Dibujar

Por último, vamos a implementar la clase “plane” y ya tendríamos definidas todas las clases que se encargan de crear los objetos que se visualizarán en la escena. Esta clase también hereda tanto de “element3d” para sobrecargar el método de dibujado, como de QOpenGLFunctions_4_1_Core para poder usar las funciones de OpenGL.

Un plano puede dibujarse mediante una única malla de triángulos, por lo cual, solo necesitaremos un VAO con sus respectivos VBOs y un IBO para el dibujo en malla de triángulos. El proceso para rellenar estas estructuras es idéntico al de los objetos de revolución, al igual que pasa con el proceso de dibujado, por lo cual paso a explicar el proceso para crear la geometría y topología del plan que si es distinto.

Para construir un plano es necesario saber el alto, el acho y el número de divisiones que se quiera tener tanto a lo ancho como a lo alto. Estos son los parámetros necesarios para su constructor.

```
plane(float ancho, float alto, unsigned int divX, unsigned int divZ);
```

Ilustración 2.149 Implementación: Clase plane. Constructor plane

Sabiendo el número de divisiones en cada dirección y el ancho y alto del plano, se puede calcular la posición donde iría cada punto. Tan solo habría que ir recorriendo el plano como si de una matriz se tratase e ir colocando los puntos en cada posición a la que nos movemos.

Para la normal de los puntos no habría ningún problema ya, que al ser un plano que actúa de suelo, la normal en todos los puntos iría en el eje Y positivo, (0,1,0).

Para las tangentes del plano tampoco habría problema ya que al ser un plano que no está inclinado todas las tangentes apuntarían al eje X positivo, (1,0,0).

Para las coordenadas de textura tan sólo habría que calcular en qué porción del plano nos encontramos siendo la coordenada.

Si nos encontramos en el primer punto, el punto (0,0), su coordenada de textura sería: $textura(x,y) = (\frac{0}{ancho}, \frac{0}{alto})$. Es decir, la (0,0).

Si nos encontramos en el extremo superior derecho, el punto (ancho, alto), su coordenada de textura sería: $textura(x, y) = (\frac{ancho}{ancho}, \frac{alto}{alto})$. Es decir, la (1,1).

```
float ini_x = 0.0;
float ini_z = 0.0;
float tam_x = ancho;
float tam_z = alto;

float avanzarX = tam_x / divX;
float avanzarZ = tam_z / divZ;
float sumaX = 0;
float sumaZ = 0;

for (float i = 0.0; i <= divZ; i++) {
    for (float j = 0.0; j <= divX; j++) {
        texturasPlane.push_back(QVector2D(sumaX / tam_x, (sumaZ / tam_z) ));
        PosNorm posNorm;
        posNorm.position = QVector3D(sumaX, 0, -sumaZ);
        posNorm.normal = QVector3D(0, 1, 0);
        posNormPlane.push_back(posNorm);
        tangentesPlane.push_back(QVector3D(1, 0, 0));
        sumaX += avanzarX;
    }
    sumaZ += avanzarZ;
    sumaX = 0;
}
```

Ilustración 2.150 Implementación: Clase plane. Creación del plano

Para el cálculo de los índices se utiliza el mismo algoritmo que en el caso de los objetos de revolución, Siendo “S” las divisiones en el eje X, y “n” las divisiones en el eje Z.

```
Entrada: Un entero S indicando el número de cortes
         longitudinales o slices y un entero n indicando
         el número de puntos que tiene la poligonal
         generatriz
Salida:  Un array de enteros I con los índices que
         conforman la topología de la malla

Para cada corte longitudinal s=0, s<S
    Para cada elemento de la generatriz i=0, i<n
        Añadir (i*(S+1))+s al array de índices I
        Añadir (i*(S+1))+(s+1) al array de índices I
    Fin para
    Añadir 0xFFFF al array de índices I
Fin para
```

Ilustración 2.151 Implementación: Clase plane. Algoritmo topología plano

```
void plane::calcularIndicesNM(unsigned int divX, unsigned int divZ)
{
    for ( unsigned int s = 0; s < divX; s++) {
        for ( unsigned int i = 0; i <= divZ; i++)
        {
            GLuint indice1 = (i* (divX + 1)) + s;
            GLuint indice2 = (i* (divX + 1)) + (s + 1);
            indicesPlaneNM.push_back(indice1);
            indicesPlaneNM.push_back(indice2);
        }
        indicesPlaneNM.push_back(FINAL);
    }
}
```

Ilustración 2.152 Implementación: Clase plane. Método calcularIndicesNM

Ahora que tenemos implementadas las clases que se encargan de crear nuestros objetos de la escena, vamos a implementar algunas clases de las que éstos hacen uso. Durante el proceso de implementación estas clases se han ido creando en paralelo junto con “revolutionobject” y “plane” conforme han ido haciendo falta, pero, por seguir una organización y no ir mezclando código y explicaciones de distintas clases, voy a definirlas ahora.

Empezamos con la clase “texture”. Esta clase se encargará de crear una textura haciendo uso de la clase de Qt QOpenGLTexture.

En el constructor pasaremos las rutas de las imágenes de las cuales queremos hacer textura y el tipo de textura que queremos hacer con cada imagen.

```
texture(QVector<QString> rutas, QVector<TexturesTypes> tipo);
```

Ilustración 2.153 Implementación: Clase texture. Constructor texture

Mediante las funciones de QOpenGLTexture podemos crear la textura a partir de una QImagen y asociarle los parámetros que queramos a dichas texturas.

```
QString filename = rutas[i];
TexturesTypes tipoTextura = tipo[i];
QImage imagen;
imagen.load(filename);
```

Ilustración 2.154 Implementación: Clase texture. Carga textura

```
QOpenGLTexture *texture = new QOpenGLTexture(imagen.mirrored());
```

Ilustración 2.155 Implementación: Clase texture. Creación textura

```
case COLOR:
{
    color = texture;
    color->setMinificationFilter(QOpenGLTexture::LinearMipMapLinear);
    color->setMagnificationFilter(QOpenGLTexture::Linear);
    color->setWrapMode(QOpenGLTexture::DirectionS,QOpenGLTexture::ClampToEdge );
    color->setWrapMode(QOpenGLTexture::DirectionT,QOpenGLTexture::ClampToEdge );
    color->generateMipMaps();
    break;
}
```

Ilustración 2.156 Implementación: Clase texture. Creación textura color

Después, esta clase implementa un método que se encarga de enlazar las texturas que tenga creadas a una unidad de textura y pasar el uniform necesario al shader cuando el objeto al que está asociada se lo pide.

```
void texture::aplicar(shaderProgram* shader)
{
    shader->use();
    if (color != nullptr)
    {
        shader->setUniform("TexSamplerColor", 0);
        color->bind(0);
    }
    if (brillo != nullptr)
    {
        shader->setUniform("TexSamplerBrillo", 1);
        brillo->bind(1);
    }
}
```

Ilustración 2.157 Implementación: Clase texture. Método aplicar

Para poder gestionar mejor todas las texturas, se ha implementado una clase denominada “bibliotexture” que actúa como un singleton. De esta forma tenemos una clase que actúa como una biblioteca de texturas pudiendo cargar todas las texturas desde el primer momento y, cuando se vaya a asignar una textura a un objeto, sólo habría que buscarla en esta biblioteca y no sería necesario crearla de nuevo.

```
class bibliotexture
{
public:
    ~bibliotexture();

    static bibliotexture *getInstance();
    //Método para añadir una nueva textura a la biblioteca
    void anadirTextura(texture* textura, std::string name);
    //método para eliminar una textura de la biblioteca
    void eliminarTextura(std::string name);
    //Método para buscar una textura en la biblioteca
    texture* buscarTextura(std::string name);
private:
    // - En un singleton el constructor es privado.
    bibliotexture();

    // - Este es el singleton, la única instancia de la clase biblioTexture que se tiene en la aplicación.
    static bibliotexture *instance;
    QMap<std::string, texture*> texturas;

};
```

Ilustración 2.158 Implementación: Clase bibliotexture

La última clase de la que hacen uso “revolutionobject” y “plane” es la clase “vao”.

Esta clase contiene un atributo de la clase QOpenGLVertexArrayObject que es la clase que ofrece Qt para crear y gestionar los VAO y varios QOpenGLBuffers que es la clase que ofrece Qt para crear y gestionar los VBOs e IBOs.

```
QOpenGLVertexArrayObject* Vao;

QOpenGLBuffer* vboPosNorm;
QOpenGLBuffer* vboTexturas;
QOpenGLBuffer* vboTangentes;
```

Ilustración 2.159 Implementación: Clase vao. VAo y VBOs

```
QOpenGLBuffer* iboMallaTriangulos;
```

Ilustración 2.160 Implementación: Clase vao. IBO

Cuando un objeto se crea una insyancia de la clase “vao”, el constructor de esta clase automáticamente un VAO de la clase QOpenGLVertexArrayObject y crea los VBOs e IBOs necesarios para poder dibujar dicho objeto.

```
vao::vao()
{
    this->Vao = new QOpenGLVertexArrayObject();
    this->Vao->create();
    generarVBOPosNorm();
    generarVBOTangentes();
    generarVBOTexturas();
    generarIBONubePuntos();
    generarIBOAlambre();
    generarIBOMallaTriangulos();

}
```

Ilustración 2.161 Implementación: Clase vao. Constructor VAO

```
void vao::generarVBOPosNorm()
{
    //Crear VBO
    this->vboPosNorm = new QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);
    vboPosNorm->create();

}
```

Ilustración 2.162 Implementación: Clase vao. Método generarVBOPosNotm

```
void vao::generarVBOTexturas()
{
    //Crear VBO
    this->vboTexturas = new QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);
    vboTexturas->create();
}
```

Ilustración 2.163 Implementación: Clase vao. Método generarVBOTexturas

```
void vao::generarVBOTangentes()
{
    //Crear VBO
    this->vboTangentes = new QOpenGLBuffer(QOpenGLBuffer::VertexBuffer);
    vboTangentes->create();
}
```

Ilustración 2.164 Implementación: Clase vao. Método generarVBOTangentes

```
void vao::generarIBOMallaTriangulos()
{
    //Crear IBO;
    this->iboMallaTriangulos = new QOpenGLBuffer(QOpenGLBuffer::IndexBuffer);
    iboMallaTriangulos->create();
}
```

Ilustración 2.165 Implementación: Clase vao. Método generarIBOMallaTriangulos

Además, tiene implementados los métodos necesarios para enlazar el VBO o IBO que se le pida en cada momento.

```
void vao::enlazarVBO(RevObjVBO vbo)
{
    switch (vbo)
    {
        case POSNORM:
        {
            //Enlazar VBO
            vboPosNorm->bind();
            break;
        }

        case TEXTURAS:
        {
            //Enlazar VBO
            vboTexturas->bind();
            break;
        }

        case TANGENTES:
        {
            //Enlazar VBO
            vboTangentes->bind();
            break;
        }


    }
}
```

Ilustración 2.166 Implementación: Clase vao. Método enlazarVBO

```
void vao::enlazarIBO(RevObjIBO ibo)
{
    switch (ibo)
    {

        case MALLATRIANGULOS:
        {
            iboMallaTriangulos->bind();
            break;
        }
    }
}
```

Ilustración 2.167 Implementación: Clase vao. Método enlazarIBO

Y ahora que ya tenemos implementado todo lo necesario para poder dibujar nuestros objetos en escena pasamos a la implementación de las luces.

Para la implementación de las luces se creó la clase “lightsource” · Esta clase contendría todos los atributos necesarios que puede necesitar una luz cualquiera (ambiental, direccional, puntual o spot) y un método al cual se llama desde el método “dibujar” de la clase “renderer” que se el método “aplicar”. Este método se encarga de pasar los uniforms necesarios al shader en función de la luz que se vaya a dibujar.

```
QVector3D Ia;
QVector3D Id;
QVector3D Is;
QVector3D position;
QVector3D direction;
QVector3D lookAt;
float anguloMenor;
float anguloMayor;
bool keyLight;
QMatrix4x4 visionMatrix;
LightTypes tipo;
lightapplicator* aplicador;
```

Ilustración 2.168 Implementación: Clase lightsource. Atributos

```
void aplicar(shaderProgram* shader);
```

Ilustración 2.169 Implementación: Clase lightsource. Método aplicar

El problema aquí es que una instancia de la clase “lightsource” contiene atributos que, en función del tipo de luz que sea, no son suyos y no han de ser pasados al shader para dibujarse. Para solventar esto se creó una clase que actúa como una interfaz, la clase “lightapplicator”. Esta clase contiene un método “aplicar” que será sobrecargado por sus clases hijas para pasarle al shader únicamente los atributos necesarios de cada luz. De esta clase heredan las clases “lightapplicatorambiental”, “lightapplicatorspot”, “lightapplicatorpuntual” y “lightapplicatordireccional” que son las clases que sobrecargarán el método aplicar.

De esta forma, una instancia de la clase “lightsource” tiene asociada una instancia hija de la interfaz “lightapplicator” y, cuando “renderer” llama a una instancia de “lightsource” para que pase sus parámetros al shader, esta instancia delega en su aplicador para que pase los parámetros necesarios.

```
class lightapplicator
{
public:

    //Método virtual que se sobrecargará en las clases hijas para aplicar un tipo de luz
    virtual void aplicar(shaderProgram* shader, lightsource* light) = 0;
    virtual LightTypes getTipo() = 0;
};
```

Ilustración 2.170 Implementación: Clase lightapplicator

```
void lightapplicatorambiental::aplicar(shaderProgram* shader, lightsource* light)
{
    shader->use();
    shader->setUniform("Ia", light->getIa());
}
```

Ilustración 2.171 Implementación: Clase lightapplicatorambiental. Método aplicar

```
void lightapplicatordireccional::aplicar(shaderProgram* shader, lightsource* light)
{
    shader->use();
    shader->setUniform("Id", light->getId());
    shader->setUniform("Is", light->getIs());
    QVector4D aux;
    aux = light->getVisionMatrix() * QVector4D(light->getDirection(),0);
    shader->setUniform("lightDirection", aux.toVector3D());
    shader->setUniform("atenuacionPosition",light->getPosition());
    shader->setUniform("M_PI",static_cast<float>(M_PI));
}
```

Ilustración 2.172 Implementación: Clase lightapplicatordireccional. Método aplicar

```
void lightapplicatorpuntual::aplicar(shaderProgram* shader, lightsource* light)
{
    shader->use();
    shader->setUniform("Id", light->getId());
    shader->setUniform("Is", light->getIs());
    QVector4D aux;
    aux = light->getVisionMatrix()*QVector4D(light->getPosition(),1);
    shader->setUniform("lightPosition", aux.toVector3D());
    shader->setUniform("M_PI",static_cast<float>(M_PI));
}
```

Ilustración 2.173 Implementación: Clase lightapplicatorpuntual. Método aplicar.

```
void lightapplicatorspot::aplicar(shaderProgram* shader, lightsource* light)
{
    shader->use();
    shader->setUniform("Id", light->getId());
    shader->setUniform("Is", light->getIs());
    QVector4D aux;
    aux = light->getVisionMatrix()*QVector4D(light->getDirection(),0);
    shader->setUniform("lightDirection",aux.toVector3D());
    QVector4D aux2;
    aux2 = light->getVisionMatrix()*QVector4D(light->getPosition(),1);
    shader->setUniform("lightPosition", aux2.toVector3D());
    float cosMenor = cos(light->getAnguloMenor() * static_cast<float>(M_PI) / 180.0f);
    shader->setUniform("cosMenor",cosMenor);
    float cosMayor = cos(light->getAnguloMayor() * static_cast<float>(M_PI) / 180.0f);
    shader->setUniform("cosMayor",cosMayor);
    shader->setUniform("M_PI",static_cast<float>(M_PI));

}
```

Ilustración 2.174 Implementación: Clase lightapplicatorspot. Método aplicar

```
void lightsource::aplicar(shaderProgram* shader)
{
    this->aplicador->aplicar(shader, this);
}
```

Ilustración 2.175 Implementación: Clase lightsource. Método aplicar

Aparte de encargarse de aplicar y gestionar los atributos de iluminación de cada luz, la clase “lightsource” también se encarga de las sombras.

Cada luz debe de generar su mapa de sombras y, para ello, hay que colocar una cámara virtual que nos dé una visión de la escena desde el punto de vista de cada luz. Como vimos en la clase “camera” para crear una cámara necesitamos una posición (posición de la luz), un punto al que mira la cámara (lookAt de la luz u objeto de interés), los planos zNear y Zfar y el fovY (a estos tres últimos se les da un valor por defecto que posteriormente podrá ser modificado por el usuario). Por lo tanto, lo tenemos todo para crear los vectores de la luz y calcular su matricez de visión y proyección. Lo único a tener en cuenta en este punto es que la luz direccional utiliza una matriz de proyección paralela en lugar de una matriz de proyección en perspectiva ya que los rayos de una luz direccional llegan paralelos a la escena.

```
void lightsource::calcularVPMatrix()
{
    //Cálculo de los vectores de la luz
    QVector3D n = position - lookAt;
    n.normalize();

    QVector3D u;
    if (distanciaEpsilon(QVector3D(0, 1, 0), n))
    {
        u = QVector3D::crossProduct(QVector3D(0, 0, 1), n);
        u.normalize();
    }
    else if (distanciaEpsilon(QVector3D(0, -1, 0), n))
    {
        u = QVector3D::crossProduct(QVector3D(0, 0, -1), n);
        u.normalize();
    }
    else
    {
        u = QVector3D::crossProduct(QVector3D(0, 1, 0), n);
        u.normalize();
    }

    QVector3D v = QVector3D::crossProduct(n, u);
    v.normalize();

    this->proyectionMatrixLuz.setToIdentity();
    this->visionMatrixLuz.setToIdentity();

    this->visionMatrixLuz.lookAt(this->position, this->lookAt, v);
    if(this->tipo == DIRECCIONAL)
    {
        this->proyectionMatrixLuz.ortho(-20,20,-20,20, this->zNear, this->zFar);
    }else
    {
        this->proyectionMatrixLuz.perspective(qRadiansToDegrees(this->fovYLuz), (this->shadowMapWidth / this->shadowMapHeight), this->zNear, this->zFar);
    }

}
```

Ilustración 2.176 Implementación: Clase lightsource. Método calcularVPMatrix

Una vez tenemos las matrices debemos de crear un FBO con una textura dónde se almacenen los valores de profundidad que se registren desde el punto de vista de la luz. Para ello haremos uso de las funciones de OpenGL por lo cual, nuestra clase “lightsource” heredará de QOpenGLFunctions_4_1_Core.

En primer lugar, le decimos a OpenGL que genere un id para nuestra textura y la enlazamos.

```
glGenTextures(1, &depthTex);
glBindTexture(GL_TEXTURE_2D, depthTex);
```

Ilustración 2.177 Implementación: Clase lightsource. Generar textura mapa de sombras

Posteriormente hay que especificar las dimensiones de la textura y hay que decirle que va a almacenar valores de profundidad (GL_DEPTH_COMPONENT32), no de color (GL_RGBA).

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32, shadowMapWidth,shadowMapHeight, 0, GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE, NULL);
```

Ilustración 2.178 Implementación: Clase lightsource. Tamaño textura

Asignamos los parámetros típicos de una textura como son el magnification filter (GL_LINEAR), el minification filter (GL_LINEAR), el wrap en s (GL_CLAMP_TO_BORDER) y el wrap en t (GL_CLAMP_TO_BORDER).

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
```

Ilustración 2.179 Implementación: Clase lightsource. Parámetros textura

Después asignamos un color al borde, esto es porque la textura de mi mapa de sombras no ocupa toda la escena y tengo que darle algún valor a los puntos que estén fuera de la escena, este valor es el valor de border que es el mayor valor de profundidad posible, así me aseguro que lo que esté fuera de mi mapa de sombras va a quedar iluminado.

```
GLfloat border[] = {1.0, 1.0, 1.0, 1.0};
```

Ilustración 2.180 Implementación: Clase lightsource. Definición del border color

```
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, border);
```

Ilustración 2.181 Implementación: Clase lightsource. Asignación del border color

Por último, hay que definir el modo de comparación, en nuestro caso GL_COMPARE_REF_TO_TEXTURE, para que los valores nuevos se comparen con los valores actuales de la textura y la función que se va a utilizar, GL_LESS, para que solo se registren valores de profundidad menores que los actuales.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,GL_COMPARE_REF_TO_TEXTURE);
//Aquí hay que especificar cómo se registran nuevos valores de profundidad. GL_LESS indica que se registran valores de profundidad menores que los actuales.
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LESS);
```

Ilustración 2.182 Implementación: Clase lightsource. Función de comparación

Con esto ya tenemos nuestra textura definida, ahora hay que crear el FBO que se encargue de albergar dicha textura en su buffer de profundidad.

En primer lugar, le decimos a OpenGL que nos cree un FBO y lo enlazamos

```
glGenFramebuffers(1, &shadowFBO2);
glBindFramebuffer(GL_FRAMEBUFFER, shadowFBO2);
```

Ilustración 2.183 Implementación: Clase lightsource. Creación del FBO

Por último, le asignamos en el GL_DEPTHH_ATTACHMENT la textura que acabamos de crear.

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,GL_TEXTURE_2D, depthTex, 0);
```

Ilustración 2.184 Implementación: Clase lightsource. Configuración del FBO

Debemos de comprobar que el FBO se ha creado correctamente y no podemos olvidarnos de volver a enlazar el buffer principal de la escena.

```
GLenum result = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if (result != GL_FRAMEBUFFER_COMPLETE)
{
    std::cout << "Framebuffer for shadows is not complete." << std::endl;
}
```

Ilustración 2.185 Implementación: Clase lightsource. Comprobación FBO

```
glBindFramebuffer(GL_FRAMEBUFFER, glwindow::getInstance()->defaultFramebufferObject());
```

Ilustración 2.186 Implementación: Clase lightsource. Enlazar framebuffer principal

Ahora desde “renderer” sólo hay que activar este FBO. y usar un shader especial que pasaré a describir posteriormente, para conseguir el mapa de sombras almacenado en la textura que hemos introducido en el DEPTH_ATTACHMENT de este FBO.

Llegamos a una de las partes más importantes de este proyecto que son los shaders. Para poder usar los shaders haremos uso de una clase denominada “shaderprogram”. Esta clase se encargará de cargar los vertex shaders y fragment shaders, compilarlos, enlazarlos a un shader program y enlazar el shader program. Además, se encargará de pasar los uniforms a la GPU.

Recordemos que un shader program está formado por varios shader object. En nuestro caso son 2, un vertex shader y un fragment shader.

El vertex shader se ejecuta una vez por vértice y recibe la información de un vértice de la geometría cada vez.

El fragment shader se ejecuta una vez por fragmento y recibe la información que proviene interpolada del rasterizador. La salida de este shader object es el color que vemos en pantalla.

Disponemos de 4 tipos de luces (ambiental, direccional, spot y puntual), dependiendo de la funcionalidad que se quiera aplicar y del tipo de luz, tendremos distintos shaders para cada una. Vamos a proceder a explicar los shaders de cada luz clasificándolos por su funcionalidad.

- Luz ambiental:

La luz ambiental es la que menor funcionalidad posee, no produce sombras ni puede ser una rim light. Su única funcionalidad es de servir de iluminación ambiente a la escena. Por ello esta luz sólo dispone de un shader que es el utilizado para la iluminación.

En su vertex shader, la luz ambiental recibe la posición, normal y coordenada de textura de un vértice por su location correspondiente. Esta location ha de coincidir con la location del VBO que contenga dichos datos.

Además, recibe las matrices de modelado y modelado*visión*proyección a través de los uniforms.

Por último, la salida de este vertex shader consiste en la posición (en coordenadas de mundo al multiplicarse por la matriz de modelado), normal (en coordenadas de mundo al multiplicarse por la matriz de modelado) y coordenada de textura.

Además, se usa una built-in-variable que ofrece OpenGL dónde se almacena la posición del vértice multiplicada por la matriz de modelado*visión*proyección, es decir, almacena la posición del vértice proyectada en el volumen de visión de la cámara.

```
#version 400

layout (location = 0) in vec3 vPosition;
layout (location = 1) in vec3 vNormal;
layout (location = 3) in vec2 vTexCoord;

uniform mat4 mvpMatrix;
uniform mat4 mModelView;
out vec3 position;
out vec3 normal;
out vec2 texCoord;

void main() {
        normal = vec3( mModelView * vec4(vNormal, 0.0) );
        position = vec3( mModelView * vec4(vPosition, 1.0) );
        texCoord = vTexCoord;
        gl_Position = mvpMatrix * vec4(vPosition, 1.0);
}
```

Ilustración 2.187 Implementación: Shader luz ambiental. Vertex shader luz ambiental

En el fragment shader, la luz ambiental recibe las posiciones, normales y coordenadas de texturas ya interpoladas.

Además, recibe la intensidad de la luz y la textura de la cual se obtiene el color a través de los uniforms

Por último, define la salida al location 0 que corresponde con el buffer de color.

La luz ambiental sigue el modelo de reflexión ambiente que establece la fórmula de iluminación como:

$$I = I_a * K_a$$

Por lo tanto, la salida del fragment shader será aplicar dicha fórmula a cada fragmento, siendo K_a el color que se extrae de samplear la textura mediante la función “textura” y las coordenadas de textura.

```
#version 400

in vec3 position;
in vec3 normal;
in vec2 texCoord;

uniform vec3 Ia;


uniform sampler2D TexSamplerColor;

layout (location = 0) out vec4 FragColor;

vec3 ads(vec4 texColor)
{
        vec3 Kad = texColor.rgb;
        return (Ia * Kad);

}

void main()
{
    vec4 texColor = texture(TexSamplerColor, texCoord);
    FragColor = vec4(ads(texColor), 1.0);
}
```

Ilustración 2.188 Implementación: Shader luz ambiental. Fragment shader luz ambiental

- Luz direccional:

La luz direccional sí presenta mayor funcionalidad ya que no sólo sirve para la iluminación, sino que puede producir sombras y puede ser utilizada como rim light y fill light.

Para ser usada en la iluminación de la escena sin producción de sombras, el vertex shader es idéntico al de la luz ambiental, pero, el fragment shader si sufre bastantes cambios.

Las entradas y salidas del fragment shader son las mismas que en la luz ambiental, pero, los uniform sí cambian.

El modelo de iluminación direccional define que los rayos llegan todos paralelos a la escena y de ahí se asume que la dirección de la luz va siempre desde el origen hasta la escena.

Además, la luz direccional (y el resto de luces a excepción de la luz ambiental) utiliza tanto el modelo de reflexión ambiental, como el modelo de reflexión difuso y especular por lo cual el color del fragmento se rige por la siguiente fórmula:

$$I = I_a * K_a + I_d * K_d * (\hat{l} * \hat{n}) + I_s * K_s * (\hat{r} * \hat{v})^{ns}$$

También, la luz direccional (y el resto de luces a excepción de la luz ambiental) puede sufrir de atenuación debida a la distancia y de atenuación debida a la profundidad, esto hace que el cálculo de la iluminación final varíe en función de estos parámetros.

$$I = fog * K_{fondo} + (1 - fog) * (I_a * K_a + fatt * (I_d * K_d * (\hat{l} * \hat{n}) + I_s * K_s * (\hat{r} * \hat{v})^{ns}))$$

Vamos a comenzar primero con la explicación del cálculo de la iluminación direccional sin atenuación, es decir, la iluminación que se rige por la fórmula:

$$I = I_a * K_a + I_d * K_d * (\hat{l} * \hat{n}) + I_s * K_s * (\hat{r} * \hat{v})^{ns}$$

Los valores de intensidad son pasados a través de los uniforms, el color (K_ay K_d) y brillo (K_s) se extraen a través de las texturas correspondientes que también se pasan por los uniforms y el valor de n_s también es extraído de la textura de brillo a través de su opacidad.

Los vectores que necesitamos calcular son los relacionados con el modelo de reflexión especular ($\hat{r}$ y $\hat{v}$) y con el modelo de reflexión difuso ($\hat{l}$ y $\hat{n}$).

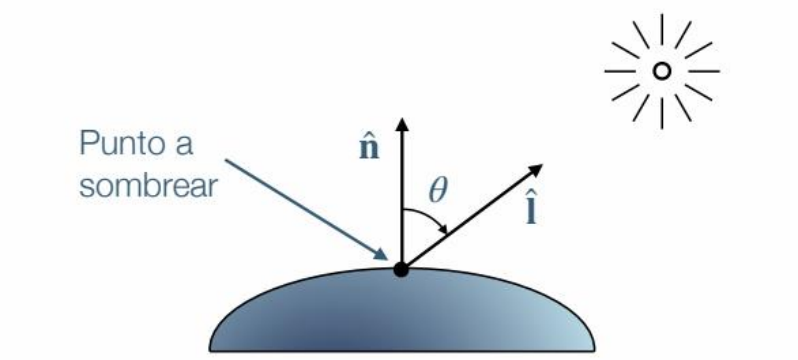


Ilustración 2.189 Implementación: Modelo de reflexión difuso

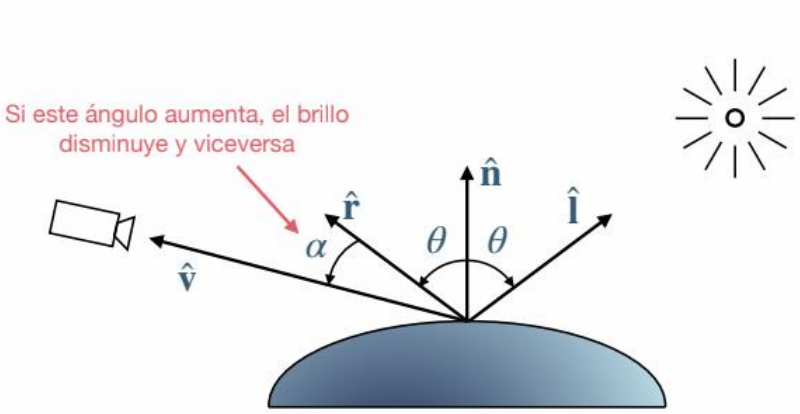


Ilustración 2.190 Implementación: Modelo de reflexión especular

Para calcular $\hat{l}$ necesitamos la dirección de la luz invertida, como es una luz direccional cuya dirección es siempre la misma, esta dirección se pasa directamente por un uniform, por lo cual, ya la tenemos.

$\hat{n}$ es la normal que viene interpolada del rasterizador, el único cambio que tendremos que hacerle es comprobar si esta normal pertenece a una cara que no está mirando hacia el observado, en cuyo caso le daremos la vuelta para que se pueda iluminar dicha cara. Además, hay que normalizarla ya que al venir del rasterizador interpolada, la normal puede no ser unitaria.

$\hat{r}$ es el vector de reflexión y se calcula mediante la reflexión de $\hat{l}$ con respecto a $\hat{n}$. Para ello hacemos uso de una función que nos proporciones OpenGL que es "reflect".

Por último, nos falta $\hat{v}$ que es el vector de visión de la cámara. Para calcularlo necesitamos hacer $posiónCámara - position$, como position está en el espacio de coordenadas de la cámara, la posición de la cámara es (0,0,0), por lo tanto, nuestro vector $\hat{v}$ es $-position$ normalizado.

Ahora vamos a ver como calcular los valores de atenuación y fog en caso de que sea necesario usarlos para la iluminación de la escena

La atenuación debida a la distancia se calcula con la siguiente fórmula:

$$fatt = \frac{1}{c_1 + c_2 * d + c_3 * d^2}$$

Los coeficientes c_1, c_2 y c_3, son pasados a través de los uniforms y la d consiste en la distancia entre la fuente luminosa y el punto a iluminar ($lightPosition -$

$position$). En el caso de la luz direccional, no tiene una posición fija, pero para este caso si sería necesario colocarle una posición para realizar los cálculos.

Por último, para la atenuación debida a la profundidad es necesario calcular un factor denominado “fog” que se calcula a partir de la siguiente función definida a trozos:

$$fog = \begin{cases} 0, & z < zmin \\ \dfrac{z - zmin}{zmax - zmin}, & zmin < z < zmax \\ 1, & z > zmax \end{cases}$$

El problema que presenta esta función es que tiene discontinuidades en zmin y zmax y la variación de fog entre los valores de profundidad próximos puede ser muy brusca al tratarse de una interpolación lineal, lo cual hace que se creen unas bandas blancas en la escena.

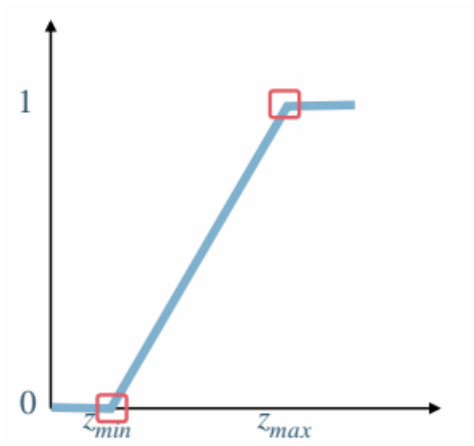


Ilustración 2.191 Implementación: Gráfico FOG

Esto tiene una fácil solución y es aplicar el coseno al resultado del fog. El problema es que nuestro fog tiene un intervalo de [0,1] y el coseno tiene un intervalo de [0,2π]. Además, de ese intervalo solo nos interesa la parte decreciente que es la que provocaría la atenuación, por lo tanto, necesitamos el intervalo [0,π]. Para ello tan sólo habría que hacer $\cos(fog * \pi)$ y ya tendríamos nuestro fog en el intervalo [0,π] para aplicarle el coseno.

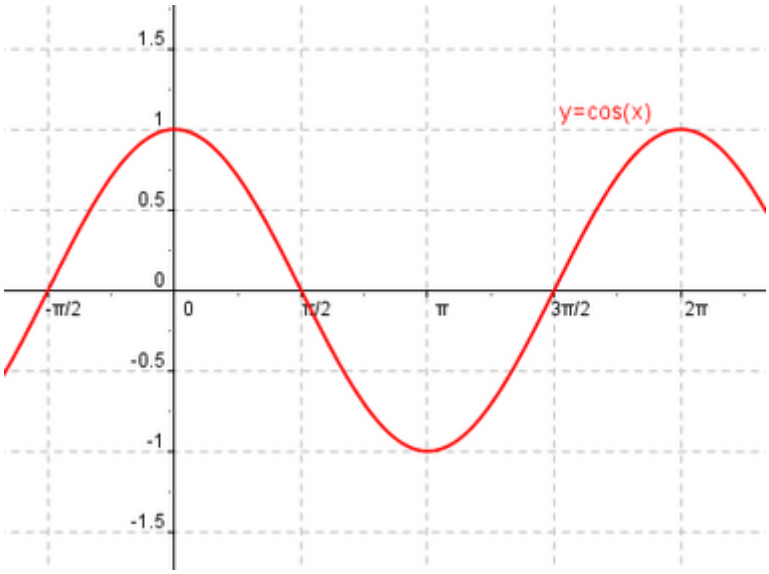


Ilustración 2.192 Implementación: Función coseno

El siguiente problema, como vemos en la gráfica del coseno, es que al aplicar este coseno obtendríamos un resultado en el intervalo [-1,1], pero nuestro factor de fog debe de estar en el intervalo [0,1]. Para conseguir ahora trasladar los puntos del intervalo [-1,1] al intervalo [0,1] tan sólo habría que dividir el resultado entre 2 y multiplicarlo por 0.5.

$$ffog = \frac{\cos(fog * \pi)}{2} * 0.5$$

Con eso obtendríamos una curva de atenuación como la de la imagen. Es una curva mucho más suave que simula mucho mejor la atenuación por la profundidad, eliminando además las bandas blancas que se producían con la interpolación lineal.

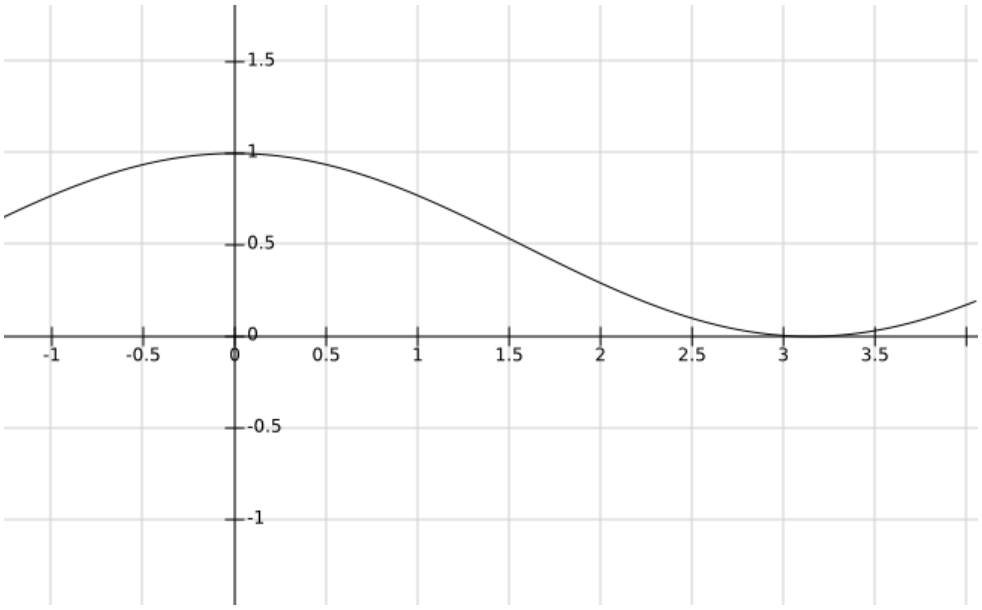


Ilustración 2.193 Implementación: Función coseno modificada

```
#version 400

in vec3 position;
in vec3 normal;
in vec2 texCoord;

uniform vec3 Id;
uniform vec3 Ia;
uniform vec3 Is;
uniform vec3 lightDirection;

uniform int atenuacionEnabled;
uniform vec3 atenuacionPosition;
uniform float c1;
uniform float c2;
uniform float c3;

uniform int fogEnabled;
uniform float zMin;
uniform float zMax;
uniform float M_PI;

uniform sampler2D TexSamplerColor;
uniform sampler2D TexSamplerBrillo;

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

Ilustración 2.194 Implementación: Shader luz direccional sin sombraas. Entradas fragment shader

```
vec3 ads(vec4 texColor,vec4 texBrillo)
{
	vec3 Kad = texColor.rgb;
	vec3 Ks = texBrillo.rgb;
	float Shininess = texBrillo.a;
	Shininess = Shininess*200;

	vec3 n;
	if (gl_FrontFacing) {
		n = normalize(normal);
	} else {
		n = normalize(-normal);
	}

	vec3 l = -lightDirection;
	vec3 v = normalize(-position);
	vec3 r = reflect(-l,n);

	float fatt = 1.0f;
	if(atenuacionEnabled == 1)
	{
	    if(c1 !=0.0f || c2!=0.0f || c3 != 0.0f)
	    {
	        vec3 vDistancia = atenuacionPosition-position;
	        float distancia = sqrt(pow(vDistancia.x,2)+ pow(vDistancia.y,2)+pow(vDistancia.z,2));
	        fatt = 1/(c1+(c2*distancia)+(c3*pow(distancia,2)));
	    }
	}

	float fog = 0.0f;
	if(fogEnabled == 1)
	{
	    if(-position.z < zMin)
	    {
	        fog = 0.0f;
	    }else if(-position.z > zMax)
	    {
	        fog = 1.0f;
	    }else
	    {
	        fog = (-position.z-zMin)/(zMax-zMin);
	    }
	    fog = cos(fog*M_PI);
	    fog /= 2.0f;
	    fog += 0.5;
	    fog = 1 - fog;
	}

	return fog*vec3(0.5f,0.5f,0.5f)+(1-fog)*((Ia * Kad) + fatt*((Id * Kad * max( dot(l, n), 0.0)) + (Is * Ks * pow( max( dot(r,v), 0.0), Shininess ))));

}
```

Ilustración 2.195 Implementación: Shader luz direccional sin sombras. Función ads fragment shader

```
void main()
{
	vec4 texColor = texture(TexSamplerColor, texCoord);
	vec4 texBrillo = texture(TexSamplerBrillo, texCoord);
	FragColor = vec4(ads(texColor,texBrillo), 1.0);
}
```

Ilustración 2.196 Implementación: Shader luz direccional sin sombras. Función main fragment shader

Para utilizar la luz direccional para la producción de sombras, es necesario utilizar 2 shaders. Primero es necesario un shader que genere el mapa de sombras, y después es necesario otro shader que dibuje la iluminación con las sombras generadas por el mapa de sombras.

En el caso del primer shader, su vertex shader es muy simple. Recibe la posición de cada vértice por el location que corresponda y recibe la matriz de modelado*visión*proyección de la luz a través de los uniforms. En este caso no se

define ninguna variable de salida, sino que se utiliza gl_Position para almacenar la posición de cada vértice proyectada en el volumen de visión de la fuente luminosa.

```
#version 400

layout (location = 0) in vec3 vPosition;

uniform mat4 mvpMatrixLuz;

void main()
{
    gl_Position = mvpMatrixLuz * vec4(vPosition, 1.0);
}
```

Ilustración 2.197 Implementación: Shader mapa de sombras. Vertex shader

En el fragment shader no hay que hacer nada ya que la información de profundidad, que es lo que necesitamos, se almacena automáticamente.

```
#version 400

void main()
{
}
```

Ilustración 2.198 Implementación: Shader mapa de sombras. Fragment shader

Ahora pasamos al caso en el que la luz direccional produzca sombras. En este caso también hay que utilizar un shader distinto.

El vertex shader es muy parecido al vertex shader para una luz direccional sin sombras, con el añadido de que ahora hay un nuevo uniform, la matriz de modelado*visión*proyección*shadowBias (mShadowMatrix) de la luz y hay una nueva variable de salida, las coordenadas de sombra. Estas coordenadas se producen aplicando la matriz mShadowMatrix a la posición del vértice.

```
#version 400

layout (location = 0) in vec3 vPosition;
layout (location = 1) in vec3 vNormal;
layout (location = 3) in vec2 vTexCoord;

uniform mat4 mvpMatrix;
uniform mat4 mModelView;
uniform mat4 mShadowMatrix;

out vec3 position;
out vec3 normal;
out vec2 texCoord;
out vec4 shadowCoord;

void main()
{
        normal = vec3( mModelView * vec4(vNormal, 0.0) );
        position = vec3( mModelView * vec4(vPosition, 1.0) );
        texCoord = vTexCoord;
        shadowCoord = mShadowMatrix * vec4(vPosition, 1.0);
        gl_Position = mvpMatrix * vec4(vPosition, 1.0);
}
```

Ilustración 2.199 Implementación: Shader luz direccional con sombras. Vertex shader

El fragment shader también es muy parecido al fragment shader de una luz direccional sin sombras. Ahora se ha añadido una nueva variable de entrada, las coordenadas de sombra interpoladas, un uniform con el mapa de sombras, un uniform con el factor mínimo de sombra (shadowwMin) y dos nuevos factores a la ecuación final que determinan el color final del fragmento si el punto está en sombra (shadowFact y shadowMul4Specular). La ecuación quedaría:

$$I = fog * K_{fondo} + (1 - fog) * (I_a * K_a + fatt * shadowFact * (I_d * K_d * (\hat{l} * \hat{n}) + shadowMul4SSpecular * (I_s * K_s * (\hat{r} * \hat{v})^{ns})))$$

El primer factor es el shadowFact, que determina si un punto está o no en sombra. Para calcular este factor se consulta el mapa de sombras en las coordenadas de sombra que se reciben, aplicándoles primero la división de perspectiva, y teniendo en cuenta también a los vecinos (todo esto lo hace la función textureProjOffset). Posteriormente se calcula la media y se utiliza la función clamp para determinar el valor shadowFactor entre shadowMin y 1.0.

La función clamp te asegura que el valor del primer parámetro estará entre el segundo parámetro (que hace de mínimo) y el tercer parámetro (que hace de máximo). Si el valor se pasa por alguno de los dos extremos, se fija su valor en ese extremo.

El segundo valor, shadowMul4Specular, se calcula en función de shadowFactor:

$$shadowMul4Specular = \begin{cases} 0, & shadowFactor < (1 - 0.00001) \\ 1, & shadowFactor \geq (1 - 0.00001) \end{cases}$$

Esto asegura que, si el punto se encuentra en sombra, no genere brillo, ya que anula la componente especular de la ecuación.

Ahora pasamos a la luz direccional en el caso en el que se comporta como una fill light. El shader es el mismo que el de una luz direccional que no produce sombras, pero pasando como término I_s un vector (0,0,0) para anular la componente especular, ya que las luces fill y rim no producen brillo. Además, la I_d de esta luz es un 40% del valor de la key light que la generó.

Por último, pasamos a la luz direccional en el caso en el que se comporta como una rim light. Aquí sí utilizamos un shader distinto pero la única diferencia con el shader de iluminación sin sombras es que el color no se extrae de ninguna textura, sino que se pasa a través de un uniform (K_{ad}) y la componente especular desaparece por completo.

```
#version 400

in vec3 position;
in vec3 normal;
in vec2 texCoord;

uniform vec3 Id;
uniform vec3 Ia;
uniform vec3 Is;
uniform vec3 Kad;
uniform vec3 lightDirection;

uniform int atenuacionEnabled;
uniform vec3 atenuacionPosition;
uniform float c1;
uniform float c2;
uniform float c3;

layout (location = 0) out vec4 FragColor;

vec3 ads()
{
        vec3 n;
        if (gl_FrontFacing) {
                n = normalize(normal);
        } else {
                n = normalize(-normal);
        }

        vec3 l = -lightDirection;
        vec3 v = normalize(-position);
        vec3 r = reflect(-l,n);

        float fatt = 1.0f;
        if(atenuacionEnabled == 1)
        {
            if(c1 !=0.0f || c2!=0.0f || c3 != 0.0f)
            {
                vec3 vDistancia = atenuacionPosition-position;
                float distancia = sqrt(pow(vDistancia.x,2)+ pow(vDistancia.y,2)+pow(vDistancia.z,2));
                fatt = 1/(c1+(c2*distancia)+(c3*pow(distancia,2)));
            }
        }

        return (Ia * Kad) + fatt*(Id * Kad * max( dot(l, n), 0.0));

}

void main()
{
        FragColor = vec4(ads(), 1.0);
}
```

Ilustración 2.200 Implementación: Shader luz direccional como rim light. Fragment shader

- Luz puntual:

Una vez que se ha explicado el funcionamiento completo de cada shader para la luz direccional, el funcionamiento en el resto de luces es muy similar, solo hay que cambiar el modo en el que se calculan ciertos parámetros y vectores.

Para usar luz puntual en la iluminación sin sombras, el shader es el mismo que en la iluminación de una luz direccional sin sombras cambiando el modo en el que se calcula el vector $\hat{l}$.

Una luz puntual, a diferencia de una luz direccional, no tiene una dirección determinada, sino que sus rayos avanzan en todas las direcciones, pero si tiene una

posición determinada, por lo cual, a la hora de calcular el vector $\hat{l}$, habría que hacer $lightPosition - position$ y normalizarlo.

```
vec3 l = normalize( lightPosition-position );
```

Ilustración 2.201 Implementación: Shader luz puntual. Cálculo vector l

Para el cálculo de la atenuación debida a la distancia, debido a que la luz puntual no tiene una dirección predefinida, se utilizaría $lightPosition - position$ Para calcular dicha dirección en cada fragmento.

En el caso de la atenuación debida a la profundidad se haría igual que en laa luz direccional

Para usar la luz puntual en la producción de sombras, el shader que se encarga de genera el mapa de sombras es el mismo que en la luz direccional.

El shader que se encarga de realizar la iluminación con sombras también es igual, solo habría que cambiar el cálculo del vector $\hat{l}$ por el descrito anteriormente.

Para utilizar una luz puntual como fill light, se usaría el mismo shader que para utilizar una luz puntual en la iluminación sin sombras, pero con un vector I_s de (0,0,0) y una I_d del 40% con respecto a la key light.

Para usar esta luz como una rim light, el shader es el mismo que para usarla en una iluminación sin sombras, pero eliminando la componente especular y usando un color K_{ad} que se pasa como uniform en lugar de extraerlo de una textura.

- Luz spot:

Al igual que pasa con la luz puntual, los shaders de la luz spot son muy parecidos a los descritos, pero cambiando la forma en la que se calculan ciertos parámetros y vectores. Además, la luz spot tiene la característica que genera un cono de iluminación.

Para la iluminación sin sombras, el shader es muy parecido al de la luz puntual. De hecho, el cálculo del vector $\hat{l}$ es el mismo. La diferencia es que en el caso de la luz spot se genera un cono de visión. Hay una zona dentro de este cono donde la iluminación es máxima, otra zona donde la iluminación se va atenuando y, fuera del cono de iluminación, no hay iluminación. Estas zonas vienen delimitadas por 2 ángulos, un ángulo menor que defina la zona de máxima iluminación y un ángulo

mayor a partir del cual la iluminación es nula. Entre los dos ángulos es dónde se produce la atenuación de la iluminación del cono.

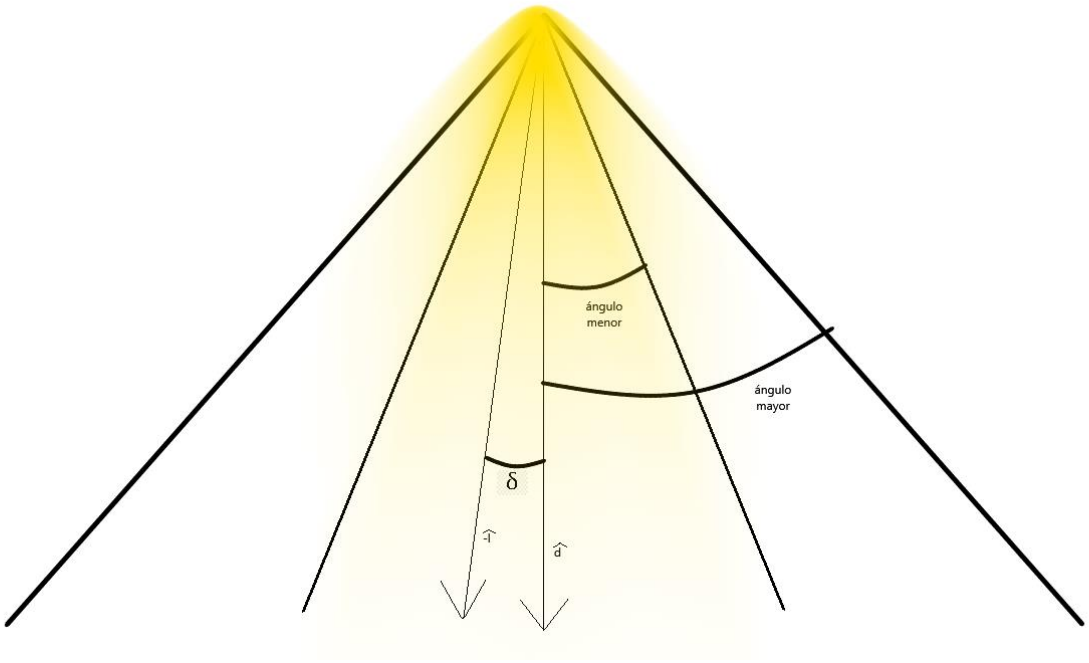


Ilustración 2.202 Implementación: Dibujo luz spot

Esto introduce por lo tanto un nuevo factor denominado factor spot que se calcula averiguando el ángulo que forman los vectores $-\hat{l}$ y $\hat{d}$ y comparando el resultado con los ángulos menor y mayor.

$$spotFactor = \begin{cases} 1, & \delta < ángulo\ menor \\ \dfrac{\delta - ángulo\ menor}{ángulo\ mayor - ángulo\ menor}, & ángulo\ menor < \delta < ángulo\ mayor \\ 0, & \delta > ángulo\ mayor \end{cases}$$

El problema que encontramos aquí es que, al ser una interpolación lineal, el cambio de atenuación es muy brusco, al igual que pasaba con el fog. El modo de solucionarlo es exactamente el mismo.

$$sf = \frac{\cos(spotfactor * \pi)}{2} + 0.5$$

Y la ecuación de la luz con el spot factor quedaría:

$$I = fog * K_{fondo} + (1 - fog) * (spotFactor * (I_a * K_a + fatt * (I_d * K_d * (\hat{l} * \hat{n}) + I_s * K_s * (\hat{r} * \hat{v})^{ns})))$$

Para usar la luz spot en la producción de sombras, el shader que se encarga de genera el mapa de sombras es el mismo que en la luz direccional.

El shader que se encarga de realizar la iluminación con sombras también es igual, solo habría que cambiar el cálculo del vector $\hat{l}$ por el descrito anteriormente y añadir el spot factor a la ecuación.

$$I = fog * K_{fondo} + (1 - fog) * (spotFactor * (I_a * K_a + fatt * shadowFact * (I_d * K_d * (\hat{l} * \hat{n}) + shadowMul4SSpecular * (I_s * K_s * (\hat{r} * \hat{v})^{ns}))))$$

Para utilizar una luz spot como fill light, se usaría el mismo shader que para utilizar una luz spot en la iluminación sin sombras, pero con un vector I_s de (0,0,0) y una I_d del 40% con respecto a la key light.

Para usar esta luz como una rim light, el shader es el mismo que para usarla en una iluminación sin sombras, pero eliminando la componente especular y usando un color K_{ad} que se pasa como uniform en lugar de extraerlo de una textura.

2.4 - Pruebas finales

2.4.1 - Pruebas de verificación del sistema

En esta sección incorporo una tabla con las pruebas realizadas y su resultado

Prueba	Resultado
Añadir una nueva luz spot con los parámetros por defecto	Correcto
Añadir una nueva luz puntual con los parámetros por defecto	Correcto
Añadir una nueva luz direccional con los parámetros por defecto	Correcto
Añadir una luz spot modificando parámetros	Correcto
Añadir una nueva luz puntual modificando parámetros	Correcto
Añadir una nueva luz direccional modificando parámetros	Correcto
Añadir dos luces del mismo tipo	Correcto
Añadir cualquier luz con la opción de key light activa y el conjunto de 3 luces activo	Correcto
Añadir cualquier luz con la opción de key light activa y el conjunto de 3 luces desactivado	Correcto
Añadir cualquier luz con la casilla de sombras marcada y las sombras activas	Correcto
Añadir cualquier luz con la casilla de sombras activa y las sombras desactivadas	Correcto
Seleccionar una luz de la lista	Correcto
Editar una luz rellenando bien los campos	Correcto
Editar una luz rellenando mal los campos	Correcto

Activar/desactivar atenuación	Correcto
Activar/desactivar fog sin ninguna key light	Correcto
Activar/desactivar el fog con alguna key light	Correcto
Activar/desactivar conjunto de 3 luces sin ninguna key light	Correcto
Activar/7desactivar el conjunto de 3 luces con alguna key light	Correcto
Eliminar una luz	Correcto
Eliminar una key light con la opción de conjunto de 3 luces activada	Correcto
Eliminar una key light con la opción de conjunto de 3 luces desactivada	Correcto
Eliminar una luz la casilla de sombras marcada y las sombras activas	Correcto
Eliminar una luz con la casilla de sombras marcadas y las sombras desactivadas	Correcto
Activar/desactivar sombras sin luces que emitan sombras	Correcto
Activar/desactivar sombras con luces que emiten sombras	Correcto
Guardar la configuración actual de la escena	Correcto
Cargar una configuración anterior de la escena	Correcto
Cambiar el objeto de la escena	Correcto
Modificar atenuación con la opción de atenuación desactivada	Correcto
Modificar atenuación con la opción de atenuación desactivada	Correcto
Modificar fog con la opción de fog desactivada	Correcto
Modificar fog con la opción de fog activada	Correcto
Modificar el mapa de sombras con la opción de sombras desactivada	Correcto
Modificar el mapa de sombras con la opción de sombras activada	Correcto
Modificar algún campo del mapa de sombras de forma incorrecta	Correcto
Mover la cámara alrededor del objeto en horizontal	Correcto
Mover la cámara alrededor del objeto en vertical	Correcto

Tabla 2.15 Pruebas de verificación del sistema

2.4.2 - Pruebas de validación del sistema

En la siguiente tabla se comprueba la validación del sistema:

Prueba	Resultado
¿Puede el usuario interactuar fácilmente con la aplicación?	Sí
¿Recibe el usuario el feedback adecuado en todo momento?	Sí
¿Se le permite realizar al usuario las tareas para las que está destinada la aplicación?	Sí
¿Puede un usuario inexperimentado hacer uso de la aplicación?	Sí
¿Es la aplicación intuitiva?	Sí
¿Es ligera la aplicación?	Sí
¿Es robusta la aplicación?	Sí

Tabla 2.16 Pruebas de validación del sistema

3 - CONCLUSIONES Y TRABAJOS FUTUROS

Como conclusión, el proyecto satisface todos los objetivos presentados y permite a los usuarios novatos poder experimentar con la iluminación de una escena, lo cual era su principal propósito.

Además, se cumbre el problema de la inexistencia de una aplicación sencilla e intuitiva con el único propósito de experimentar con la iluminación.

Como propuestas de trabajo futuro, la aplicación podría permitir a los usuarios cargar sus propios modelos .obj con sus propias texturas para realizar pruebas de iluminación. También, la aplicación podría permitir cargar y utilizar modelos sin texturas y permitir cargar más de un modelo. Incluso, podría poder crear distintas cámaras para ver la escena desde distintos puntos de vista.

Otra propuesta futura podría ser permitir al usuario mandarnos feedback sobre la aplicación. Haciéndonos saber posibles fallos, mejoras o ideas para de implementación.

4 - APÉNDICES

4.1 - Instalación y configuración del sistema

Para la instalación y ejecución de la aplicación deberá de seguir los siguientes pasos:

- En primer lugar, descomprima el zip donde viene comprimida la aplicación. Le deberá de aparecer una carpeta como la siguiente:

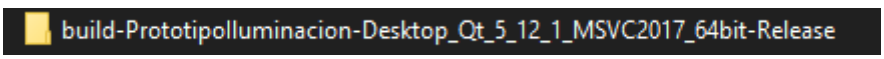


Ilustración 4.1 Instalación y configuración del sistema: Carpeta descomprimida

- Dentro de la carpeta deberá de buscar una carpeta llamada reléase y meterse dentro.

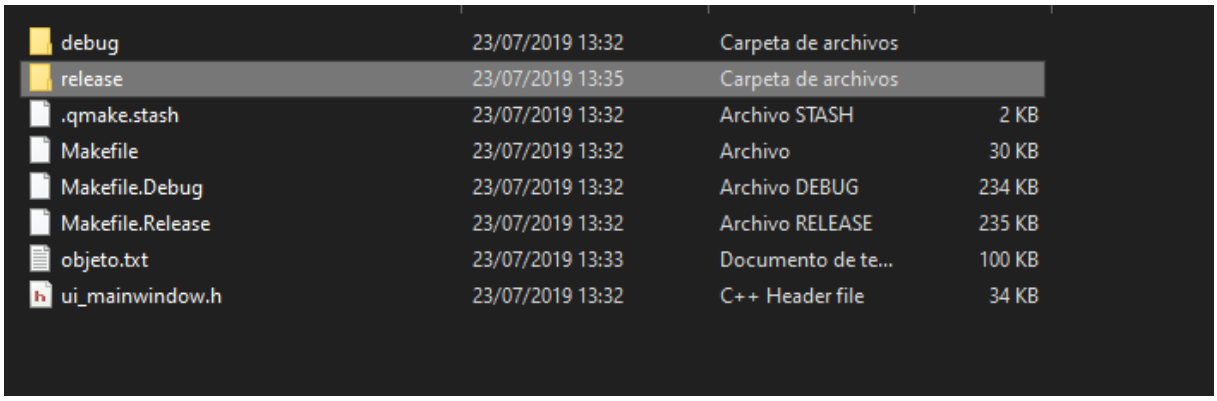


Ilustración 4.2 Instalación y configuración del sistema: Contenido carpeta

- Dentro de la carpeta “release” encontrará el ejecutable del programa.

4.2 - Manuales de usuario

Una vez se ejecute nuestra aplicación aparecerá una ventana como la siguiente:

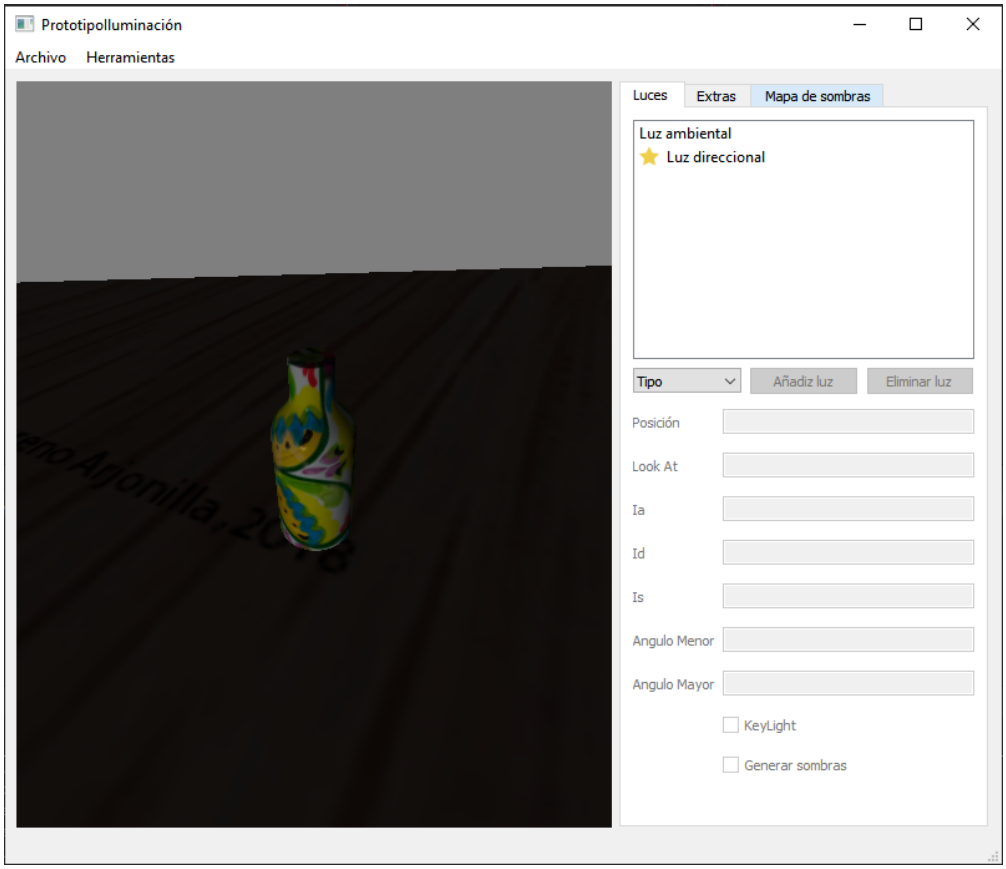


Ilustración 4.3 Manual de usuario: Ventana principal

Desde esta ventana podremos acceder a toda la funcionalidad de la aplicación.

En función de la tarea que se quiera llevar a cabo, haremos uso de unas opciones u otras. En este manual se va a explicar paso a paso como realizar las distintas tareas que ofrece la aplicación.

- Añadir una nueva luz:

Para añadir una nueva luz deberá de elegir el tipo de luz que quiere añadir de un desplegable. Después configure sus parámetros a gusto y dele click al botón de añadir. La escena se refrescará automáticamente con su nueva luz.

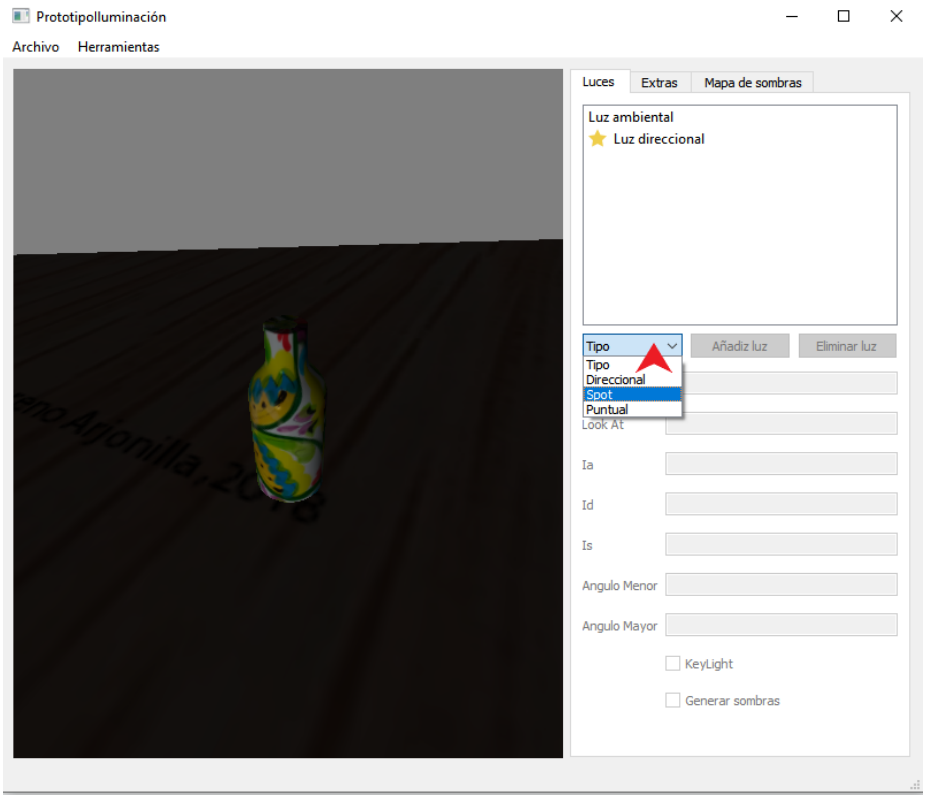


Ilustración 4.4 Manual de usuario: Añadir luz. Elección tipo de luz

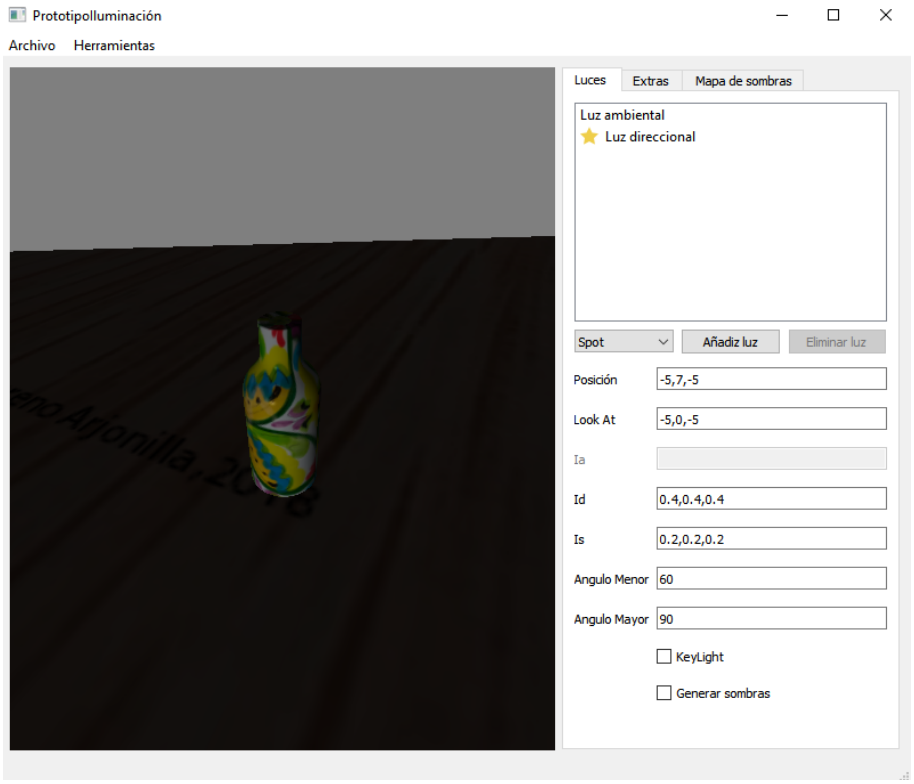


Ilustración 4.5 Manual de usuario: Añadir luz. Carga predeterminada

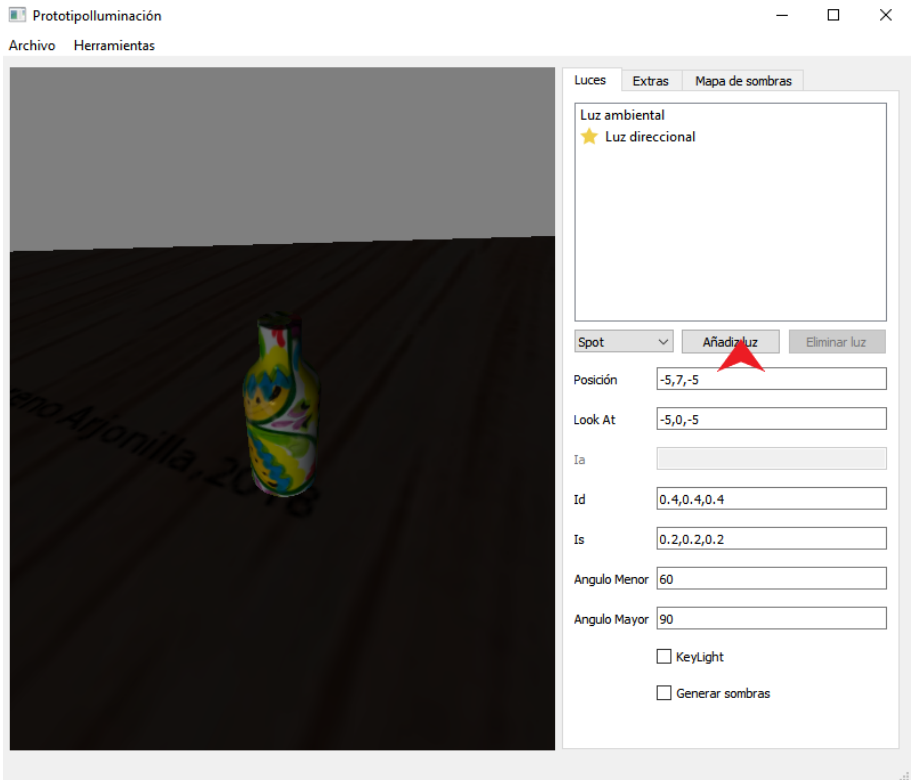


Ilustración 4.6 Manual de usuario: Añadir luz. Click "Añadir luz"

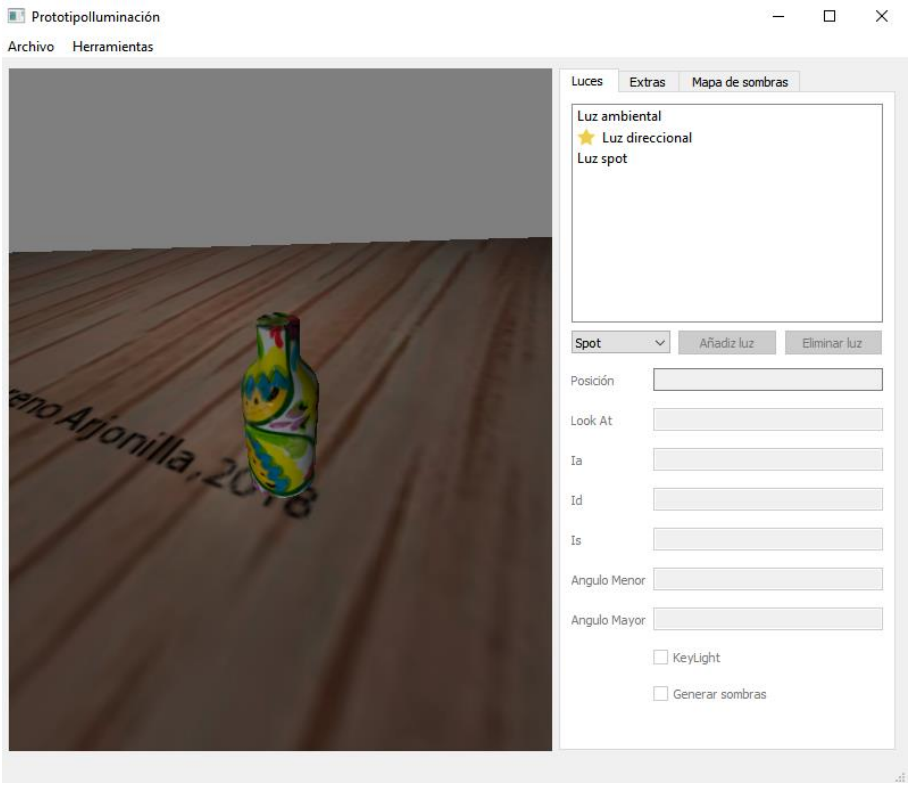


Ilustración 4.7 Manual de usuario: Añadir luz. Luz añadida

- Editar una luz existente:

Para editar una luz deberá elegir la luz en la lista de luces en escena. Después modifique sus campos a gusto y el sistema le avisará si los campos son válidos (verde) o no (rojo).

Si los campos son válidos la escena se refrescará automáticamente con sus cambios.

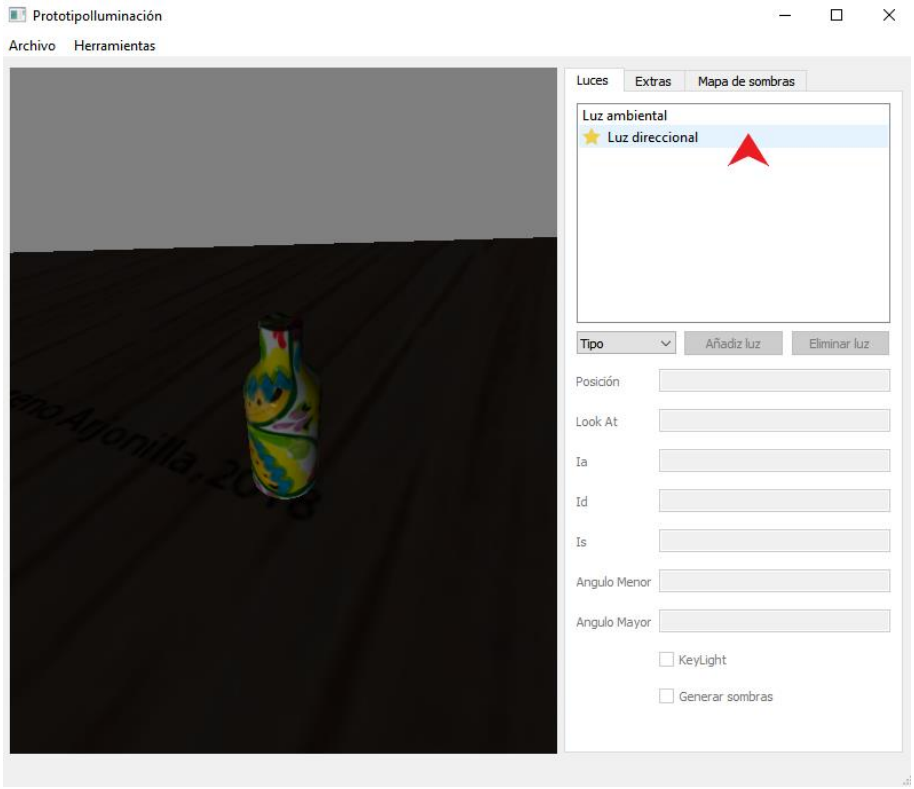


Ilustración 4.8 Manual de usuario: Editar luz. Elección luz

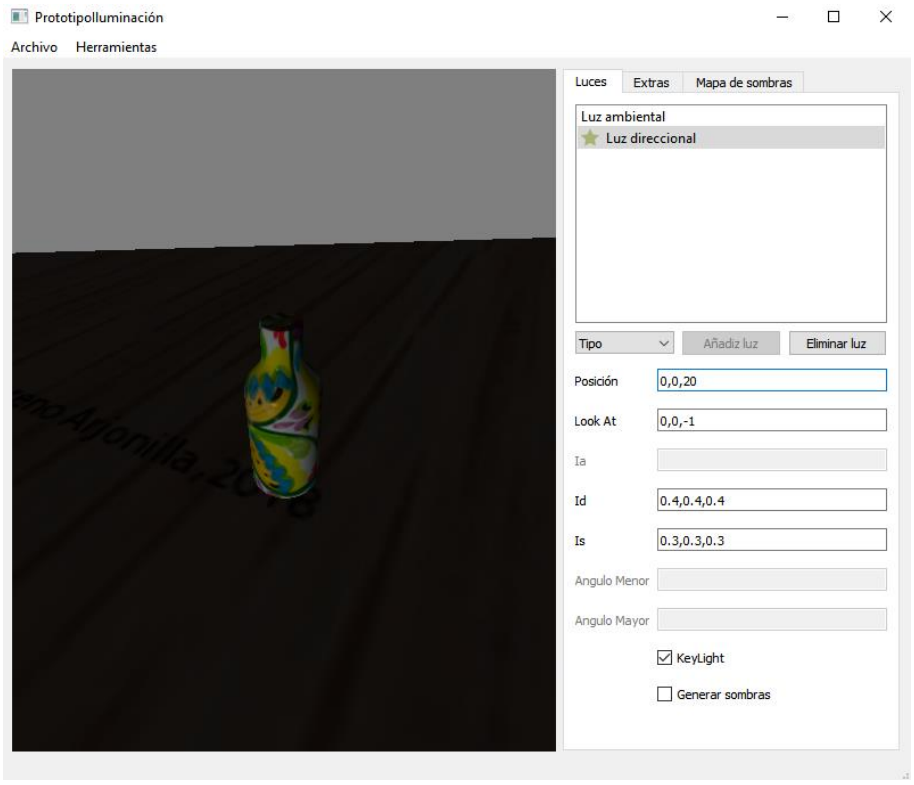


Ilustración 4.9 Manual de usuario: Editar luz. Campos cargados

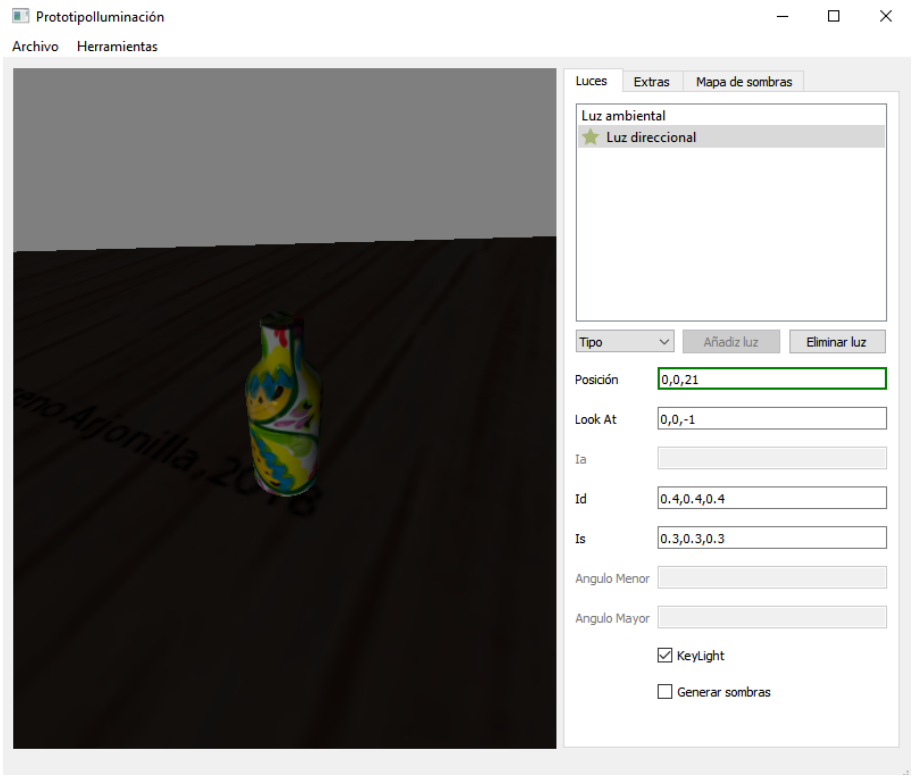


Ilustración 4.10 Manual de usuario: Editar luz. Campo editado correctamente

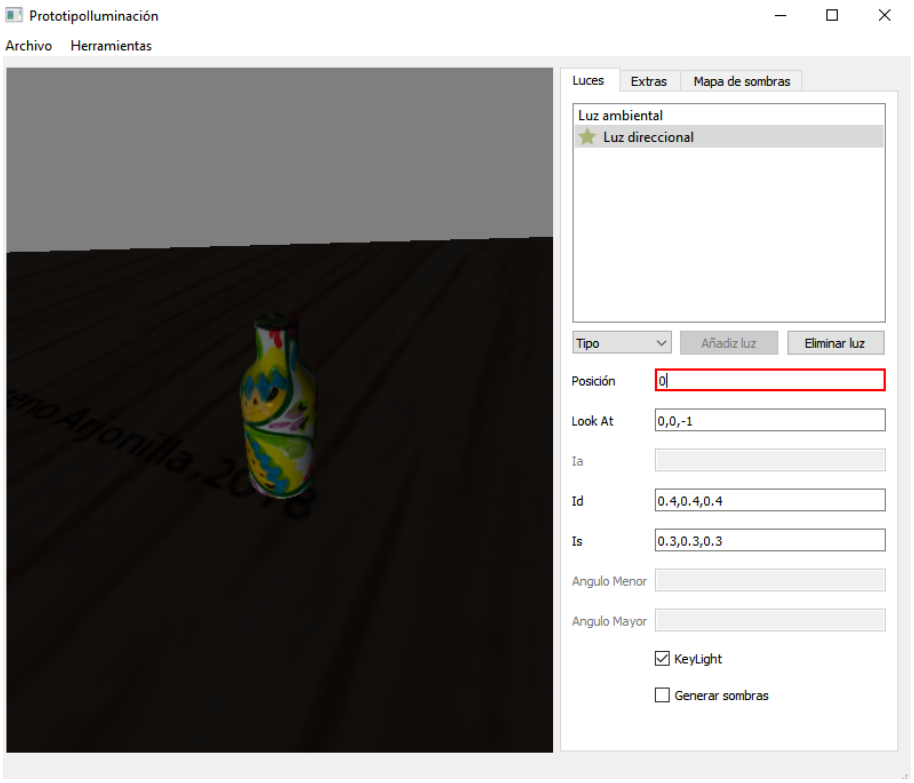


Ilustración 4.11 Manual de usuario: Editar luz. Campo editado incorrectamente

- Eliminar luz:

Para eliminar una luz deberá de seleccionar la luz que desea eliminar de la lista de luces y hacer click en el botón eliminar.

La escena se actualizará automáticamente con los cambios.

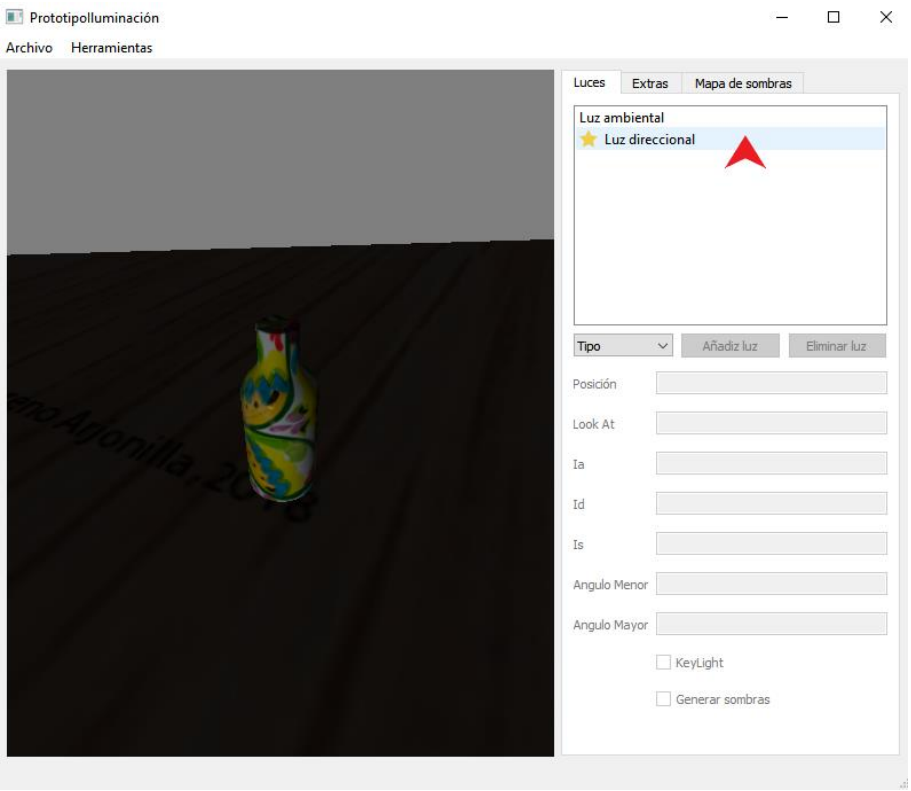


Ilustración 4.12 Manual de usuario: Eliminar luz. Elección luz

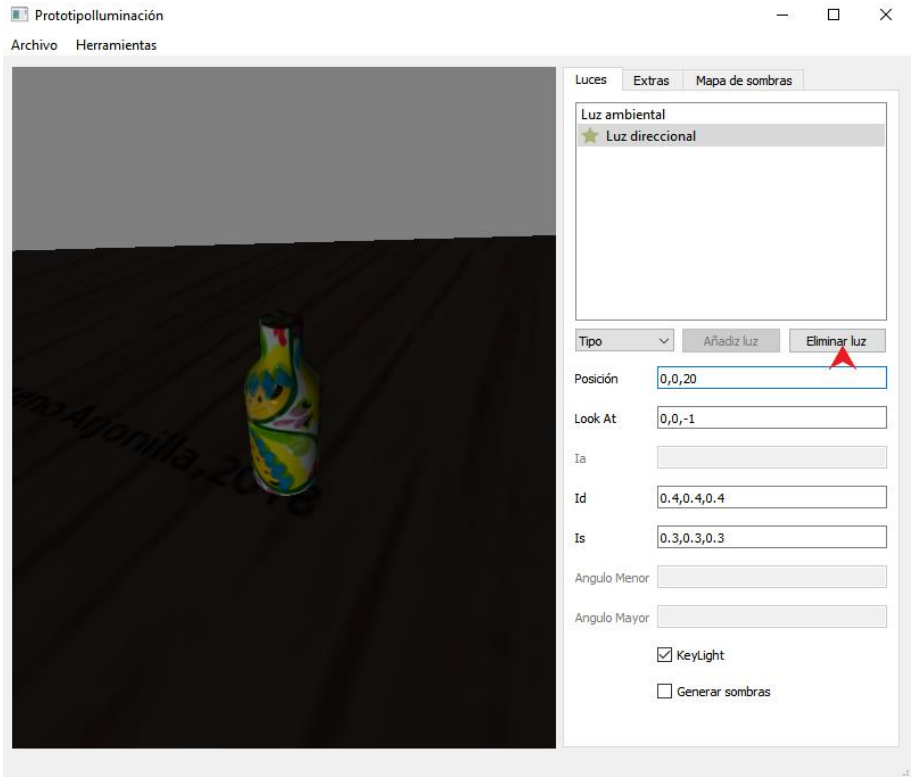


Ilustración 4.13 Manual de usuario: Eliminar luz. Click "Eliminar luz"

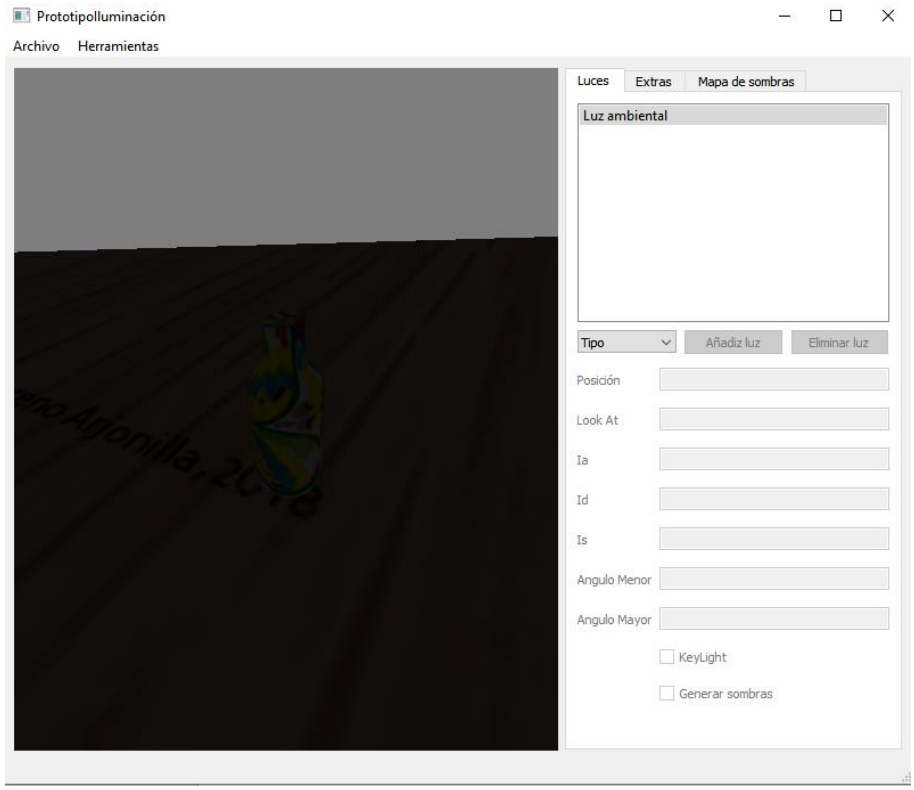


Ilustración 4.14 Manual de usuario: Eliminar luz. Luz eliminada

- Activar/desactivar atenuación:

Para activar/desactivar la atenuación deberá de ir al menú de herramientas y hacer click en la opción “Atenuación”. La opción se marcará/desmarcará y la escena se actualizará.

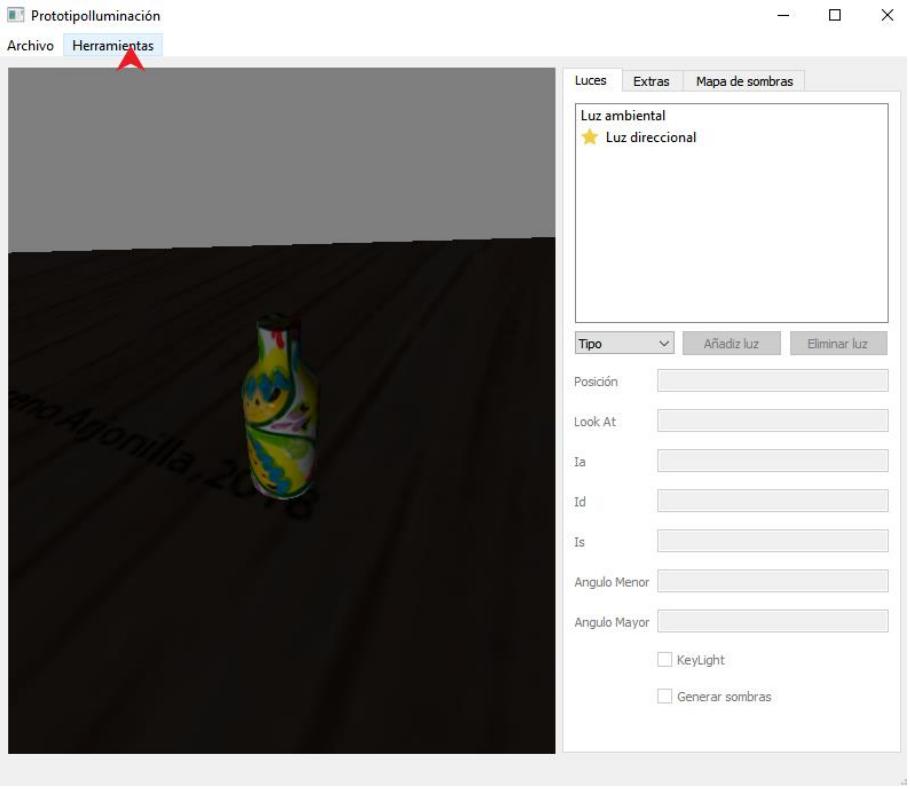


Ilustración 4.15 Manual de usuario: Activar/desactivar atenuación. Desplegar menú “Herramientas”

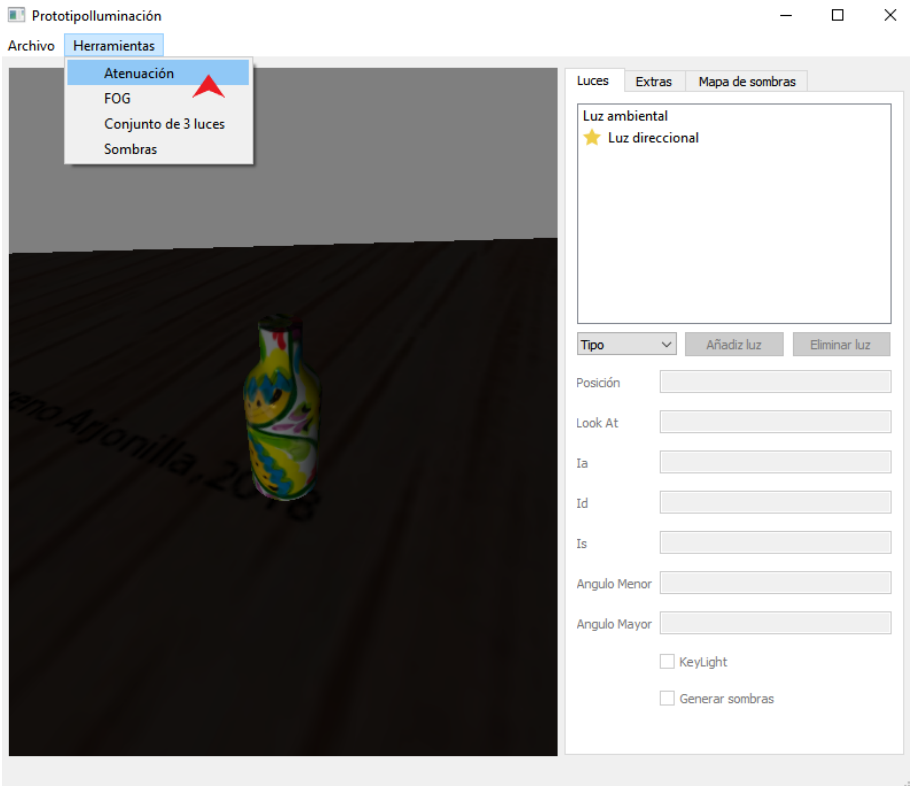


Ilustración 4.16 Manual de usuario: Activar/desactivar atenuación. Click en la opción "Atenuación"

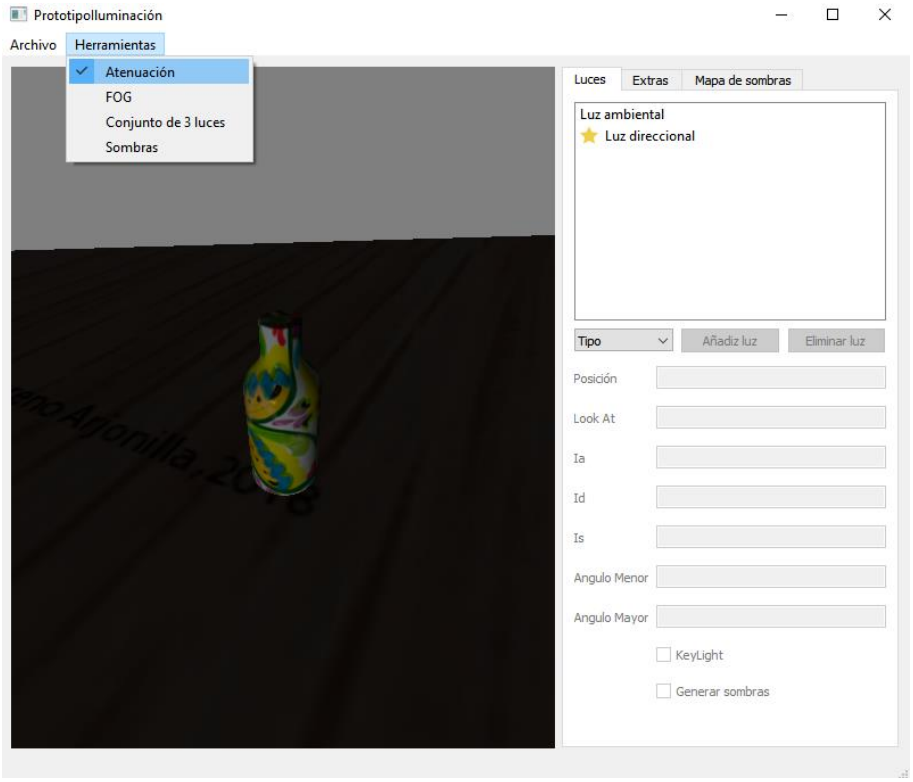


Ilustración 4.17 Manual de usuario: Activar/desactivar atenuación. Atenuación activa

- Editar atenuación:

Para poder editar la atenuación, la opción de atenuación debe de estar activa.

Deberás de ir a la pestaña “extras” y cambiar los valores que desees.

El sistema te notificará en verde si los campos son correctos y actualizará la escena.

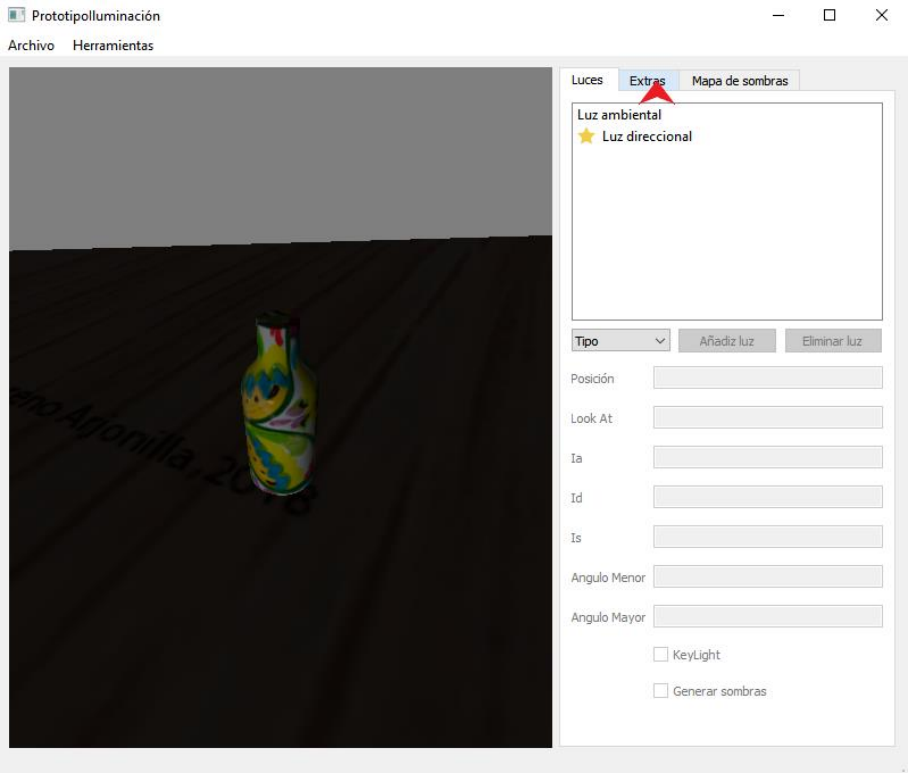


Ilustración 4.18 Manual de usuario: Editar atenuación. Click en pestaña “Extras”

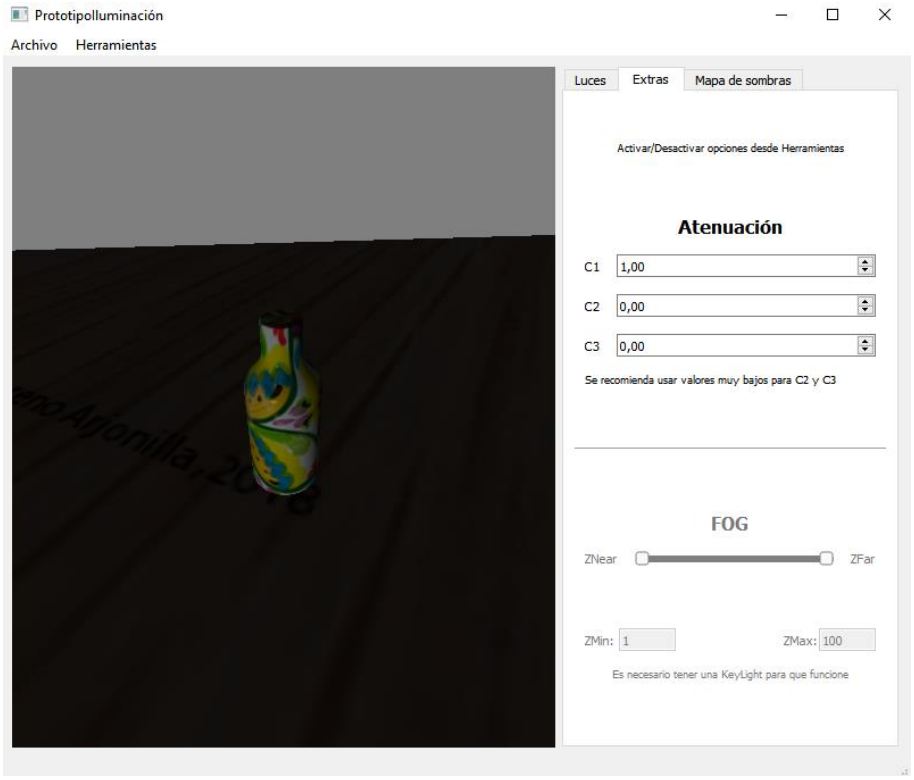


Ilustración 4.19 Manual de usuario: Editar atenuación. Modificar campos

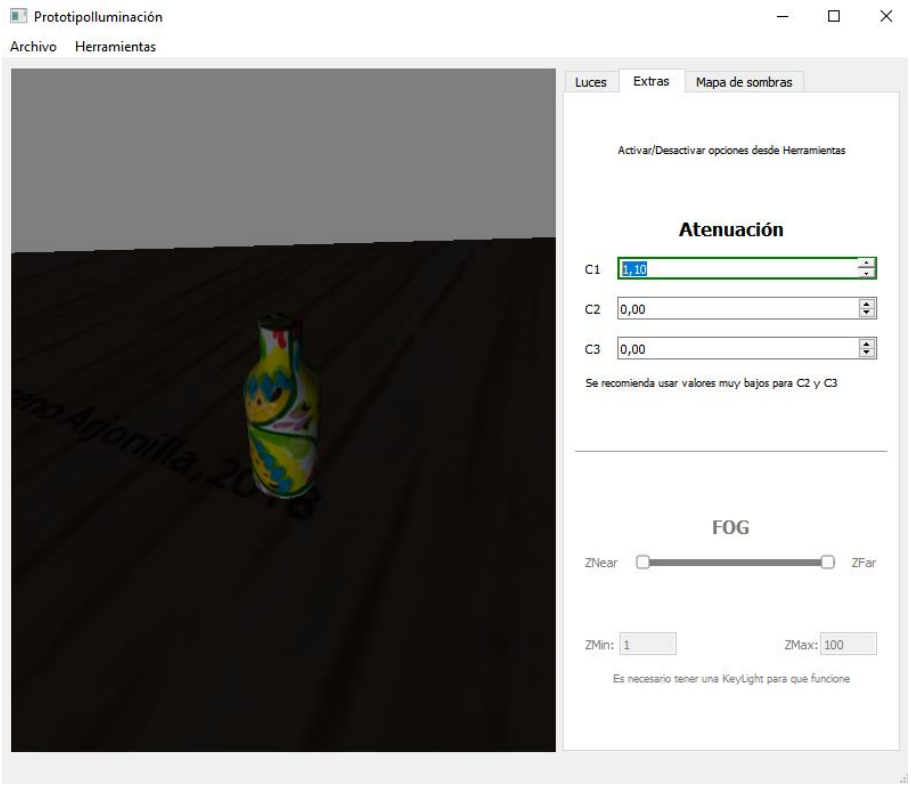


Ilustración 4.20 Manual de usuario: Editar atenuación. Campos modificados correctamente

- Activar/desactivar fog:

Para activar/desactivar la atenuación deberá de ir al menú de herramientas y hacer click en la opción “FOG”. La opción se marcará/desmarcará y la escena se actualizará.

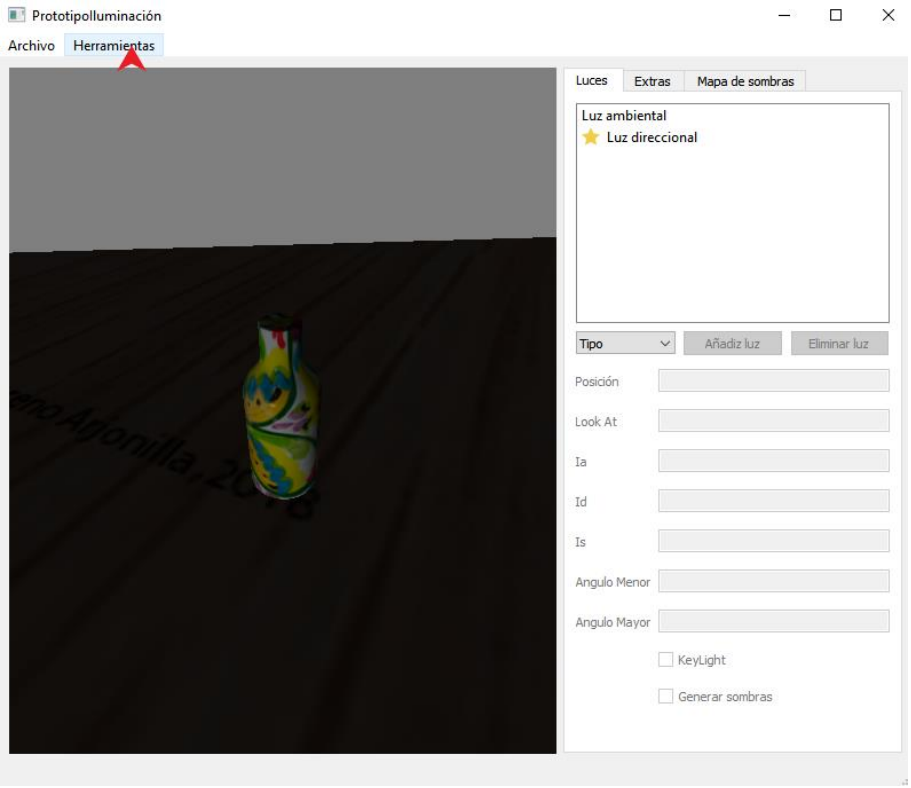


Ilustración 4.21 Manual de usuario: Activar desactivar FOG. Desplegar menú “Herramientas”

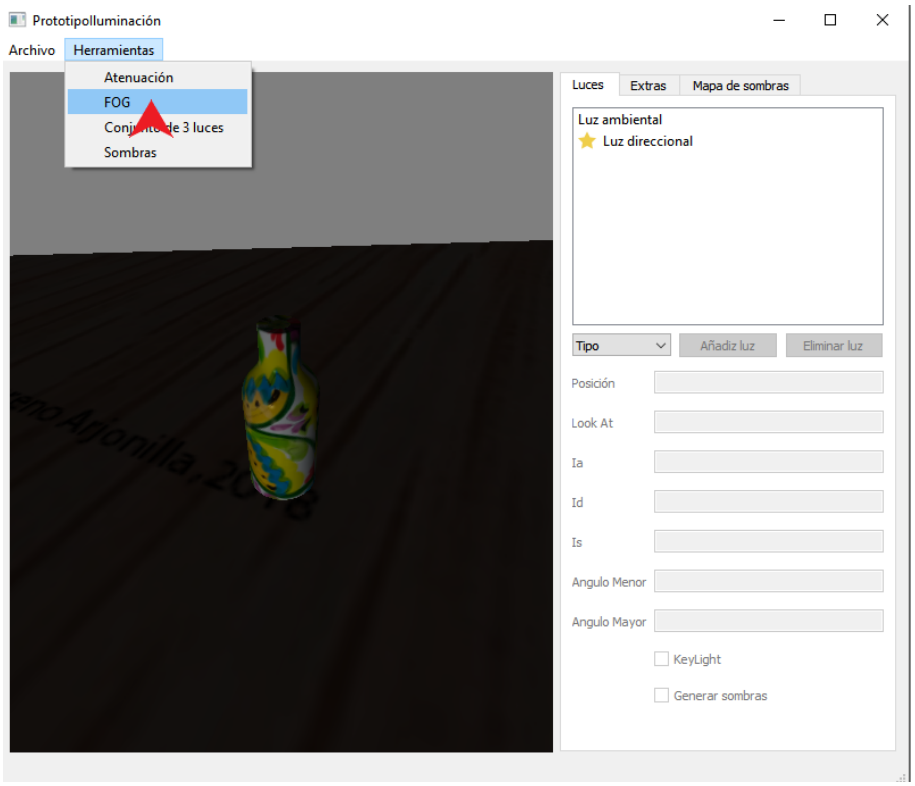


Ilustración 4.22 Manual de usuario: Activar/desactivar FOG. Click en la opción "FOG"

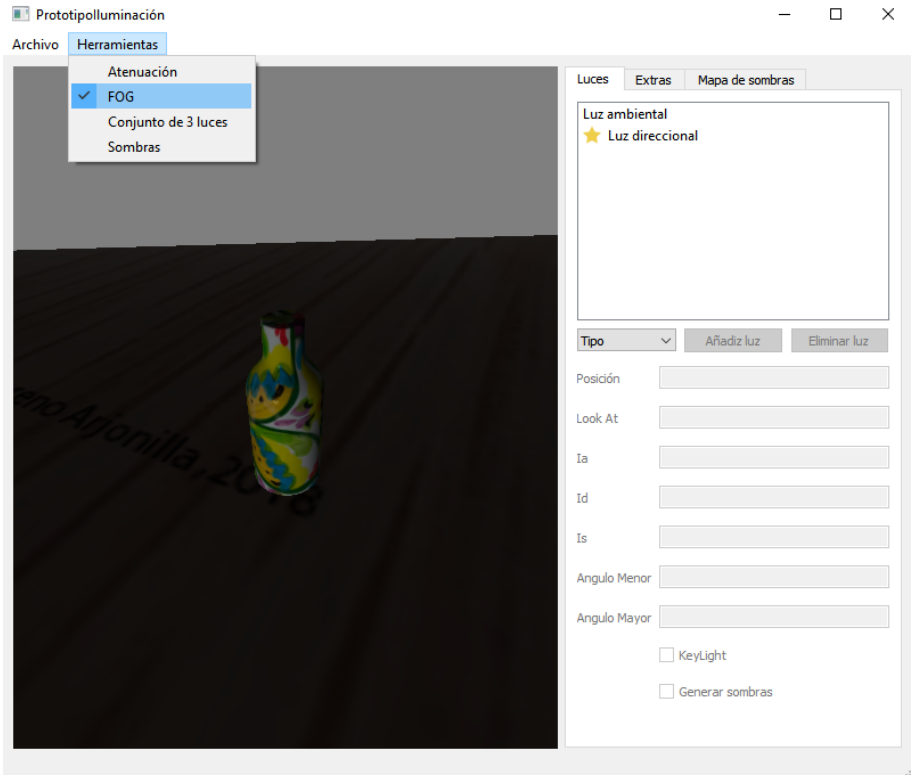


Ilustración 4.23 Manual de usuario: Activar/desactivar FOG. FOG activado

- Editar fog:

Para poder editar el fog, la opción de fog debe de estar activa.

Deberás de ir a la pestaña “extras” y cambiar los valores que desees.

El sistema te notificará en verde si los campos son correctos y actualizará la escena.

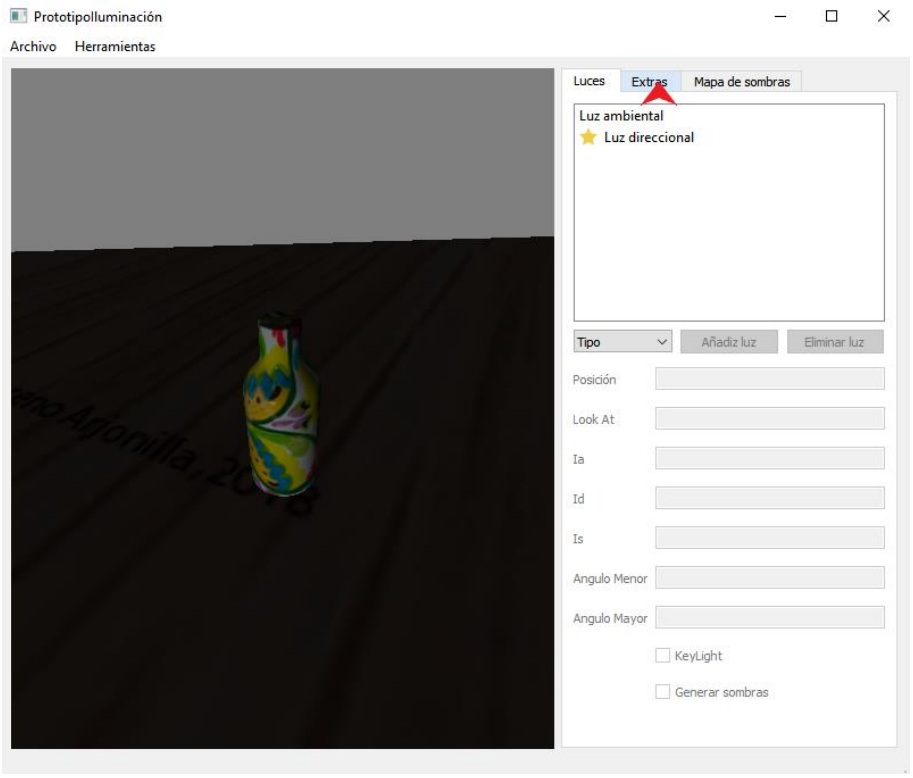


Ilustración 4.24 Manual de usuario: Editar FOG. Click en la pestaña “Extras”

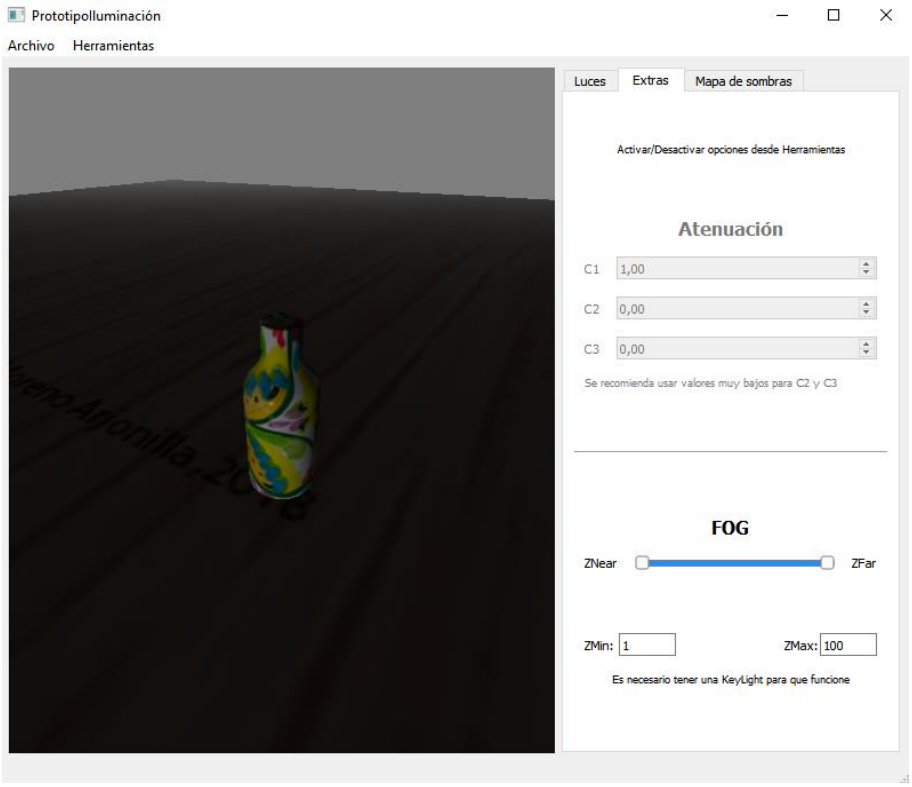


Ilustración 4.25 Manual de usuario: Editar FOG. Modificar campos

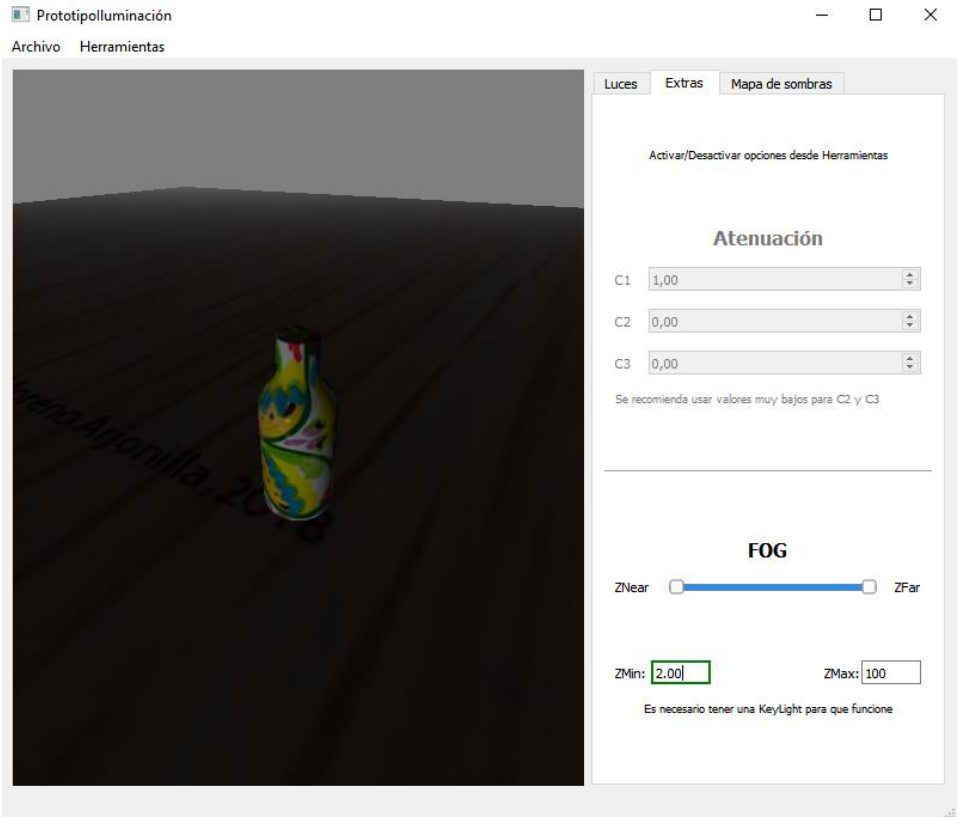


Ilustración 4.26 Manual de usuario: Activar/desactivar FOG. Campos modificados correctamente

- Activar/desactivar conjunto de 3 luces:

Para activar/desactivar la atenuación deberá de ir al menú de herramientas y hacer click en la opción “Conjunto de 3 luces”. La opción se marcará/desmarcará y la escena se actualizará. Para poder utilizar esta opción deberá de tener definida una luz como key light. Esta luz aparecerá con una estrella en la lista de luces.

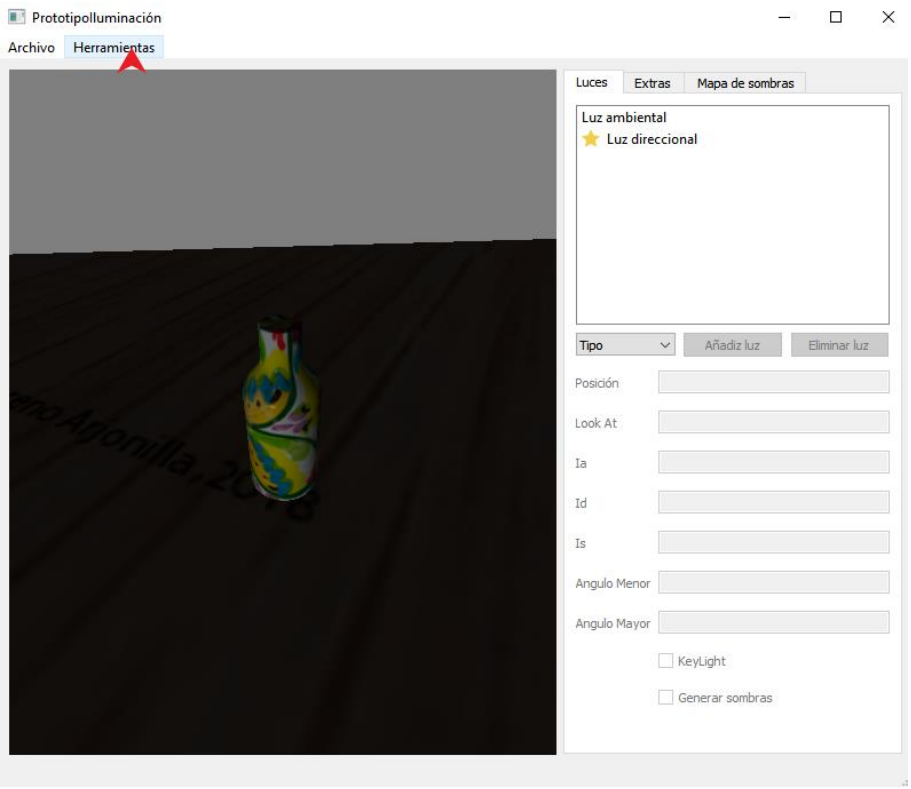


Ilustración 4.27 Manual de usuario: Activar/desactivar conjunto de 3 luces. Desplegar menú “Herramientas”

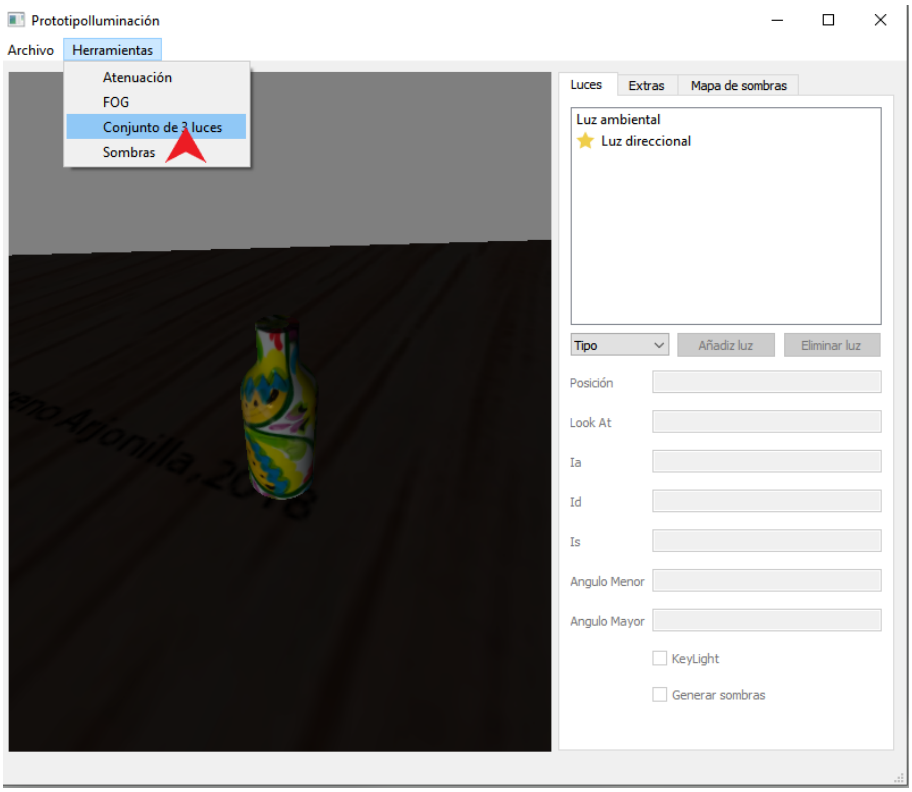


Ilustración 4.28 Manual de usuario: Activar/desactivar conjunto de 3 luces. Click en la opción "Conjunto de 3 luces"

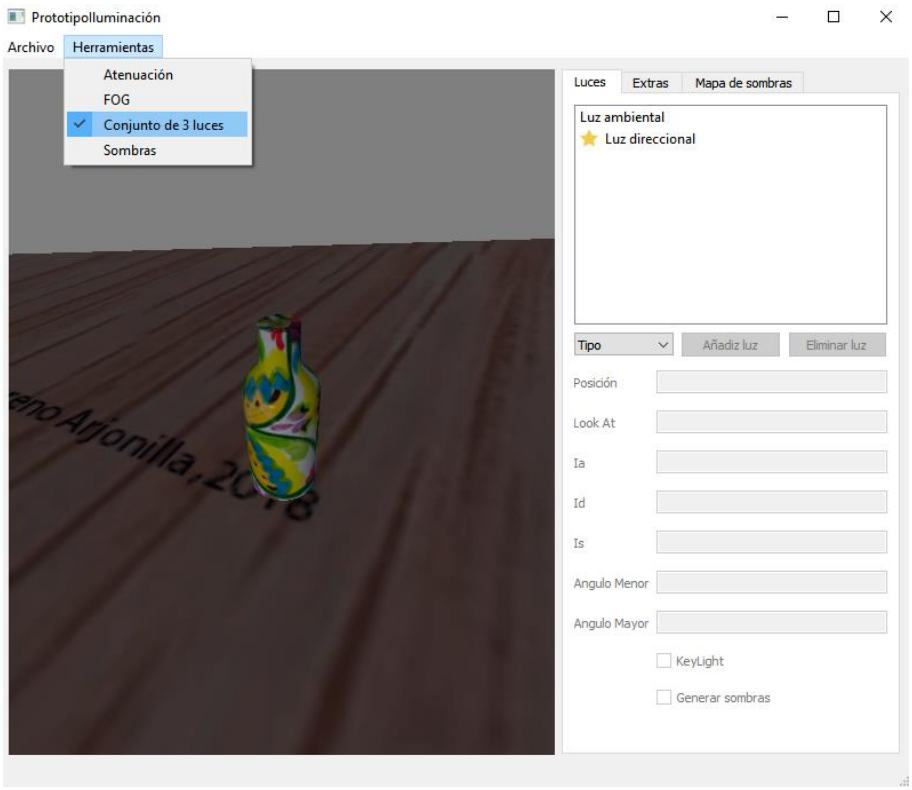


Ilustración 4.29 Manual de usuario: Activar/desactivar conjunto de 3 luces. Conjunto de 3 luces activo

- Activar/desactivar sombras:

Para activar/desactivar la atenuación deberá de ir al menú de herramientas y hacer click en la opción “Sombras”. La opción se marcará/desmarcará y la escena se actualizará. Para poder utilizar esta opción deberá de tener alguna luz con la opción de sombras activada.

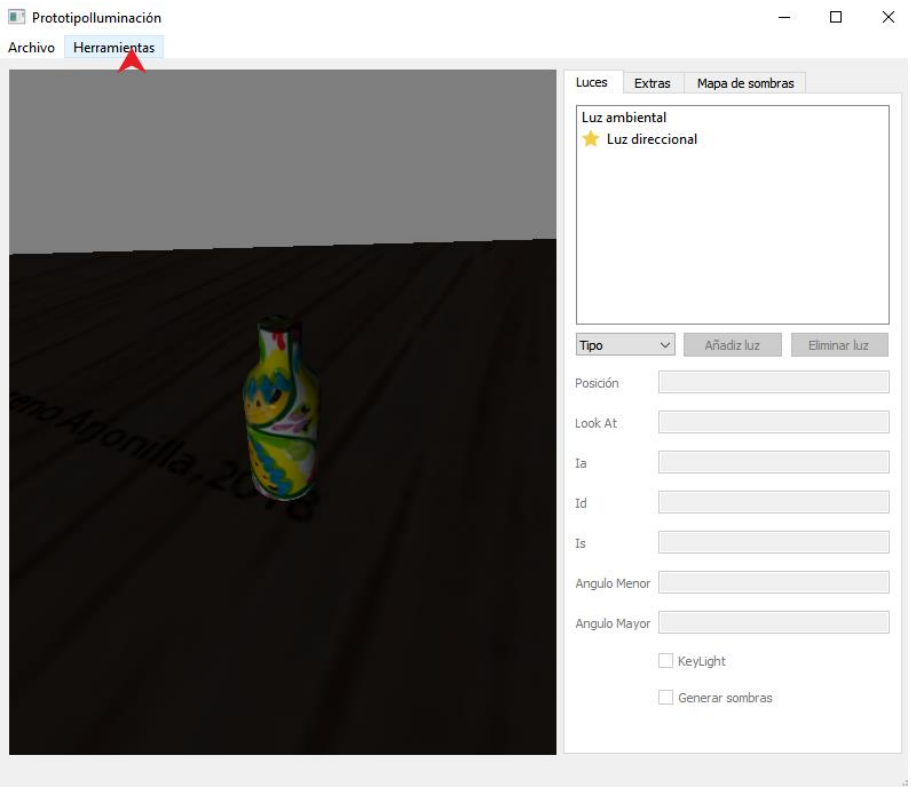


Ilustración 4.30 Manual de usuario: Activar/desactivar sombras. Desplegar menú “Herramientas”

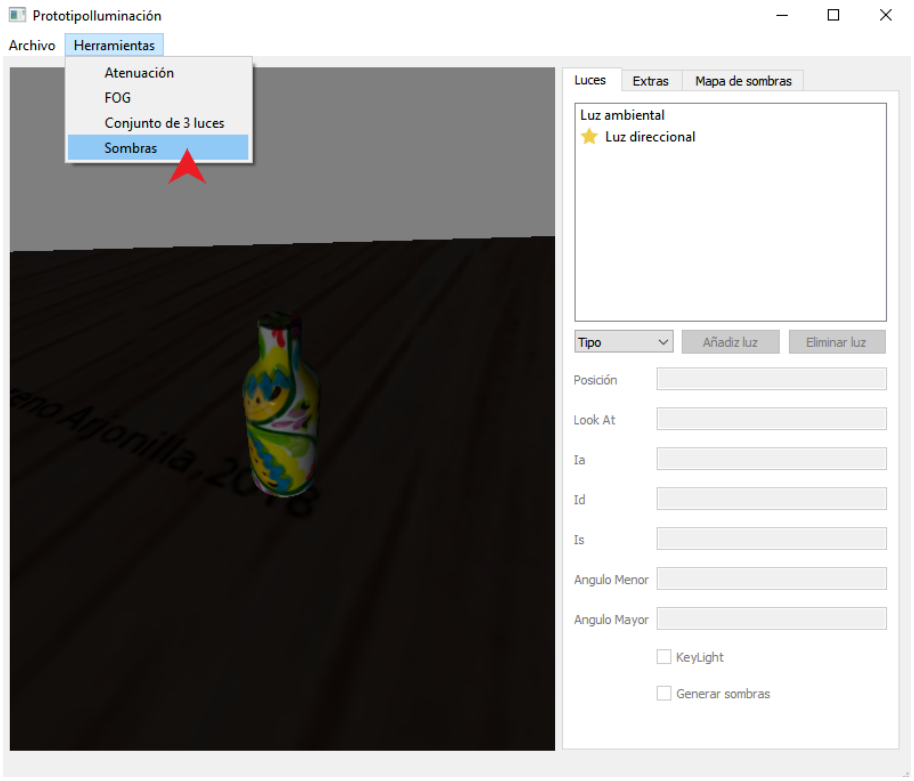


Ilustración 4.31 Manual de usuario: Activar/desactivar sombras. Click en la opción "Sombras"

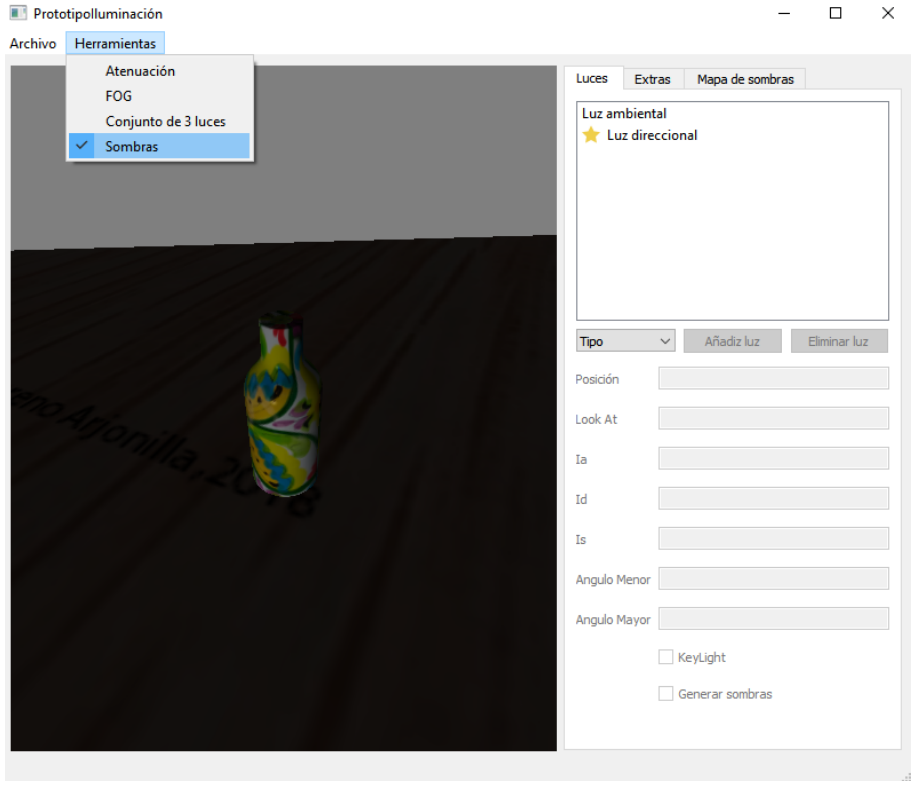


Ilustración 4.32 Manual de usuario: Activar/desactivar sombras. Sombras activas

- Editar mapa de sombras:

Para poder editar el mapa de sombras, la opción de sombras debe de estar activa. Además, deberás de tener alguna luz con la opción “sombras” marcada.

Para editar el mapa de sombras debes ir a la pestaña “Mapa de sombras”. Ahí te aparecerá una lista con las luces que producen sombras. Deberás de elegir la luz a editar y cambiar su configuración.

Si la configuración es correcta el sistema te lo notificará en verde y actualizará la escena. Sino es correcta te lo notificará en rojo y no se actualiza la escena.

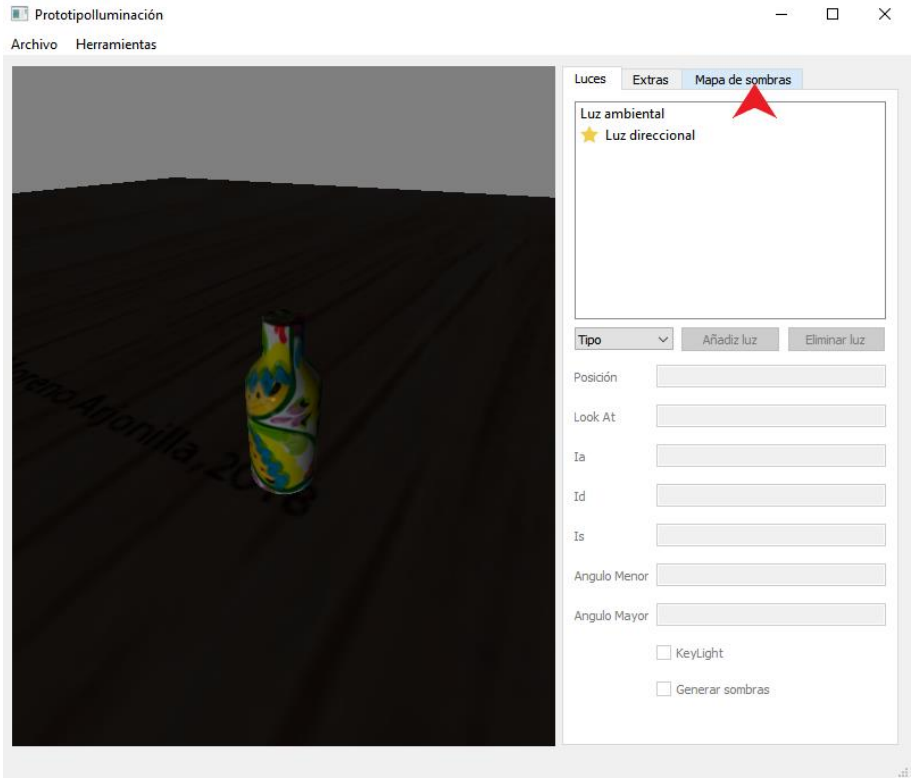


Ilustración 4.33 Manual de usuario: Editar mapa de sombras. Click en la pestaña “Mapa de sombras”

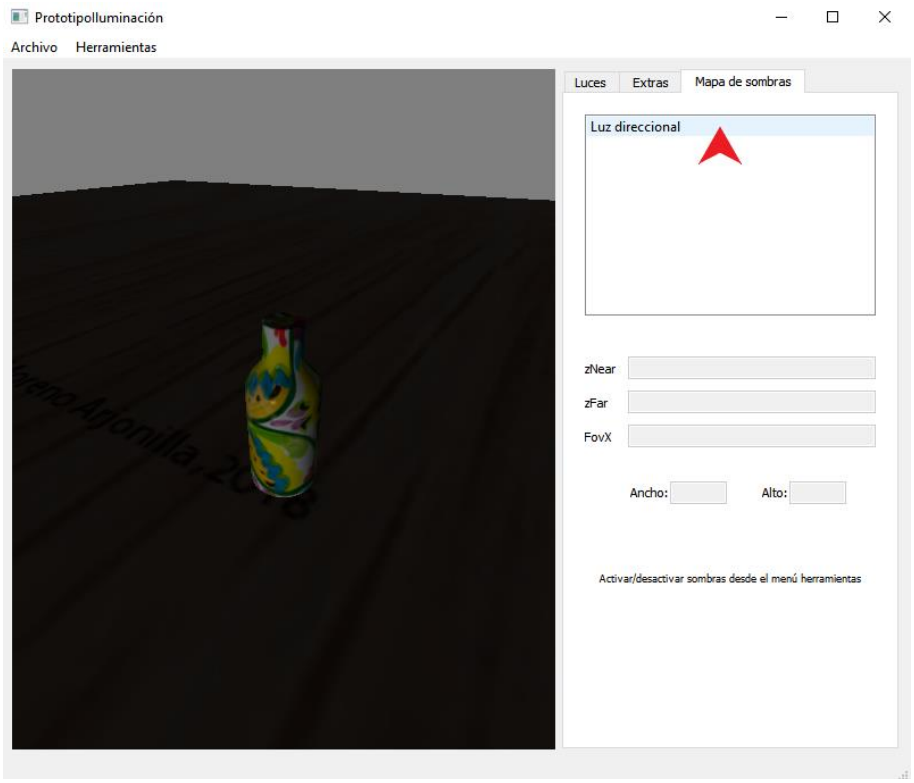


Ilustración 4.34 Manual de usuario: Editar mapa de sombras. Seleccionar luz

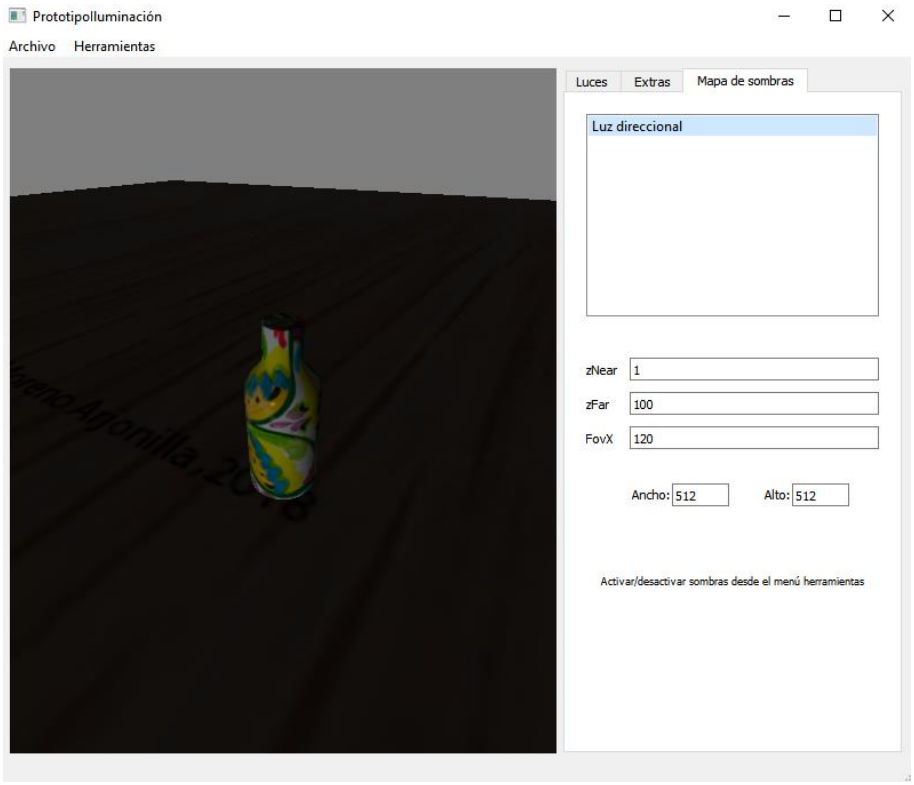


Ilustración 4.35 Manual de usuario: Editar mapa de sombras. Carga de los campos

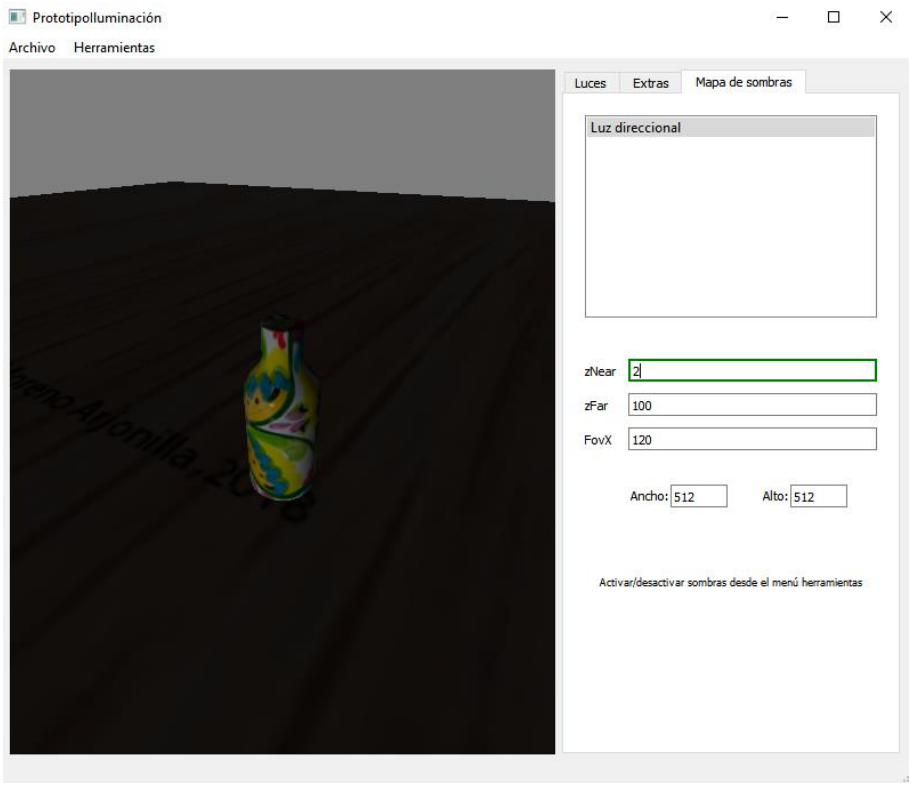


Ilustración 4.36 Manual de usuario: Editar mapa de sombras. Campo editado correctamente

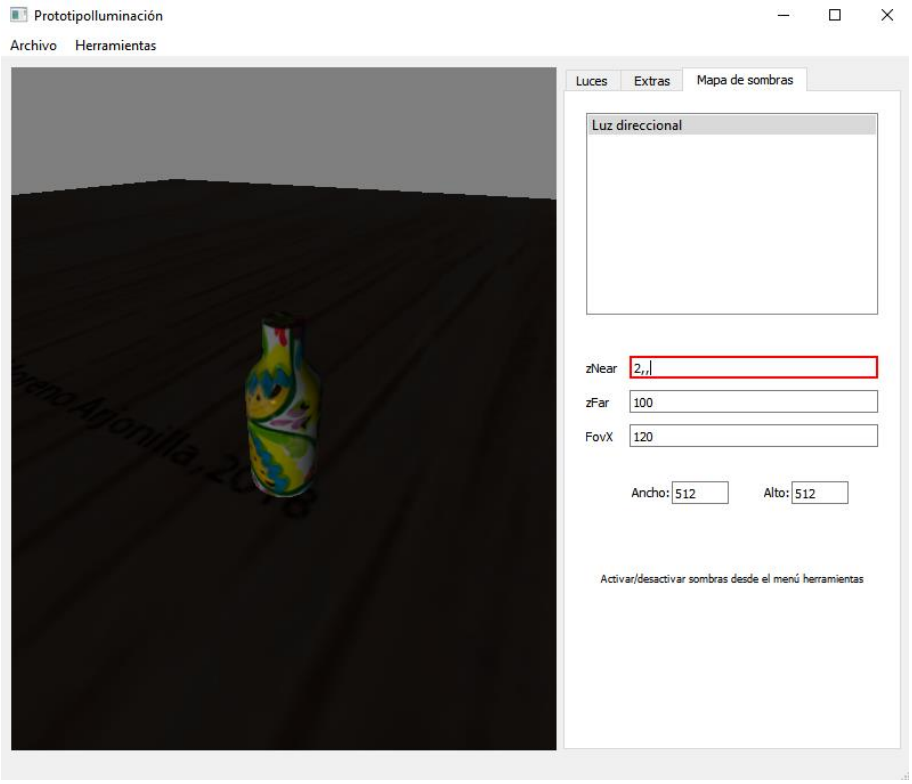


Ilustración 4.37 Manual de usuario: Editar mapa de sombras. Campo editado incorrectamente

- Mover la cámara alrededor del objeto:

Para mover la cámara alrededor del objeto deberás de mantener pulsado encima de la escena y mover el ratón en la dirección que se desee mover la cámara.

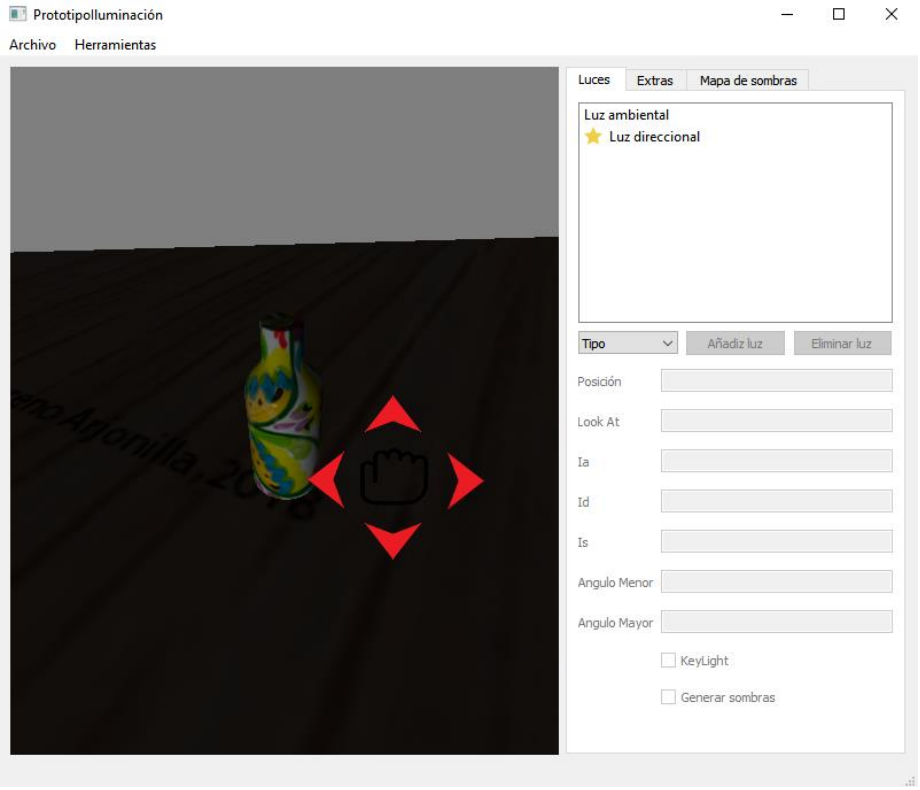


Ilustración 4.38 Manual de usuario: Mover cámara. Click y arrastre sobre la escena

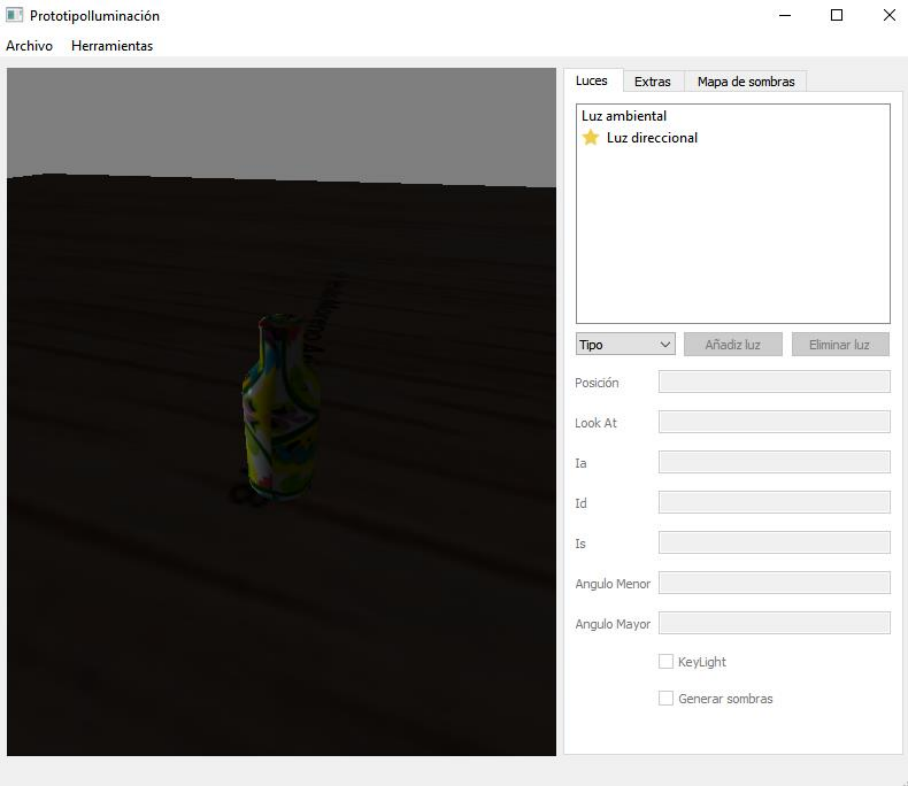


Ilustración 4.39 Manual de usuario: Mover cámara. Resultado

- Cambiar objeto en escena:

Para cambiar el objeto que se visualiza en escena deberá ir al menú “Archivo”, dentro encontrará un desplegable llamado “Objetos” y ahí deberá marcar el objeto que quiera que aparezca en escena.

Cuando seleccione un objeto, la escena se actualizará con dicho objeto.

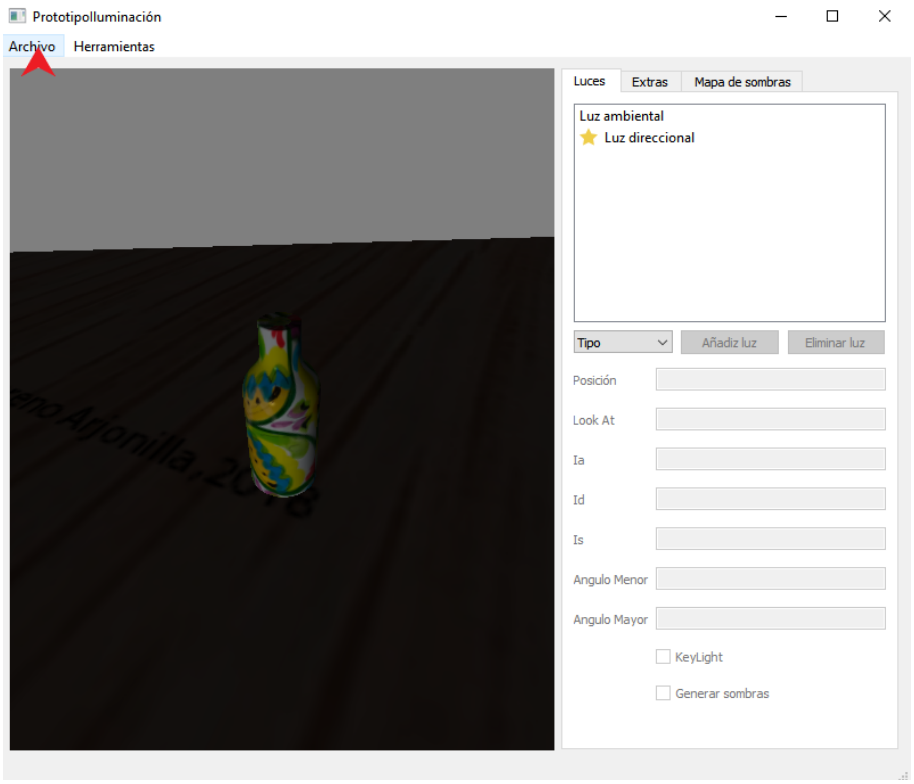


Ilustración 4.40 Manual de usuario: Cambiar objeto. Desplegar menú "Archivo"

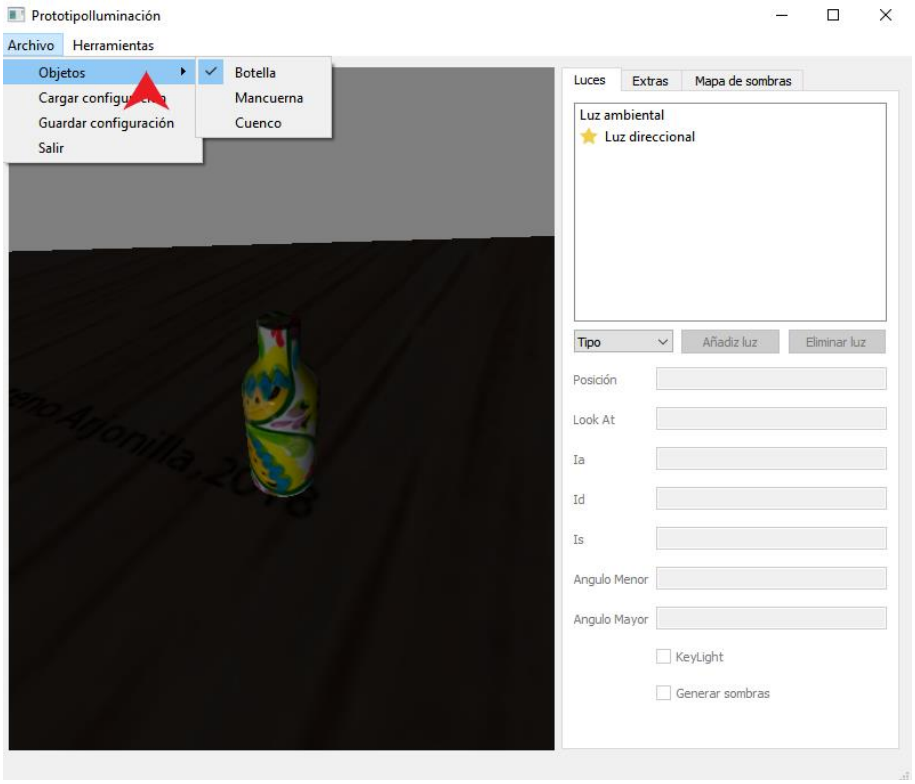


Ilustración 4.41 Manual de usuario: Cambiar objeto. Desplegar opción "Objetos"

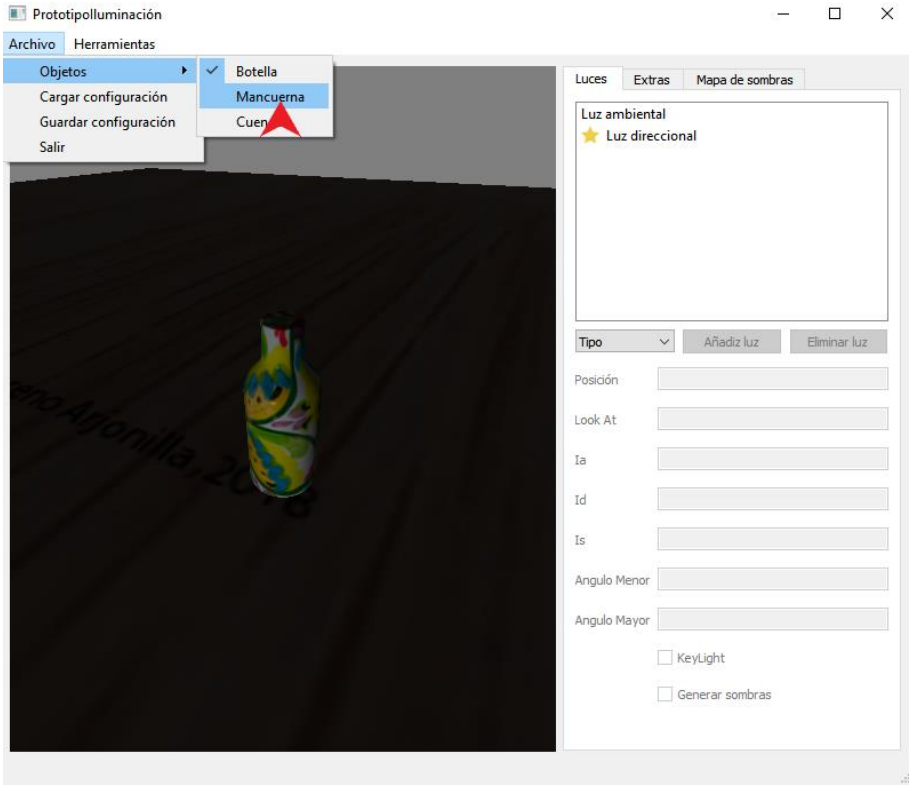


Ilustración 4.42 Manual de usuario: Cambiar objeto. Click en el objeto

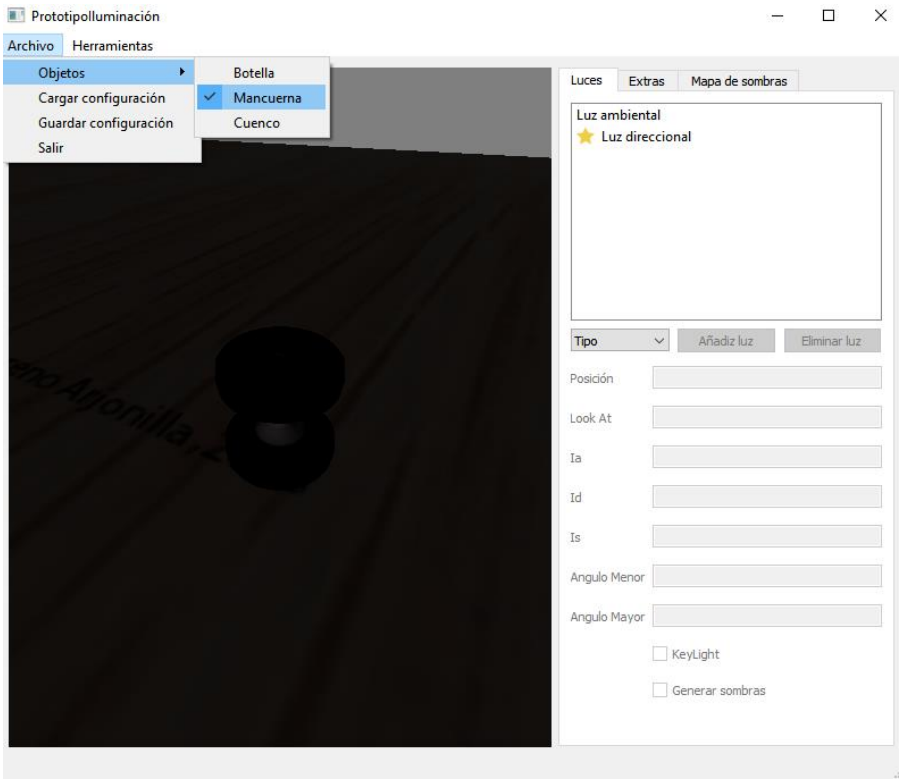


Ilustración 4.43 Manual de usuario: Cambiar objeto. Resultado

- Cargar configuración:

Para cargar un archivo deberá de ir al menú “Archivo” y a la opción “Cargar configuración”. Se le abrirá un explorador de archivos donde deberá de elegir un fichero .txt con la configuración. Una vez seleccione el archivo la escena se recargará automáticamente.

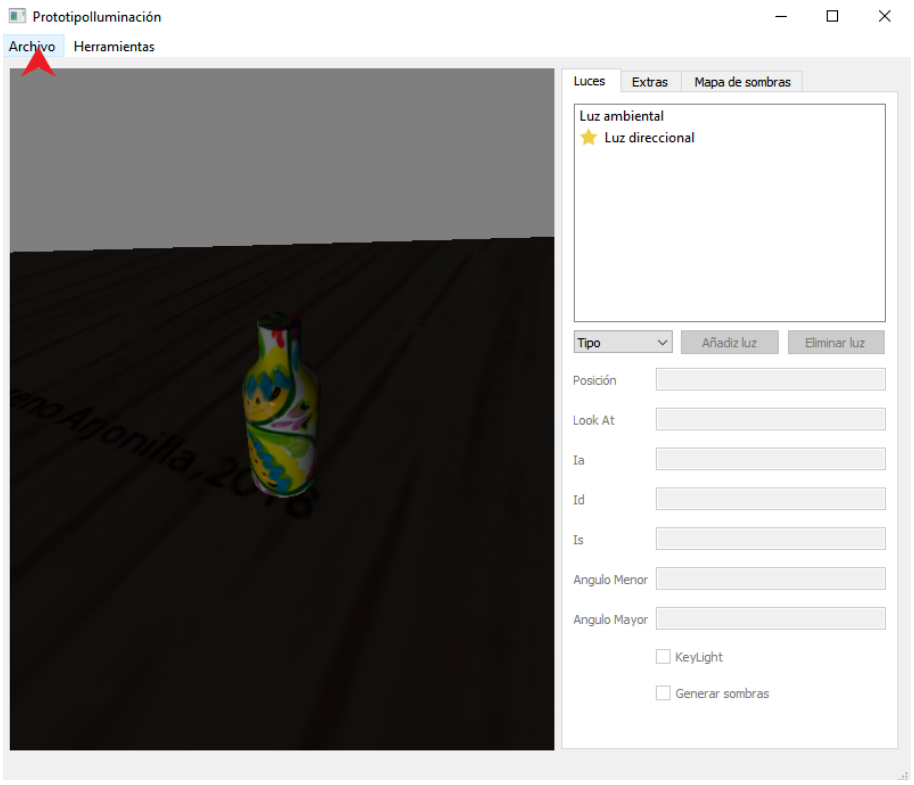


Ilustración 4.44 Manual de usuario: Cargar configuración. Desplegar menú “Archivo”

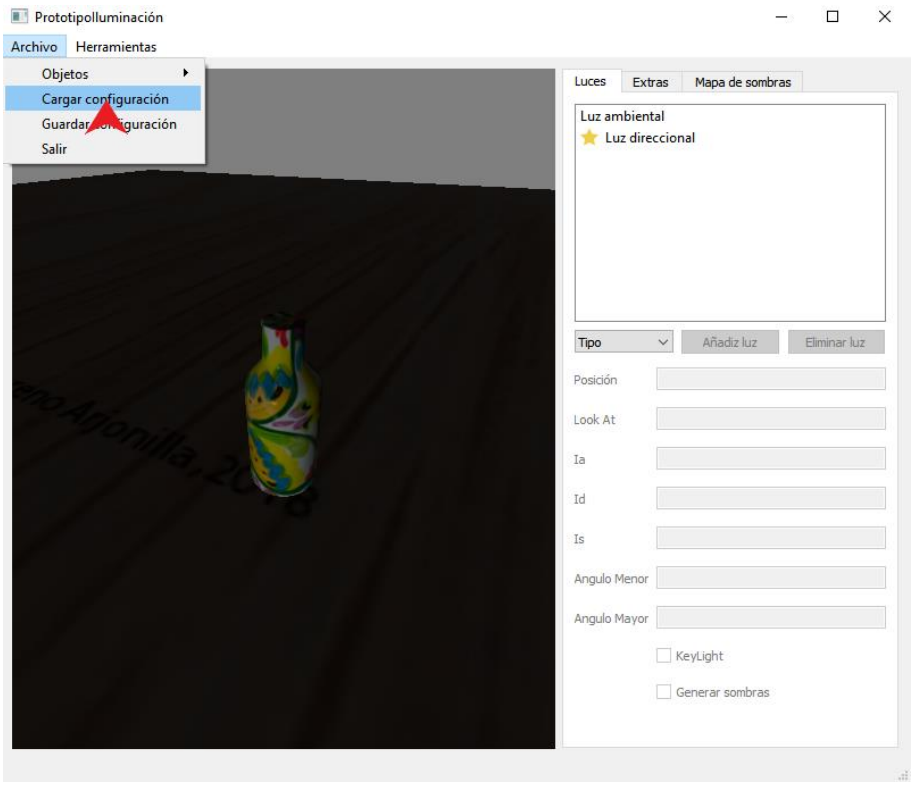


Ilustración 4.45 Manual de usuario: Cargar configuración. Click en la opción "Cargar configuración"

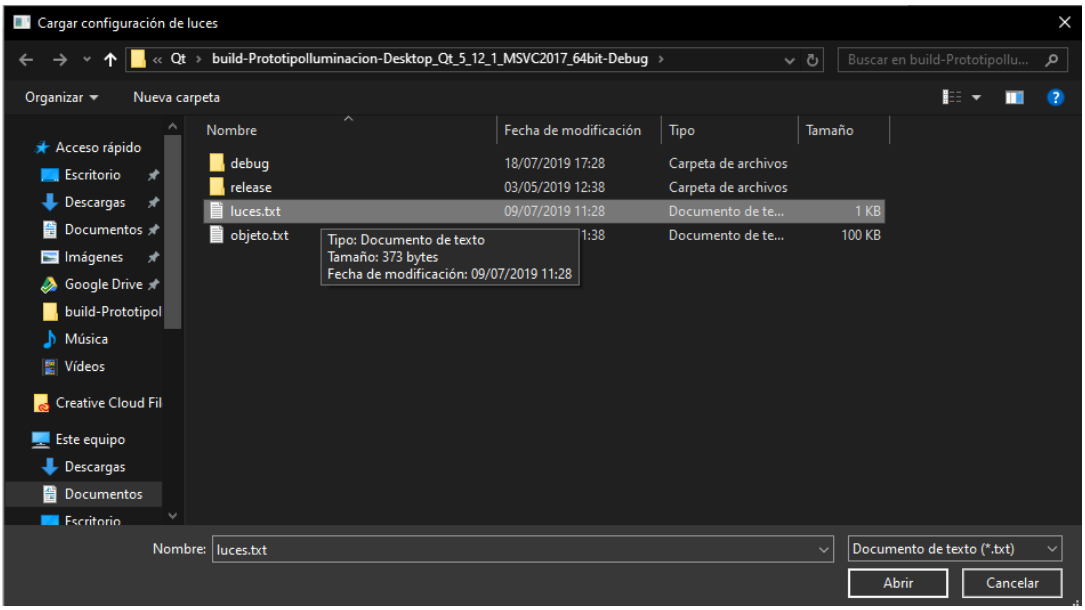


Ilustración 4.46 Manual de usuario: Cargar configuración. Selección archivo

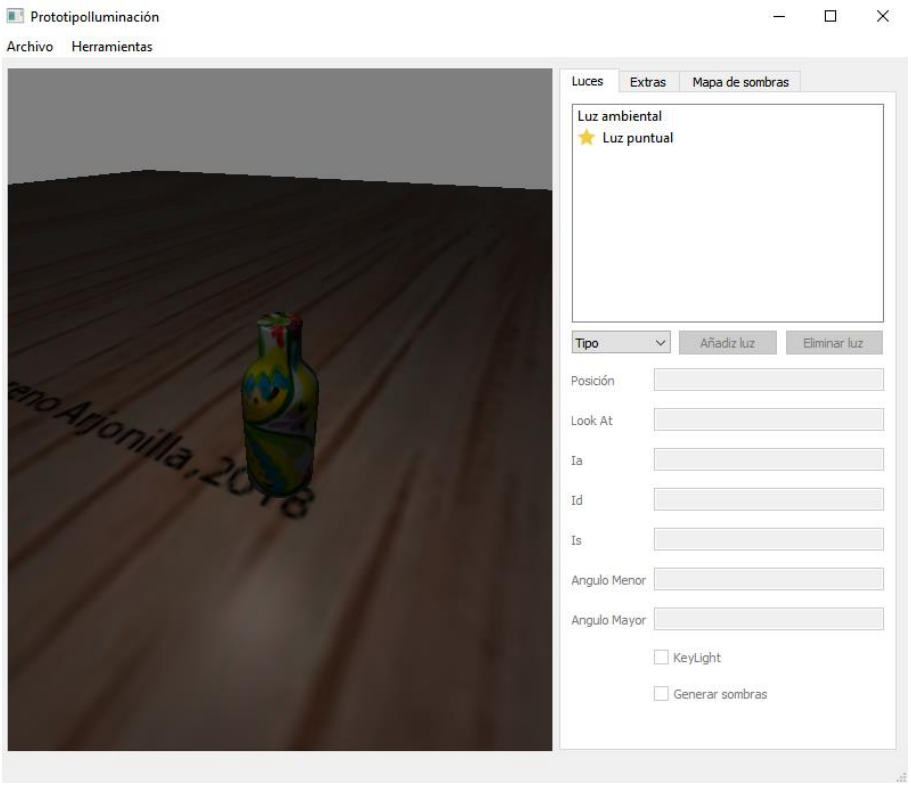


Ilustración 4.47 Manual de usuario: Cargar configuración. Resultado

- Guardar configuración:

Para guardar su configuración actual de la escena deberá de ir al menú “Archivo” y a la opción “Guardar configuración”. Se le abrirá un explorador de archivos donde podrá elegir la ruta y nombre de su fichero.

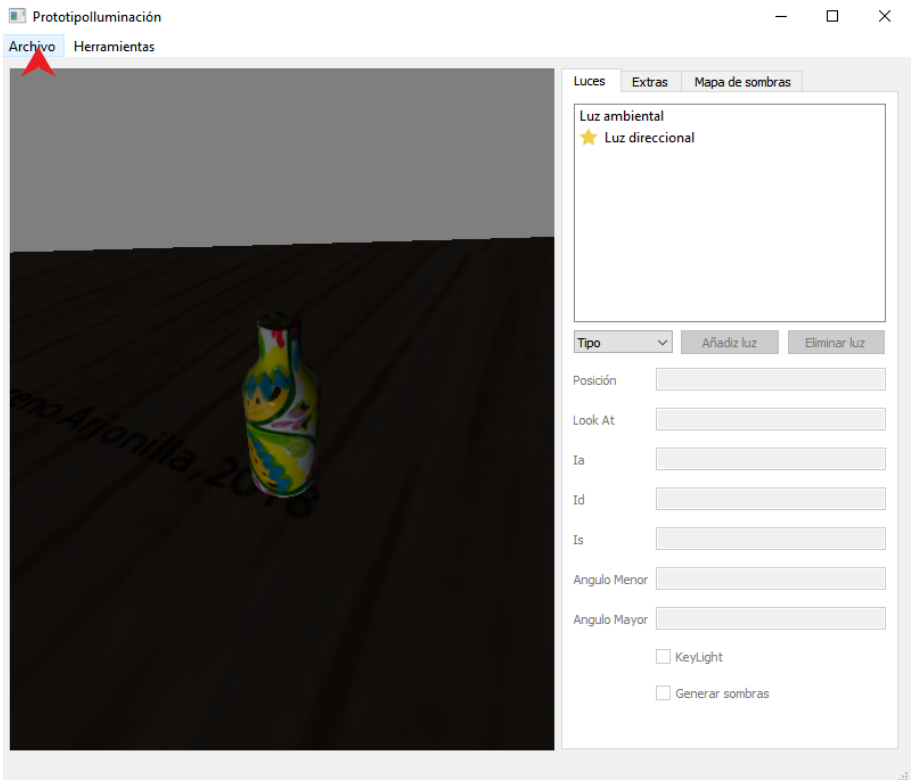


Ilustración 4.48 Manual de usuario: Guardar configuración. Desplegar menú "Archivo"

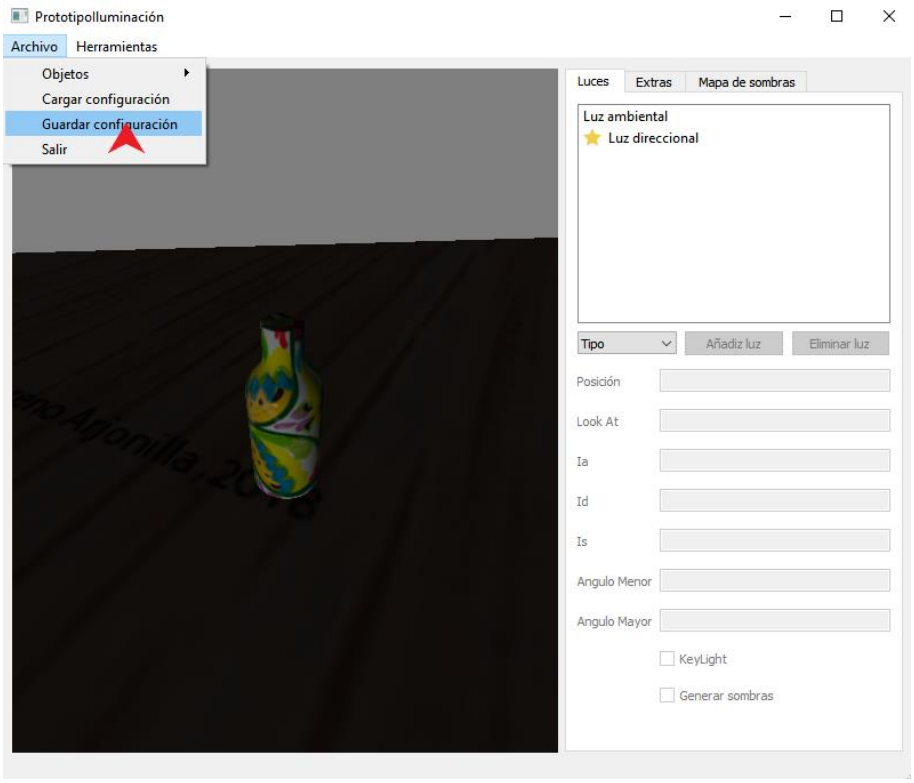


Ilustración 4.49 Manual de usuario: Guardar configuración. Click en la opción "Guardar configuración"

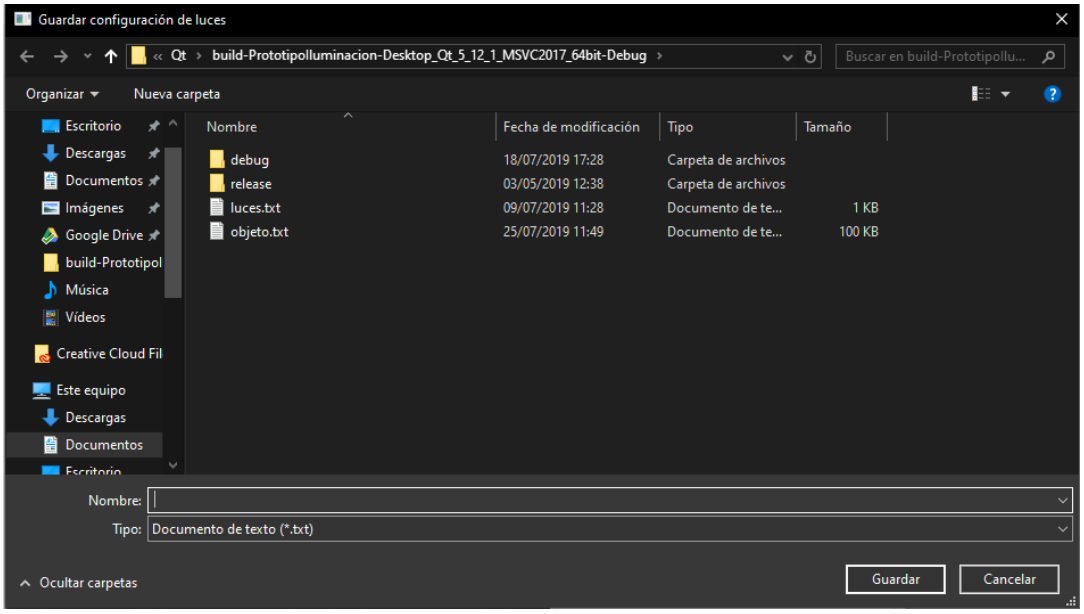


Ilustración 4.50 Manual de usuario: Guardar configuración. Ruta y nombre del archivo

- Salir de la aplicación:

  Para salir de la aplicación puede hacer uso de la "X" en la esquina superior derecha o, ir al menú "Archivo" y hacer click en la opción "Salir".

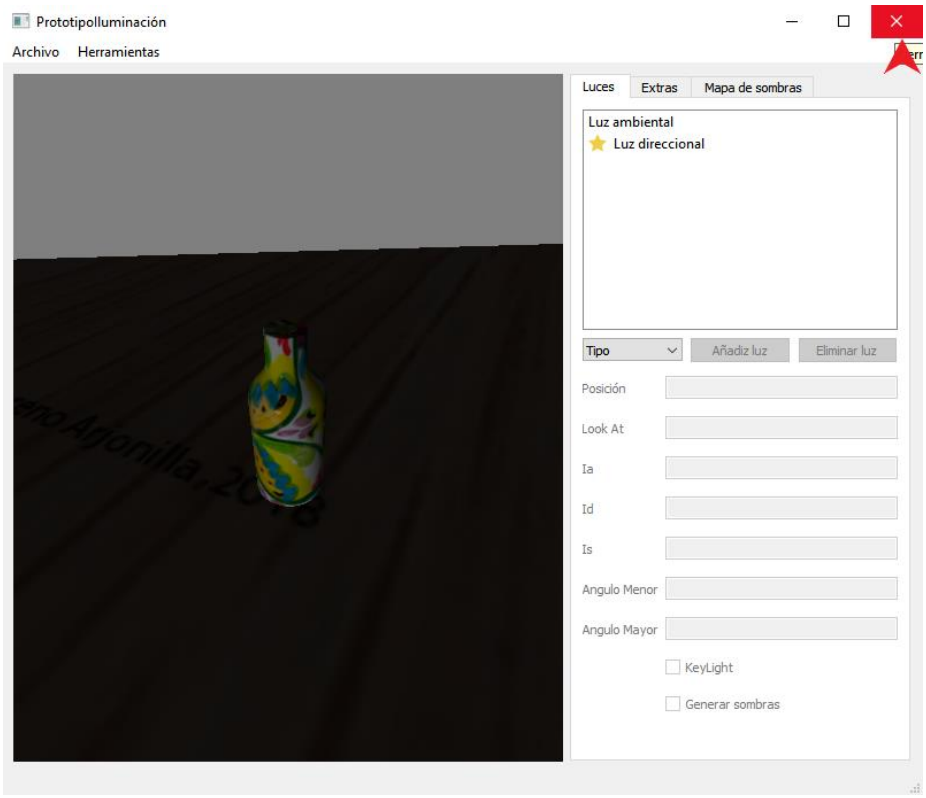


Ilustración 4.51 Manual de usuario: Salir de la aplicación. Opción 1. Click en la "X"

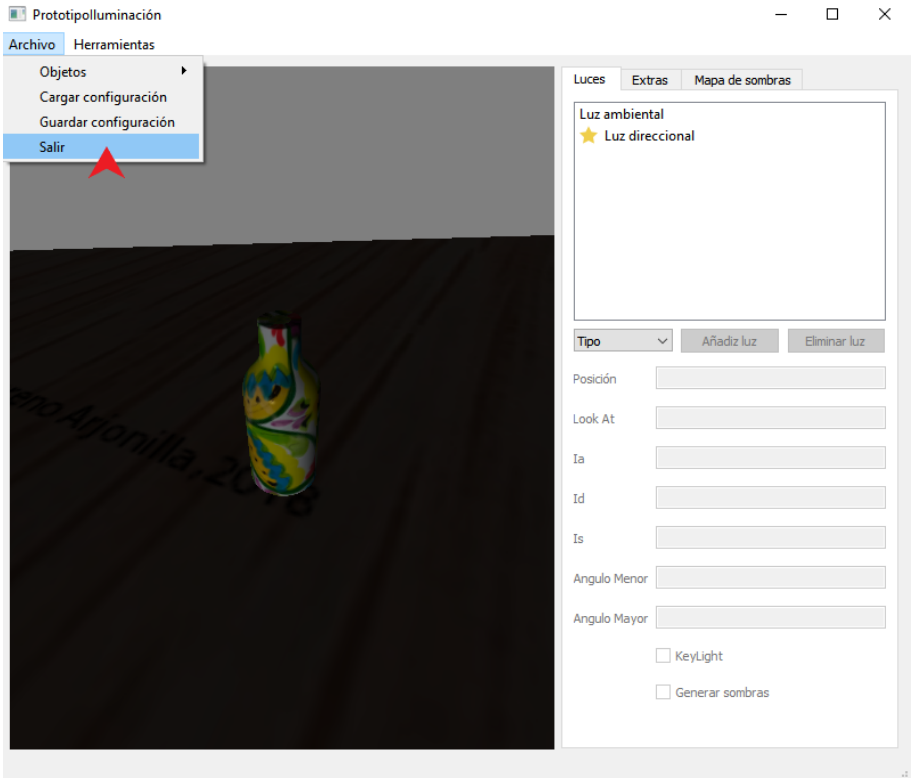


Ilustración 4.52 Manual de usuario: Salir de la aplicación. Opción 2. Desplegar menú "Archivo" y click en "Salir"

4.3 - Guía original del Trabajo Fin de Título

El documento adjunto puede ser descargado a partir del siguiente enlace.

Guía del Trabajo Fin de Grado

5 - BIBLIOGRAFÍA

Pressman, R. (2010). *Ingeniería del Software.* McGraw-Hill.

Virginia Bowman Wissler. Illumitaded pixels. The why, what and how of digital lighting. Course Technology PTR.

Ron Foster. Real-Time Shader Programming. Morgan Kaufmann.

Thomas Akenine-Möller, Eric Haines y Natty Hoffman. Real-Time Rendering 3ªEd. A K Peters/CRC Press.

Jacobo Rodríguez. GLSL Essentials. Packt Publishing.

David Wolff. OpenGL 4.0 Shading Language Cookbook. Packt Publishing.

Apuntes y diapositivas de la asignatura de Programación de Aplicaciones Gráficas.

LearnOpenGL.

OpenGL tutorials.

Kronos.

Qt Documentation.