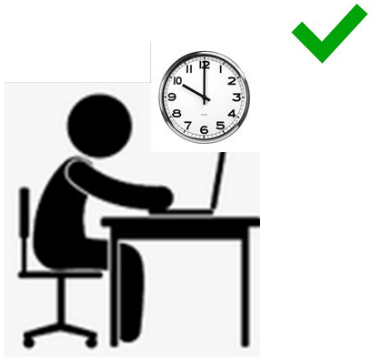
Universidad Intercontinental de la Empresa

Asignatura

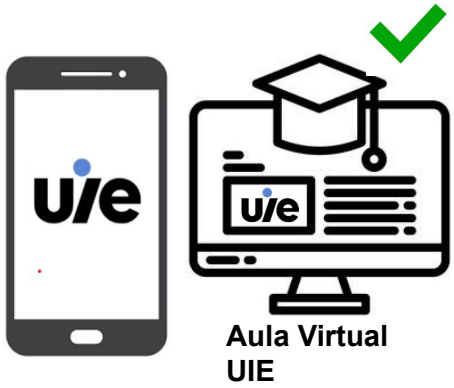
Computación gráfica

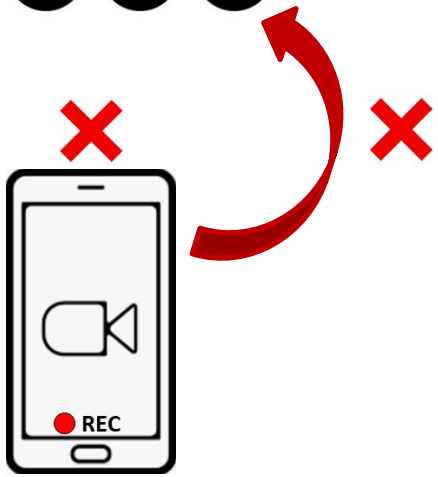
Profesor

David Cereijo Graña




Recordar


Aula Virtual
UIE


REC


Recordar

Participar

Universidad Intercontinental de la Empresa

Sesión 11

Unidad III

Gráficos 3D

Tema 3.5

Construcción computacional de gráficos 3D

Modo inmediato de OpenGL

El **modo inmediato** es un método de renderizado de OpenGL donde las primitivas gráficas se definen y envían a la GPU en tiempo real durante la ejecución del programa.

Este enfoque se basa en especificar cada vértice de una primitiva individualmente dentro de bloques de código delimitados por llamadas a **`glBegin()`** y **`glEnd()`**.

```
# Inicia la definición de triángulos
glBegin(GL_TRIANGLES)
# Define el color del vértice 1
glColor3f( red: 1.0,  green: 0.0,  blue: 0.0)
# Define las coordenadas del vértice 1
glVertex3f( x: 0.0,  y: 1.0,  z: 0.0)
# Define el color del vértice 1
glColor3f( red: 0.0,  green: 1.0,  blue: 0.0)
# Define las coordenadas del vértice 1
glVertex3f(-1.0, -1.0,  z: 0.0)
# Define el color del vértice 1
glColor3f( red: 0.0,  green: 0.0,  blue: 1.0)
# Define las coordenadas del vértice 1
glVertex3f( x: 1.0, -1.0,  z: 0.0)
# Finaliza la definición de triángulos
glEnd()
```

Características del modo inmediato

El modo inmediato presenta la siguientes características:

1. **Sencillo y fácil de usar:** adecuado para iniciarse y llevar a cabo tareas simples de renderizado.
2. **Procedural:** las primitivas se definen de manera secuencial en el código, lo que facilita la comprensión de cómo se construyen las figuras gráficas.
3. **Máquina de estados:** en este modo, OpenGL funciona como una máquina de estados, donde las configuraciones (como colores, transformaciones, etc.), afectan a las primitivas definidas posteriormente.
4. **Ineficiente para renderizados complejos:** cada llamada a glVertex() y otras funciones similares envía datos directamente a la GPU, lo que resulta en una elevada sobrecarga y baja eficiencia para escenas complejas.

OpenGL moderno: VBOs, VAOs y shaders

A partir de OpenGL 3.0 y versiones posteriores, el modo inmediato ha sido declarado oficialmente *deprecated*, y el renderizado se basa en en métodos más eficientes y flexibles:

- **Vertex Buffer Objects (VBOs):** permiten almacenar los datos de vértices en la memoria de la GPU, reduciendo la necesidad de enviar datos repetidamente desde la CPU.
- **Vertex Array Objects(VAOs):** simplifican la gestión de los VBOs al encapsular la configuración de los atributos de los vértices.
- **Shaders (Vertex shaders y Fragment shaders):** proporcionan un control completo sobre el pipeline de renderizado, permitiendo su programación.

¿Qué es un shader?

Un *shader* o sombreador es un pequeño programa que se ejecuta en la GPU y se encarga de procesar diferentes aspectos de los gráficos.

Existen distintos tipos de *shaders*, pero los principales son:

- ***Vertex shader.***
- ***Fragment shader.***

Vertex shader

El ***vertex shader*** o sombreador de vértices es el responsable de procesar cada vértice de un modelo **3D**. Su principal tarea es transformar las posiciones de los vértices desde el espacio del modelo hasta el espacio de recorte (*clip space*), aplicando transformaciones como proyecciones, traslaciones, escalados y rotaciones.

- **Entrada:** datos de cada vértice.
- **Salida:** vértices transformados y otros atributos.

Fragment shader

El ***fragment shader*** o sombreador de fragmentos se encarga de procesar cada fragmento (píxel potencial) generado por la rasterización de primitivas. Su principal tarea es determinar el color final de cada fragmento, aplicando iluminación, texturas, y otros efectos.

- **Entrada:** datos de salida del *vertex shader*, como posición, normales, coordenadas de textura..
- **Salida:** color final del fragmento.

Fragment shader

	Vertex shader	Fragment shader
Unidad de procesamiento	Vértices.	Fragmentos (píxel potencial).
Frecuencia de ejecución	Una vez por cada vértice.	Una vez por cada fragmento.
Funcionalidad	Transformar la posición de los vértices y pasar atributos al fragment shader	Aplicar efectos visuales y determinar el color final del fragmento.
Datos de entrada	Posición, color, normales, coordenadas de textura…	Vértices transformados por el vertex shader y otros atributos.
Datos de salida	Posición transformada y otros atributos.	Color final del fragmento.

Dibujando un punto en modo inmediato

Dibujando un punto en modo inmediato

```
1  import pygame
2  from pygame.locals import *
3  from OpenGL.GL import *
4  from OpenGL.GLU import *
5  from configuracion import *
6
7  PUNTO_X = 0.0
8  PUNTO_Y = 0.0
9  PUNTO_Z = 0.0
```

Dibujando un punto en modo inmediato

```
11  def inicializar_escena():
12      # Inicializa Pygame
13      pygame.init()
14      # Crea la ventana gráfica
15      pygame.display.set_mode( size: (SCREEN_WIDTH, SCREEN_HEIGHT), DOUBLEBUF | OPENGL)
16      # Establece el título de la ventana
17      pygame.display.set_caption('Dibujado de un punto en modo inmediato')
18      # Configura la proyección en perspectiva
19      gluPerspective(FOV, SCREEN_ASPECT_RATIO, NEAR_PLANE, FAR_PLANE)
20      # Desplaza la cámara hacia atrás
21      glTranslatef( x: 0.0,  y: 0.0, -5)
22      # Establece el tamaño del punto
23      glPointSize(TAMANO_PUNTO)
```

Dibujando un punto en modo inmediato

```
25  def dibujar_punto():  1 usage
26      # Inicia la definición de puntos
27      glBegin(GL_POINTS)
28      # Establece el color rojo
29      glColor3f(*COLOR_ROJO)
30      # Define la posición del punto en el centro
31      glVertex3f(PUNTO_X, PUNTO_Y, PUNTO_Z)
32      # Finaliza la definición de puntos
33      glEnd()
```

Dibujando un punto en modo inmediato

```
35  inicializar_escena()
36
37  while True:
38      for event in pygame.event.get():
39          if event.type == pygame.QUIT:
40              pygame.quit()
41
42      # Limpia los buffer de color y profundidad
43      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
44
45      # Dibuja el punto
46      dibujar_punto()
47
48      # Intercambia los buffers
49      pygame.display.flip()
50      # Espera
51      pygame.time.wait(int(MILLISECONDS_PER_SECOND / FPS))
```

main_punto_modo_inmediato.py

Dibujando un punto con shaders

Dibujando un punto con shaders

El proceso de dibujado con shaders consta de los siguientes pasos:

1. **Definición de shaders:**
    - **Vertex shader:** procesa cada vértice individualmente, aplicando transformaciones geométricas.
    - **Fragment shader:** determina el color y otros atributos de cada fragmento.
2. **Compilación de shaders:**
    - Cada shader (vertex y fragment) debe ser compilado por separado.
3. **Enlazado de shaders:**
    - Los shaders compilados se enlazan en un único programa que shaders que se utilizará para llevar a cabo el renderizado.

Dibujando un punto con shaders

4. **Crear y cargar buffers:**
   - **Vertex Buffer Object (VBO):** almacena los datos de los vértices e la memoria de la GPU para un acceso rápido durante el renderizado.
   - **Element Buffer Object (EBO) (Opcional):** almacena índices que definen cómo se conectan los vértices para formar primitivas (como triángulos).
5. **Configurar el Vertex Array Object (VAO):**
   - Guarda las configuraciones de los atributos de vértices y cómo se deben interpretar los datos almacenados en los VBOs y EBOs.
6. **Configuración de Uniforms y Atributos:**
   - **Uniforms:** variables globales que permanecen constantes para todas las primitivas procesadas por el shader.
   - **Atributos:** variables que cambian para cada vértice, como posición, color, normales…

Dibujando un punto con shaders

```
1  import pygame
2  from pygame.locals import *
3  from OpenGL.GL import *
4  from OpenGL.GLU import *
5  from configuracion import *
6
7  PUNTO_X = 0.0
8  PUNTO_Y = 0.0
9  PUNTO_Z = 0.0
```

Dibujando un punto con shaders

```
11  # Código fuente del Vertex Shader
12  vertex_shader_code = """
13  #version 330
14  in vec3 position;
15  void main()
16  {
17      gl_Position = vec4(position, 1.0);
18  }
19  """
```

Dibujando un punto con shaders

```
21  # Código fuente del Fragment Shader
22  fragment_shader_code = """
23  #version 330
24  out vec4 fragment_color;
25  void main()
26  {
27      fragment_color = vec4(1.0, 0.0, 0.0, 1.0);
28  }
29  """
```

Dibujando un punto con shaders

```
31  def compilar_shader(shader_tipo, shader_fuente):  2 usages
32      shader_id = glCreateShader(shader_tipo)
33      glShaderSource(shader_id, shader_fuente)
34      glCompileShader(shader_id)
35      # Verificamos la compilación
36      compilacion_exitosa = glGetShaderiv(shader_id, GL_COMPILE_STATUS)
37      if not compilacion_exitosa:
38          mensaje_error = "\n" + glGetShaderInfoLog(shader_id).decode("utf-8")
39          glDeleteShader(shader_id)
40          raise Exception(mensaje_error)
41      return shader_id
```

Dibujando un punto con shaders

```
43  def crear_programa():  1 usage
44      vertex_shader_id = compilar_shader(GL_VERTEX_SHADER, vertex_shader_code)
45      fragment_shader_id = compilar_shader(GL_FRAGMENT_SHADER, fragment_shader_code)
46      programa_id = glCreateProgram()
47      glAttachShader(programa_id, vertex_shader_id)
48      glAttachShader(programa_id, fragment_shader_id)
49      glLinkProgram(programa_id)
50      # Verificamos el enlazado
51      enlazado_exitoso = glGetProgramiv(programa_id, GL_LINK_STATUS)
52      if not enlazado_exitoso:
53          mensaje_error = "\n" + glGetShaderInfoLog(shader_id).decode("utf-8")
54          raise Exception(mensaje_error)
55      # Los shaders individuales pueden ser eliminados después del enlace
56      glDeleteShader(vertex_shader_id)
57      glDeleteShader(fragment_shader_id)
58      return programa_id
```

Dibujando un punto con shaders

```
60  def inicializar_escena():
61      # Inicializa Pygame
62      pygame.init()
63      # Crea la ventana gráfica
64      pygame.display.set_mode( size: (SCREEN_WIDTH, SCREEN_HEIGHT), DOUBLEBUF | OPENGL)
65      # Establece el título de la ventana
66      pygame.display.set_caption('Dibujado de un punto con shaders')
67      # Establece el tamaño del punto
68      glPointSize(TAMANO_PUNTO)
```

Dibujando un punto con shaders

```
71      inicializar_escena()
72
73      # Creamos el programa con los shaders
74      programa = crear_programa()
75      # Usamos el programa creado
76      glUseProgram(programa)
```

Dibujando un punto con shaders

```
85    # Genera un identificador único para un nuevo Vertex Buffer Object (VBO)
86    VBO = glGenBuffers(1)
87
88    # Enlaza el VBO generado como el buffer activo para almacenar datos de vértices
89    glBindBuffer(GL_ARRAY_BUFFER, VBO)
90
91    # Carga los datos de los vértices en el buffer actualmente enlazado (VBO)
92    # 'puntos.nbytes' especifica el tamaño de los datos en bytes
93    # 'puntos' son los datos de los vértices que se cargarán
94    # 'GL_STATIC_DRAW' indica que los datos no cambiarán con frecuencia
95    glBufferData(GL_ARRAY_BUFFER, puntos.nbytes, puntos, GL_STATIC_DRAW)
```

Dibujando un punto con shaders

```
99    # Definimos el Vertex Array Object (VAO)
100   # Genera un identificador único para un nuevo Vertex Array Object (VAO)
101   VAO = glGenVertexArrays(1)
102
103   # Enlaza el VAO generado como el Vertex Array Object activo
104   glBindVertexArray(VAO)
105
106   # Habilita el atributo de vértice en la ubicación 0
107   # Esto indica que se usará este atributo para almacenar datos de vértices
108   glEnableVertexAttribArray(0)
109
110   # Define cómo se interpretarán los datos de vértices almacenados en el VBO
111   # Parámetros:
112   # 0 - Índice del atributo de vértice
113   # 3 - Número de componentes por atributo de vértice (x, y, z)
114   # GL_FLOAT - Tipo de dato de cada componente
115   # GL_FALSE - No normalizar los datos
116   # 0 - Tamaño del stride (espacio entre atributos consecutivos)
117   # None - Desplazamiento inicial en el buffer
118   glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, None)
```

Dibujando un punto con shaders

```
120     while True:
121         for event in pygame.event.get():
122             if event.type == pygame.QUIT:
123                 pygame.quit()
124
125         glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
126
127         glBindVertexArray(VAO)
128         # Dibuja un solo punto
129         glDrawArrays(GL_POINTS, first: 0, count: 1)
130
131         # Intercambia los buffers
132         pygame.display.flip()
133         # Espera
134         pygame.time.wait(int(MILLISECONDS_PER_SECOND / FPS))
```

main_punto_shaders.py

main_punto_3_shaders.py

Shader vs modo inmediato

- Tal y como hemos podido ver en los ejemplos, el código del shader se ejecuta por vértice (*vertex shader*) o por fragmento (*fragment shader*).
- Esto significa que el código que escribimos lo debemos escribir solo para un vértice o fragmento.
- Además, no es necesario escribir loops que procesen todos los vértices del modelo o todos los fragmentos que deben aparecen en la pantalla, sino que simplemente pasamos los datos a la GPU, que es la que se encarga de procesarlos en paralelo.

GLSL

GLSL (OpenGL Shading Language)

GLSL es un lenguaje de programación específico diseñado para escribir shaders que se ejecutan en la GPU. Sus principales características son:

- **Sintaxis similar a C:** las líneas se deben finalizar con **";"**, y se deben abrir y cerrar llaves **"{}"** para contener bloques de código (esto suele ser causa de errores al venir de Python).
- **Ejecución paralela eficiente:** diseñado para ejecutarse de manera eficiente en arquitecturas paralelas de las GPU.
- **Variables de entrada y salida:** utiliza atributos para recibir datos de los vértices y variables uniformes para recibir datos constantes, comunes a múltiples primitivas.
- **Funciones integradas:** incluye una amplia variedad de funciones matemáticas y utilidades específicas para gráficos, como transformaciones vectoriales, operaciones con matrices y cálculos de iluminación.

Tipos de datos

GLSL utiliza diferentes tipos de datos, diseñados específicamente para facilitar la programación de shaders en el pipeline gráfico de OpenGL.

Estos tipos de datos se pueden categorizar en:

- Tipos de datos básicos (S*calars*).
- Tipos de datos vectoriales (*Vectors*).
- Tipos de datos matriciales (*Matrices*).
- Tipos de datos sampler (*Samplers*).
- Tipos de datos arrays (*Arrays*).
- Tipos de datos estructurados (*Structs*).

Tipos de datos básicos (Scalars)

Los tipos de datos básicos (no vectoriales) de GLSL son:

TIPO DE DATO	DESCRIPCIÓN	EJEMPLO DE USO
`bool`	Valores booleanos (**true** o **false**).	Condicionales, flags, activación/desactivación de características.
`int`	Enteros con signo de **32** bits.	Valores enteros que requieren signo, como por ejemplo una dirección.
`uint`	Enteros sin signo de **32** bits.	Indexación de Arrays, contadores
`float`	Coma flotante de precisión simple (**32** bits).	Coordenadas y transformaciones.
`double`	Coma flotante de precisión doble (**64** bits).	Cálculos de alta precisión.

Tipos de datos vectoriales (Vectors)

Cada uno de los tipos básicos tiene un equivalente vectorial de **2**, **3** y **4** componentes (el dígito **n** puede ser **2**, **3** o **4**).

TIPO DE DATO	DESCRIPCIÓN
`bvecn`	Vector de **2**, **3** o **4** booleanos: **bvec2**, **bvec3**, **bvec4**.
`ivecn`	Vector de **2**, **3** o **4** enteros con signo de **32** bits: **ivec2**, **ivec3**, **ivec4**.
`uvecn`	Vector de **2**, **3** o **4** enteros sin signo de **32** bits: **uvec2**, **uvec3**, **uvec4**.
`vecn`	Vector de **2**, **3** o **4** números en coma flotante de precisión simple (**32** bits): **vec2**, **vec3**, **vec4**.
`dvecn`	Vector de **2**, **3** o **4** números en coma flotante de precisión doble (**64** bits): **dvec2**, **dvec3**, **dvec4**.

Swizzling

El ***swizzling*** es una característica de **GLSL** que permite reorganizar, duplicar o extraer componentes individuales de vectores.

En **GLSL** cada vector tiene sus componentes etiquetados con letras que representan sus componentes:

- **x**, **y**, **z**, **w**: utilizados para coordenadas espaciales.
- **r**, **g**, **b**, **a**: utilizados para coordenadas de color..
- **s**, **t**, **p**, **q**: utilizados para coordenadas de texturas.

Estas letras son intercambiables y representan los mismos componentes. Por ejemplo, **x**, **r** y **s** se refieren al primer componente, **y**, **g** y **t** al segundo, etc.

Swizzling: reordenación y selección de componentes

El *swizzling* permite **reordenar** o **seleccionar** componentes individuales de un vector para formar un nuevo vector.

Ejemplos:

```
vec3 original = vec3(1.0, 2.0, 3.0);

vec2 nuevo = original.xy; // nuevo = vec2(1.0, 2.0)

vec3 reordenado = original.yxz; // reordenado = vec3(2.0, 1.0, 3.0)
```

Swizzling: duplicación de componentes

El *swizzling* también permite **duplicar** componentes para crear nuevos vectores ocn componentes repetidos.

Ejemplos:

```
vec3 original = vec3(1.0, 2.0, 3.0);

vec2 duplicado1 = original.xxxx; // duplicado1 = vec4(1.0, 1.0, 1.0, 1.0)

vec2 duplicado2 = original.yyy; // duplicado2 = vec3(1.0, 1.0, 1.0)
```

Swizzling: asignación a componentes específicos

Mediante el *swizzling* también es posible **asignar valores a componentes específicos** de un vector existente, sin afectar a los demás componentes.

Ejemplos:

```
vec4 color = vec4(1.0, 0.5, 0.2, 1.0);

color.rgb = vec3(0.0, 1.0, 0.0); // color = vec4(0.0, 1.0, 0.0, 1.0)

color.a = 0.5; // color = vec4(0.0, 1.0, 0.0, 0.5)
```

Reglas y restricciones del swizzling: la longitud debe ser consistente

Al crear un nuevo vector mediante *swizzling*, la longitud del vector resultante debe ser coherente con la cantidad de componentes especificados.

Ejemplos:

```
vec3 v = vec3(1.0, 2.0, 3.0);

vec2 a = v.xy; // Correcto

vec4 b = v.xyzw; // Incorrecto, pues v solo tiene 3 componentes.
```

Reglas y restricciones del swizzling: no se puede escribir en componenes duplicados

Es posible leer cualquier combinación de componentes, pero solo se puede escribir en componentes únicos.

Ejemplos:

```
v.xy = vec2(4.0, 5.0) // Correcto

v.xx = vec2(6.0, 7.0) // Incorrecto, no se puede escribir en x dos veces.
```

Tipos de datos matriciales (Matrices)

Además de los tipos de datos vectoriales, GLSL también tiene tipos matriciales. Todos los tipos almacenan valores en coma flotante de precisión simple o doble. (el dígito **n** puede ser **2**, **3** o **4**).

TIPO DE DATO	DESCRIPCIÓN
`matnxm`	Matriz de **n** columnas y **m** filas (ejemplos: `mat2x2`, `mat4x3`). Nótese que es al revés de la notación comúnmente utilizada en matemáticas.
`matn`	Abreviatura para `matnxn`, es decir, una matriz cuadrada de **n** columnas y **n** filas.

Tipos de datos Sampler

Los samplers son tipos de datos especiales utilizados para acceder a texturas en los shaders, que permite muestrear (sample) datos de manera eficiente.

Ejemplos:

- **sampler1D, sampler2D, sampler3D, samplerCube:** permiten acceder a texturas **1D**, **2D**, **3D** y texturas de cubo o *cubemaps*, respectivamente.
- **isampler2D, usampler2D, etc.:** permiten acceder a texturas con tipos de datos específicos (entero, entero sin signo…).

Tipos de datos Arrays

GLSL permite definir arrays de cualquier tipo de datos, incluyendo arrays de vectores, matrices y estructuras.

Ejemplos:

```
float valores[10];

vec3 posiciones[3];
```

Tipos de datos estructurados (Structs)

GLSL permite definir **estructuras** (*structs*) para agrupar múltiples variables bajo un mismo nombre, facilitando la organización de datos complejos.

Ejemplo:

```
Struct Material {
    vec3 difuso;
    vec 3 especular;
    float brillo
};
```

Modificadores de tipo

GLSL permite usar modificadores de tipo para alterar el comportamiento de las variables, especialmente en el contexto de la transmisión de datos entre shaders:

- **in** y **out**
- **uniform**
- **attribute** (deprecated) y **layout(location = X)**

Modificadores de tipo: in y out

Especifican la dirección de los datos entre shaders:

- **`in:`** datos de entrada al shader.
- **`out:`** datos de salida desde el shader.

Ejemplo:

```
// Vertex shader
out vec3 vertexColor;

// Fragment shader
in vec3 vertxColor;
```

Modificadores de tipo: uniform

Una variable **`uniform`** es una variable global, cuyo valor permanece constante para todos los vértices/fragmentos durante una llamada de dibujo.

Se utilizan pasar al shader valores como matrices de transformación, parámetros de iluminación, texturas, etc.

Ejemplo:

```
uniform mat4 matrizTransformacion;
uniform sampler2D textura;
```

Modificadores de tipo: attribute (deprecated) y layout(location = X)

En versiones anteriores de GLSL, se usaba **`attribute`** para datos de vértice. En versiones modernas, se usa **`layout(location = X)`** para especificar la ubicación del atributo.

Ejemplo:

```
layout(location = 0) in vec3 position;
layout(location = 1) in vec3 color;
```

main_punto_3_colores_shaders.py

main_triangulo_colores_shaders.py

main_triangulo_2_colores_shaders.py

main_triangulo_colores_shaders.py

Referencias bibliográficas

- Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., Phillips, R.L. (1993). *Introduction to Computer Graphics*. Addison-Wesley Publishing Company.
- Méndez, M. (2022). *Introducción a la graficación por computadora*. https://proyectodescartes.org/iCartesiLibri/PDF/GraficacionComputadora.pdf