Prerequisitos, materiales y compilación.

En esta sección se detallan las herramientas software necesarias para realizar las prácticas en los sistemas operativos Linux (Ubuntu u otros), macOS (de Apple) y Windows (de Microsoft), así como los recursos o materiales que se proporcionan para las prácticas y la forma de compilar el código.

0.1. Prerequisitos software

0.1.1. Sistema operativo Linux

Compiladores

Para hacer las prácticas de esta asignatura en Ubuntu, en primer lugar debemos de tener instalado algún compilador de C/C++. Se puede usar el compilador de C++ open source CLANG o bien el de GNU (ambos pueden coexistir). Podemos usar **`apt`** para instalar el paquete **`g++`** (compilador de GNU) o bien **`clang`** (compilador del proyecto LLVM).

Driver de la GPU

Respecto de la librería OpenGL, es necesario tener instalado algún driver de la tarjeta gráfica disponible (GPU) en el ordenador. Incluso si no hay tarjeta gráfica disponible (por ejemplo en un máquina virtual eso puede ocurrir), es posible usar OpenGL implementado en software (aunque es más lento). Lo más probable es que tu instalación ya cuente on el driver apropiado para la tarjeta gráfica.

En cualquier caso, en Ubuntu, si no tenemos instalado el driver y instalarlo, seria necesario en primer lugar averiguar el fabricante y modelo de la GPU, lo cual puede hacerse con esta orden:

```
sudo lshw -c video
```

Una vez conocido el fabricante y modelo de la GPU (nVidia,Intel,AMD), podemos seguir las intrucciones específicas de cada uno de los fabricantes para instalar el correspondiente driver.

Librería GLEW

La librería GLEW es necesaria en Linux para que las funciones de la versión 2.1 de OpenGL y posteriores pueden ser invocadas (inicialmente esas funciones abortan al llamarlas). GLEW se encarga, en tiempo de ejecución, de hacer que esas funciones esten correctamente enlazadas con su código.

Para instalarla, se puede usar el paquete debian **`libglew-dev`**. En Ubuntu, se usa la orden

```
sudo apt install libglew-dev
```

Librería GLFW

La librería GLFW se usa para gestión de ventanas y eventos de entrada. Se puede instalar con el paquete debian **`libglfw3-dev`**. En Ubuntu, se puede hacer con:

```
sudo apt install libglfw3-dev
```

Librería JPEG

Esta librería sirve para leer y escribir archivos de imagen en formato jpeg (extensiones **`.jpg`** o **`.jpeg`**). Se usará para leer las imágenes usadas como texturas. La librería se debe instalar usando el paquete debian **`libjpeg-dev`**. En Ubuntu, se puede hacer con la orden

```
sudo apt install libjpeg-dev
```

0.1.2. Sistema Operativo macOS

Compilador

En primer lugar, para poder compilar los programas fuente en C++, debemos asegurarnos de tener instalado y actualizado *XCode* (conjunto de herramientas de desarrollo de Apple, incluye compiladores de C/C++ y entorno de desarrollo). Una vez instalado, usaremos la implementación de OpenGL que se proporciona con XCode (la librería GLEW no es necesaria en este sistema operativo). Para verificar que se ha instalado correctamente, podemos probar a ejecutar en la línea de órdenes:

```
clang++ --version
```

(la versión es la 10.0.1 en ulio de 2019).

Uso de *homebrew* para CMake, GLFW y librería JPEG

Homebrew es un gestor de paquetes para macOS que permite instalar facilmente librerías de desarrollo. Nos permite instalar fácilmente la librería GLFW y la librería de lectura de archivos JPEG. Para compilar el código es necesario disponer de la orden **`cmake`**, la cual también se puede instalar con *Hombrew*.

Si no tenemos disponible *homebrew*, debemos de instalarlo antes según las instrucciones en la página Web (☞ brew.sh).

Para instalar CMake, GLFW y la librería de JPEG simplemente será necesario usar los correspondientes paquetes (☞ cmake, ☞ glfw y ☞ jpeg), usando estas órdenes en la línea de comandos:

```
brew install cmake
brew install glfw
brew install jpeg
```

Si todo va bien, esto dejará los archivos **.h** en **/usr/local/include** (o en el caso de GLFW, en la subcarpeta **GLFW** dentro de la anterior), y los archivos **.a** o **.dylib** en **/usr/local/lib**. Para comprobar que **cmake** se ha instalado correctamente, teclea

```
cmake --version
```

Instalación de *CMake* sin *Hombrew*

La orden CMake se puede instalar con el archivo **.dmg** para macOS que se encuentra en esta página web:

☞ https://cmake.org/download/

Una vez descargado e instalado este archivo, se puede comprobar que está disponible en la línea de órdenes, escribiendo:

```
cmake --version
```

Instalación de GLFW sin *homebrew*

Para instalarla, se debe acceder a la página de descargas del proyecto GLFW:

☞ http://www.glfw.org/download.html

Aquí se descarga el archivo **.zip** pulsando en el recuadro titulado *source package*. Después se debe abrir ese archivo **.zip** en una carpeta nueva vacía. En ese carpeta vacía se crea una subcarpeta raiz (de nombre **glfw-...**). Después compilamos e instalamos la librería con estas órdenes:

```
cd glfw-....
cmake -DBUILD_SHARED_LIBS=ON .
make
sudo make install
```

Si no hay errores, esto debe instalar los archivos en la carpeta **/usr/local/include/GLFW** (cabeceras **.h**) y en **/usr/local/lib** (archivos de librerías dinámicas con nombres que comienzan con **libglfw** y con extensión **.dylib**,)

Instalación de la librería JPEG sin *homebrew*

Es necesario instalar los archivos correspondientes a la versión de desarrollo de la librería de lectura de archivos de imagen en formato JPEG. Se debe compilar el código fuente de esta librería, para ello basta con descargar el archivo con el código fuente de la versión más moderna a un carpeta nueva vacía, y después compilar e instalar los archivos. Se puede hacer con esta secuencia de órdenes:

```
mkdir carpeta-nueva-vacia
cd carpeta-nueva-vacia
curl --remote-name http://www.ijg.org/files/jpegsrc.v9b.tar.gz
tar -xzvf jpegsrc.v9b.tar.gz
cd jpeg-9b
```

```
./configure
make
sudo make install
```

Estas ordenes se refieren a la versión 9b de la librería, para futuras versiones habrá que cambiar **`9b`** por lo que corresponda. Si todo va bien, esto dejará los archivos **`.h`** en **`/usr/local/include`** y los archivos **`.a`** o **`.dylib`** en **`/usr/local/lib`**. Para poder compilar, debemos de asegurarnos de que el compilador y el enlazador tienen estas carpetas en sus paths de búsqueda, es decir, debemos de usar la opción **`-I/usr/local/include`** al compilar, y la opción **`-L/usr/local/lib`** al enlazar.

0.1.3. Sistema Operativo Windows

Las prácticas se pueden desarrollar en Windows.

Para compilar en Windows se puede usar *Microsoft Visual Studio* (la versión *Community* no tiene coste alguno). Se debe de seleccionar las componentes para desarrollo de aplicaciones de escritorio. Visual Studio incorpora **`cmake`**, así como los archivos de cabecera (para incluir) y librerías (para enlazar) de OpenGL.

Para compilar GLEW y GLFW se pueden obtener los archivos de cabeceras y librerías precompiladas para Windows de las páginas Web de estas librerías. Sin embargo, en el momento de redactar esto no están todavía preparados y probados completamente los archivos de configuración de cmake para windows.

0.1.4. Uso de *VS Code* de Microsoft

El programa *VS Code* es un editor de código fuente de *Microsoft* gratuito y que puede usarse en Linux, macOS y Windows. Este programa no incorpora un compilador, en su lugar usa alguno que el usuario haya instalado en su ordenador. Entre los archivos que se entregan hay archivos de configuración que facilitan la tarea de editar, compilar, ejecutar y depurar el código de prácticas usando *VS Code* y el compilador correspondiente.

La posibilidad de usar *VS Code* no excluye a otros editores (p.ej. *atom*), y en cualquier caso el programa siempre se puede compilar en la línea de órdenes con **`cmake`** (lo vemos a continuación).

0.2. Materiales y compilación

0.2.1. Materiales para las prácticas. Entregas.

Los archivos que se proporcionan se encuentran organizados en estas carpetas y sub-carpetas:

- **`materiales`**: esta carpeta contiene archivos de código fuente, imágenes, modelos 3D (archivos **`.ply`**) y otros, que se deben usar tal cual se entregan, sin que el alumno deba modificar ninguno de ellos. Tiene estas sub-carpetas:
  - **`src-cpp`**: archivos de código fuente C++ (**`.cpp`** y **`.h`**)
  - **`src-shaders`**: archivos de código fuente GLSL (**`.glsl`**)
  - **`plys`**: archivos con modelos 3D en formato PLY (**`.ply`**)
  - **`imgs`**: archivos de imágenes para texturas en formato JPEG (**`.jpg`** o **`.jpeg`**)

- **`src`**: esta carpeta contiene código fuente que debe ser completado por el alumno, son archivos **`.cpp`** y **`.h`**.
- **`builds`**: carpeta con archivos de configuración para compilar desde la línea órdenes (con **`cmake`**) y para editar el código fuente con VS Code de Microsoft. Tiene dos subcarpetas, se debe trabajar en la que corresponda al sistema operativo que usemos:
  - **`linux`**: para usar en Ubuntu u otros linux.
  - **`macos`**: para usar en macOS de Apple.

  En estas subcarpetas es donde se aloja el archivo ejectuable correspondiente al sistema operativo.
- **`archivos-alumno`**: archivos recopilados o creados por el alumno, distintos de los archivos fuente en la carpeta **`src`**. Aquí el alumno debe incluir archivos de imagen o archivos con modelos PLY que él mismo haya recopilado. También debe incluir archivos PDF o imagenes que debe crear y que forman parte de las entregas, según se describen en el guión de prácticas.

Estas carpetas se deben incluir en una carpeta nueva, que llamaremos *carpeta raiz* y cuyo nombre puede ser cualquiera.

La entregas de prácticas consisten en subir los archivos fuente de la carpeta **`src`** y los archivos de la carpeta **`archivos-alumno`**, si hay alguno. En ningún caso se deben subir archivos objeto o ejecutables (resultados de la compilación que son específicos del s.o. y arquitectura hardware usados por el alumno), tampoco ningún archivo que esté en la carpeta **`materiales`**, ni archivos de configuración de **`.vscode`**.

0.2.2. Compilación en la línea de órdenes

Se describe aquí el proceso de compilación en la línea de órdenes para macOS y Linux usando el programa *CMake*.

El uso de **`cmake`** requiere crear inicialmente los archivos de configuración de compilación necesarios, una sola vez, o después cuando queremos regenerar dichos archivos por cualquier motivo.

Sistemas operativos Linux y macOS, con *CMake*

Se deben de dar estos pasos:

- Ejecutar un terminal macOS o Linux, hacer **`cd`** a la carpeta **`builds/macos`** o **`builds/linux`**, según corresponda.
- Hacer **`cd`** a la subcarpeta **`cmake`**, la cual debe estar vacía inicialmente. En esa carpeta, escribir una sola vez esta orden:

  ```
  cmake ..
  ```

  Si no hay errores, esto generará en la carpeta **`cmake`** diversos archivos y sub-carpetas, entre ellos el archivo **`Makefile`** .
- Cada vez que se quiera compilar, en la carpeta **`cmake`**, hay que escribir esta orden:

  ```
  make
  ```

  Si no hay errores, esta orden debe de generar el archivo ejecutable **`pracs_ig_macos_exe`** o **`pracs_ig_linux_exe`** en la carpeta **`cmake`**. Para eliminar los archivos generados al compilar y forzar que se vuelva a compilar todo, se puede ejecutar:

```
make clean
```

y luego de nuevo **make**

- Para ejecutar el archivo resultado de la compilación, basta con hacer **cd** a la carpeta **cmake** y escribir

```
./pracs_ig_linux_exe
```

o bien

```
./pracs_ig_macos_exe
```

Hay que tener en cuenta que si añadimos algún archivo en la carpeta **src** (un nuevo archivo **.cpp** o **.h**) es necesario volver a generar los archivos de compilación en **cmake** para que el nuevo archivo se tenga en cuenta al compilar. Para eso, podemos vaciar la carpeta **cmake** y después volver a ejecutar **cmake ..** en ella.

0.2.3. Edición y compilación con *VS Code*

El código fuente se puede editar, compilar y ejecutar desde el editor *code*. Para ello se incluyen en **builds/macos** y en **builds/linux** sendos archivos de nombre **pracs-ig.code-workspace** (además de las carpetas de nombre **.vscode**). Para trabajar en el *espacio de trabajo* (*workspace*) de las prácticas se debe abrir un archivo **.code-workspace** (el que corresponda al sistema operativo) con el programa *code*.

Al abrir el espacio de trabajo se tiene la posibilidad de editar los códigos fuentes que se encuentran en la carpeta **src**, asimismo podemos ver (y no debemos editar), los fuentes que hay en **materiales**.

Para generar los archivos de compilación (si no estaban ya) hay que pinchar el sub-menu *Terminal*, dentro de eso la opción *Ejecutar tarea....* y finalmente seleccionar la opción *Regenerar archivos de compilación.* Esto hay que hacerlo una sola vez al principio, o bien cuando añadamos nuevos archivos a la carptea **src**.

En el sub-menú *Terminal* hay otras tareas posibles, y además hay una opción llamada *Ejecutar tarea de compilación*. Esa opción permite compilar y ejecutar las prácticas sin abandonar *code*.

Todas las acciones de compilar y ejecutar que realiza *code* se ponen en marcha por dicho programa usando las mismas órdenes que hemos visto para la compilación y ejecución desde la línea de órdenes.

0.3. Clases auxiliares

En esta sección se describebn algunas clases auxiliares que se usarán en las prácticas

0.3.1. Clases para tuplas de valores enteros o reales con 2,3 o 4 valores

En el archivo **tup_mat.h** se declaran varias clases para tuplas de valores numéricos (reales en simple o doble precisión y enteros con o sin signo). Cada tupla contiene unos pocos valores del mismo tipo (entre 2 y 4). Por cada tipo de valores y cada número de valores hay un tipo de tupla distinto (**Tupla2f**,**Tupla3f**,**Tupla3u**, etc...).

Aquí vemos ejemplos de uso de esta

```
// adecuadas para coordenadas de puntos, vectores o normales en 3D
// también para colores (R,G,B)
Tupla3f  t1 ;   // tuplas de tres valores tipo float
Tupla3d  t2 ;   // tuplas de tres valores tipo double

// adecuadas para la tabla de caras en mallas indexadas
Tupla3u  t4 ;   // tuplas de tres valores tipo unsigned

// adecuadas para puntos o vectores en coordenadas homogéneas
// también para colores (R,G,B,A)
Tupla4f  t5 ;   // tuplas de cuatro valores tipo float
Tupla4d  t6 ;   // tuplas de cuatro valores tipo double

// adecuadas para puntos o vectores en 2D, y coordenadas de textura
Tupla2f  t7 ;   // tuplas de dos valores tipo float
Tupla2d  t8 ;   // tuplas de dos valores tipo double
```

Este trozo código válido ilustra las distintas opciones, para creación, consulta y modificación de tuplas:

```
float      arr3f[3] = { 1.0, 2.0, 3.0 } ;
unsigned   arr3u[3] = { 1, 2, 3 } ;

// declaraciones e inicializaciones de tuplas
Tupla3f  a( 1.0, 2.0, 3.0 ), b, c(arr3f) ; // b indeterminado
Tupla3u  d( 1, 2, 3 ), e, f(arr3u) ;       // e indeterminado

// accesos de solo lectura, usando su posición o índice en la tupla (0,1,2,...),
// o bien varias constantes predefinidias para coordenadas (X,Y,Z) o colores (R,G,B):
float x1 = a(0), y1 = a(1), z1 = a(2),    //
      x2 = a(X), y2 = a(Y), z2 = a(Z),    // apropiado para coordenadas
      re = c(R), gr = c(G), bl = c(B) ;   // apropidado para colores

// conversiones a punteros
float *       p1 = a ; // conv. a puntero de lectura/escritura
const float * p2 = b ; // conv. a puntero de solo lectura

// accesos de escritura
a(0) = x1 ;  c(G) = gr ;

// escritura en un 'ostream' (cout) (se escribe como: (1.0,2.0,3.0)
cout << "la tupla 'a' vale: " << a << endl ;
```

En C++ se pueden sobrecargar los operadores binarios y unarios usuales (`+`, `-`, etc...) para operar sobre las tuplas de valores reales:

```
// declaraciones de tuplas y de valores escalares
Tupla3f  a,b,c ;
float    s,l ;

// operadores binarios y unarios de asignación/suma/resta/negación
a = b ;
a = b+c ;
a = b-c ;
a = -b  ;
```

```
// multiplicación y división por un escalar
a = 3.0f*b ;      // por la izquierda
a = b*4.56f ;     // por la derecha
a = b/34.1f ;     // mult. por el inverso

// otras operaciones
s = a.dot(b)        ;  // producto escalar (usando método dot)
s = a|b             ;  // producto escalar (usando operador binario barra )
a = b.cross(c)      ;  // producto vectorial (solo para tuplas de 3 valores)
l = a.lengthSq()    ;  // calcular módulo o longitud al cuadrado
a = b.normalized(); // hacer a= copia normalizada de b (a=b/modulo de b) (b no cambia)
```

1. Visualización de modelos simples.

1.1. Objetivos

Con esta práctica se quiere que el alumno aprenda:

- A crear estructuras de datos que permitan representar objetos 3D sencillos (mallas indexadas)
- A utilizar las órdenes para visualizar secuencias de vértices correspondientes a mallas indexadas.

1.2. Desarrollo

Para el desarrollo de esta práctica se entrega el esqueleto de una aplicación gráfica basada en eventos, mediante GLFW, y con la parte gráfica realizada por OpenGL. Para facilitar su uso, la aplicación permite abrir una ventana, mostrar unos ejes y mover una cámara básica.

El alumno deberá crear, como mínimo, los modelos de un **tetraedro** y un **cubo**. Para ello, creará las estructuras de datos que permitan representarlos mediante sus vértices y caras (con una estructura de tipo *malla indexada*). Asimismo, escribirá el código necesario para visualizar mallas indexadas usando *Vertex Buffer Objects* (VBOs) y *Vertex Array Objects* (VAOs) en **modo diferido**.

Las mallas se podrán visualizar usando cada uno de los tres **modos de visualización de polígonos**:

- Modo **puntos**: se visualiza un punto en la posición de cada vértice del modelo.
- Modo **alambre**: se viualiza como un segmento cada arista del modelo.
- Modo **sólido**: se visualizan los triángulos rellenos todos de un mismo color (plano).

En el modo sólido, además, se podrá activar o desactivar el **visualización de aristas**. Cuando el dibujo de aristas está activado, en el modo sólido se ven los triángulos opacos y además las aristas del modelo sobre ellos (todas de color negro).

1.3. Teclas a usar. Interacción.

El programa permite pulsar las siguientes teclas:

- **tecla p/P**: cambia la escena actual (pasa a la siguiente, o de la última a la primera). Hay una escena por cada práctica (ver más abajo).
- **tecla o/O**: cambia el objeto activo dentro de la escena actual (pasa al siguiente, o del último al primero)
- **tecla m/M**: cambia el modo de visualización de polígonos actual (pasa al siguiente, o del último al primero)
- **tecla w/W**: activa o desactiva la visualización de aristas.
- **tecla i/I**: activa o desactiva la iluminación (no tiene efecto hasta que no se implemente la práctica 4, antes de eso no hay iluminación)
- **tecla f/F**: cambia entre uso de normales de triángulo y uso de normales de vértices interpoladas para iluminación (igualmente, es útil únicamente a partir de la práctica 4).
- **tecla e/E**: activa o desactiva el dibujado de los ejes de coordenadas.

- **tecla q/Q o ESC**: terminar el programa.
- **teclas de cursor**: rotaciones de la cámara entorno al origen.
- **teclas +/-, av.pág/re.pág.**: aumentar/disminuir la distancia de la camara al origen (zoom).

Los eventos correspondientes se gestionan en la función gestora del evento de pulsar o levantar una tecla (`FGE_PulsarLevantarTecla`), dentro del arhivo `src/main.cpp`.

También se da la posibilidad de gestionar la camara con el ratón:

- **desplazar el ratón con el botón derecho pulsado**: rotaciones de la cámara entorno al origen.
- **rueda de ratón (scroll)**: aumentar/disminuir la distancia de la camara al origen (zoom).

Estos eventos se gestionan en las funciones gestoras de movimiento de ratón y de botones del raton (`FGE_MovimientoRaton` y `FGE_PulsarLevantarTecla`, respectivamente), también en el archivo `src/main.cpp` (la gestión de la cámara se estudiará en la práctica 5).

1.4. Estructura del código: funciones y clases.

El archivo `main.cpp` (carpeta `src`) incluye código de inizialización (creación de cauces y escenas, inicialización de GLFW y OpenGL, en ese orden), y después la llamada al bucle principal para gestión de eventos de GLFW (`BucleEventosGLFW`). En cada iteración del bucle, se invoca la función `VisualizarEscena`, para visualizar la escena actual (inicialmente solo hay una escena, en las siguientes prácticas iremos añadiendo escenas con otros tipos de objetos).

En la primera práctica se completará el código de creación (constructor) de la clase `Escena1`, derivada de `Escena`. La clase `Escena1` contendrá varios objetos de clases derivadas de `MallaInd`. Cada uno de esos objetos contiene las tablas de coordenadas de vértices, atributos e indices correspondientes a una malla indexada.

A continuación se detalla la funcionalidad de las distintas clases relevantes:

1.4.1. Contexto y modos de visualización y de envío

En el archivo `practicas.h` (dentro de la carpeta `src`) se declara la clase `ContextoVis` (por *Contexto de Visualización*), que contiene, como variables de instancia, distintos parámetros y variables de estado usados durante la visualización de objetos y escenarios en las prácticas. Los métodos encargados de visualizaer objetos tienen como parámetro un puntero (`cv`) a una instancia de esta clase. Esto permite usar una instancia para fijar todos esos parámetros, de forma que sean accesibles desde los citados métodos de visualización.

Entre otras cosas, el contexto de visualización contiene una variable que codifica el *modo de visualización de polígonos* actual (puntos, alambre, sólido, etc...), en concreto está en la variable de instancia `modo_visu`, que es un valor del tipo enumerado `ModosVisu`, tipo que también se declara en ese archivo de cabecera. Las declaración del tipo enumerado es así:

```
enum class ModosVisu
{
   relleno, lineas, puntos,
   num_modos  // al convertirlo a entero, da el número de modos
} ;
```