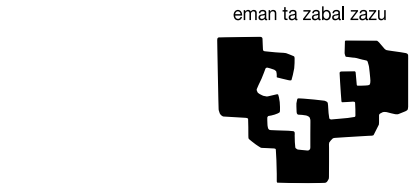


Universidad del País Vasco
Euskal Herriko Unibertsitatea

Mapeo de textura

Daniel Ruskov

October 1, 2020

Abstract

Gráficos por computador. Práctica 1 - creación de una aplicación para el mapeo de texturas implementada en lenguaje de programación C con la ayuda de OpenGL.

1 Introducción y objetivos

El objetivo principal es implementar una aplicación que mapea una textura sobre triángulos definidos por 3 puntos tridimensionales (x, y, z), donde $x, y \in [0, 500)$ y $z \in (-250, 250)$. A cada vértice de un triángulo se le asignan coordenadas $u, v \in [0, 1)$ de la textura a mapear.

La textura a utilizar es una foto en formato ppm, que se cargará en la aplicación desde el archivo **foto.ppm**.

La aplicación lee desde el fichero **triangles.txt** la definición de los triangulos que se definan en el fichero: cada línea del fichero que comience con el caracter t define un triángulo mediante 15 números, cinco para cadavértice del triángulo como se muestra a continuación:

t x y z u v x y z u v x y z u v

La aplicación dibuja en cada momento un triángulo con la textura mapeada. El triángulo a dibujarse puede cambiar mediante la tecla <**ENTER**>.

La finalización de la ejecución se realiza pulsando la tecla <**ESC**>.

2 Desarrollo

Para el desarrollo de la práctica se han generado los siguientes ficheros .c y .h:

1. **cargar-triangulos.h** - define de forma global las estructuras utilizadas en los ficheros .c.

```
typedef struct punto
{
    float x, y, z, u,v;
} punto;

typedef struct hiruki
{
    punto p1,p2,p3;
} hiruki;
```

2. **cargar-triangulos.c** - este código es el encargado de leer la información de los triangulos a utilizar y guardarla en las estructuras definidas. Un puntero apuntara al triangulo actual y el número de triangulos se guarda en una variable.

3. **cargar-ppm.c** - en él se ubica el código que lee la imagen foto.ppm y guarda la informacion de todos los pixeles en un buffer, junto con las dimensiones horizontal y vertical. La información de un pixel se reparte en tres bytes que contienen la información del color r, g y b.

4. **dibujar-puntos.c** - aquí encontramos el código realmente desarrollado en la práctica. En él se definen las variables globales como se muestra a continuación:

```
unsigned char *bufferra;    // puntero a los pixels rgb de foto.ppm
int dimx, dimy;             // dimensiones x e y de foto.ppm
int num_triangles, indice=0;// numero de triangulos e indice del actual
hiruki *triangulosptr;      // puntero a los triangulos definidos por los puntos
```

   Ademas, explicaremos brevemente el funcionamiento de cada una de las funciones implementadas en el:

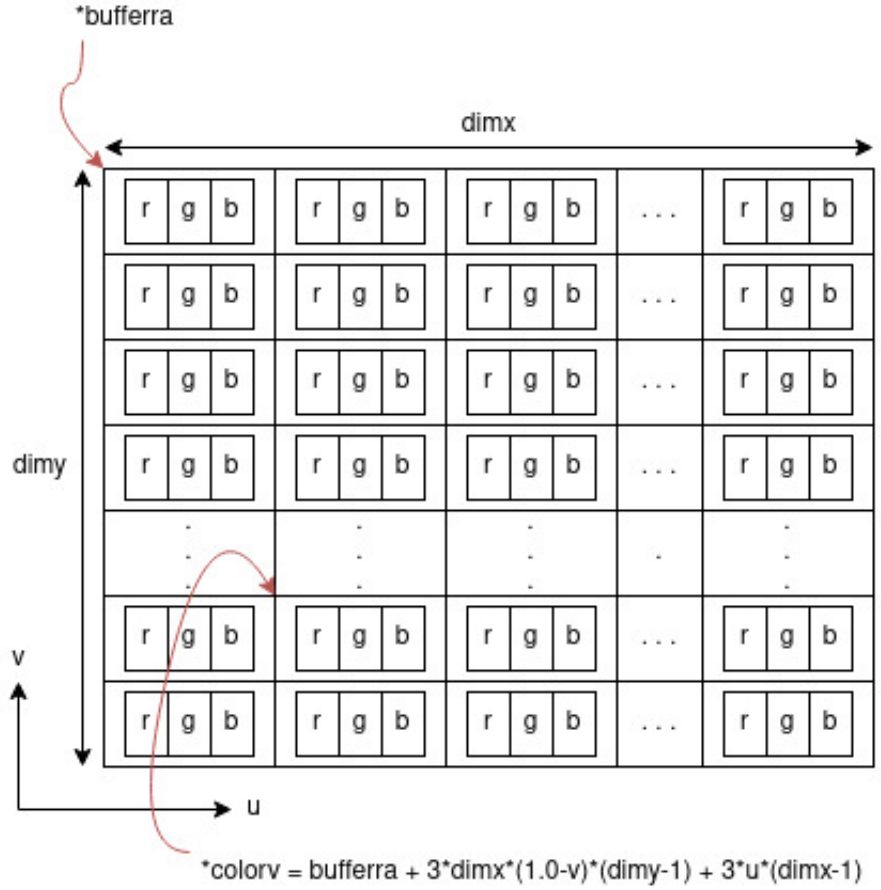
- **void comprobar_triangulos_modulo_signo (void);**
  Comprueba si las coordenadas de todos los puntos de todos los triángulos cumplen que $x, y \in [0, 500), z = 0$ y $u, v \in [0, 1)$ antes de empezar a dibujar cualquiera de los tringulos. En caso de que alguna coordenada no cumpla los limites, hace el modulo correspondiente y ajusta el signo para que sea positivo, de forma que la foto.ppm se veria como un toroide 3D. Un ejemplo de comprobacion y reajuste puede ser el siguiente caso:

  A(x=0.00, y=0.00, z=0.00, u=-0.50, v=0.00)
  A(x=0.00, y=0.00, z=0.00, u=0.50, v=0.00)
  B(x=-600.00, y=300.00, z=0.00, u=1.50, v=1.90)
  B(x=400.00, y=300.00, z=0.00, u=0.50, v=0.90)
  C(x=111.00, y=222.00, z=700.00, u=10.00, v=-13.70)
  C(x=111.00, y=222.00, z=0.00, u=0.00, v=0.30)

- **unsigned char *color_textura (float u, float v);**
  Dadas las coordenadas $P(u, v)$, devuelve un puntero hacia el pixel correspondiente a ese punto de la textura mediante el siguiente calculo con las dimensiones horizontal y vertical:

  $$bufferra + 3 * dimx * (1.0 - v) * (dimy - 1) + 3 * u * (dimx - 1)$$

  Nótese, que por cómo esta ordenado el buffer, debemos tratar los elementos de forma como si accediesemos a la posicion de una matriz empezando por la esquina superior izquierda (tratar los elementos del buffer como una matriz de dimx * dimy, y para el cálculo de la posición correcta debemos hacer $(1.0 - v)$, ya que el punto $(0, 0)$ de las coordenadas es desde el punto inferior izquierdo.



- **void dibujar_pixel (int i, int j, int r, int g, int b);**
  Dadas las coordenadas x e y de un punto y los valores r, g y b del color, dibuja el pixel correspondiente a $(x, y, 0)$ con ese color.
- **void dibujar_seg (punto inters1, punto inters2);**
  Dados dos puntos de intersección de un segmento (linea horizontal a altura $y = h$, con puntos de corte con dos de los lados del triángulo), calcula Δu y Δv en proporcion con Δx mediante regla de tres, y con su ayuda, para cada valor de x del segmento dibuja el color correspondiente de la textura buscado con los valores incrementales de u y v en el buffer.
  Nótese que para coger la información del color correspondiente apuntado por *colorv, hay que acceder a tres campos diferentes (colorv[0], colorv[1] y colorv[2]) que son los valores de r, g y b.
- **void establecer_orden (punto **sptrptr, punto **iptrptr, punto *mptrptr);**
  Funcion que establece el orden de los tres puntos de un triángulo en el eje vertical como superior, inferior y mediano, haciendo que cada puntero apunte al vértice correspondiente en orden.
- **void calcular_interseccion (punto *sptr, punto *iptr, int y, punto inters);**
  Calcula el punto de interseccion entre un eje $y = h$ y uno de los lados del triangulo definido por dos de los vértices con los que hace corte mediante regla de tres (proporcionalidad). Guarda los valores de x, y, z, u y v.
- **void dibujar_triangulo (void);**
  Empezando por el vértice superior y hasta el vértice inferior, dibuja el triangulo de salida con los pixeles mapeados de la textura. Se tienen en cuenta todos los casos particulares en los que se puedan definir los triangulos en el plano. Se establece el orden de los vertices, se calculan los puntos de intersección para cada valor de y en los que el triangulo esta definido y se dubuja el segmento, repitiendo el procedimiento en loops.
- **static void marraztu (void);**
  Contiene configuraciones de OpenGL y la llamada a dibujar_triangulo().
- **static void teklatua (unsigned char key, int x, int y);**
  Cuando se pulsa una tecla y según su código, si es <ENTER> se incrementa el indice del triángulo a dibujar (si llega al limite se reinicia a 0); si es <ESC> se finaliza la ejecucion; e.o.c imprime el valor de la tecla pulsada.
- **int main (int argc, char **argv);**
  Programa principal con configuraciones de OpenGL, llamadas a las funciones para cargar los triangulos y hacer la comprobacion de coordenadas, la foto ppm y las llamadas en bucle a marraztu y teklatua.

5. **compilar.sh** - fichero adicional creado para la compilacion de forma comoda. Contiene el siguiente comando:

   ```
   $ gcc cargar-ppm.c cargar-triangulos.c dibujar-puntos.c cargar-triangulos.h
       -lGL -lglut -lm -o ejecutable
   ```

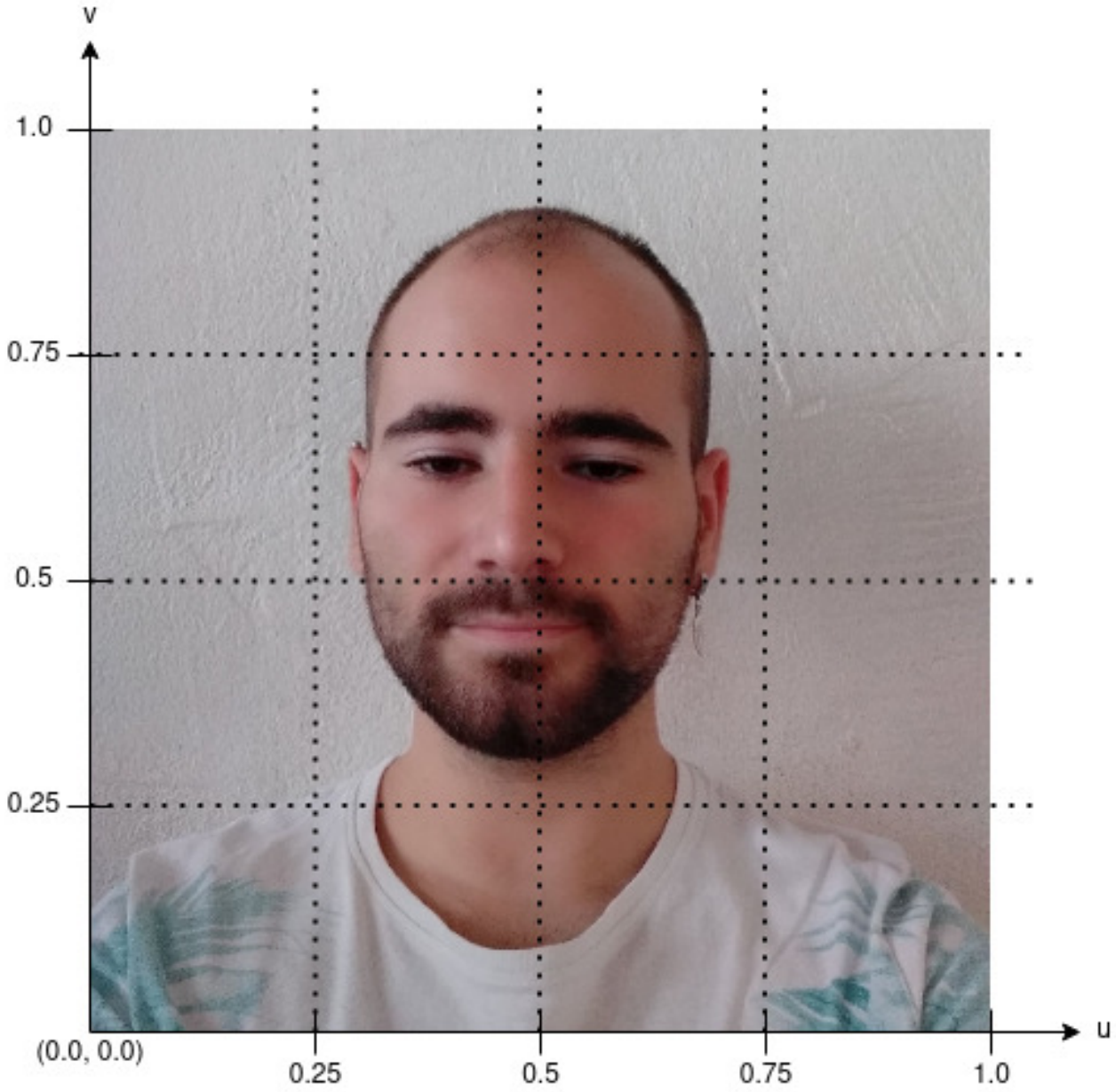
3 Manual de uso

- <**ENTER**>: cambiar al siguiente triángulo.
- <**ESC**>: finalizar ejecución.

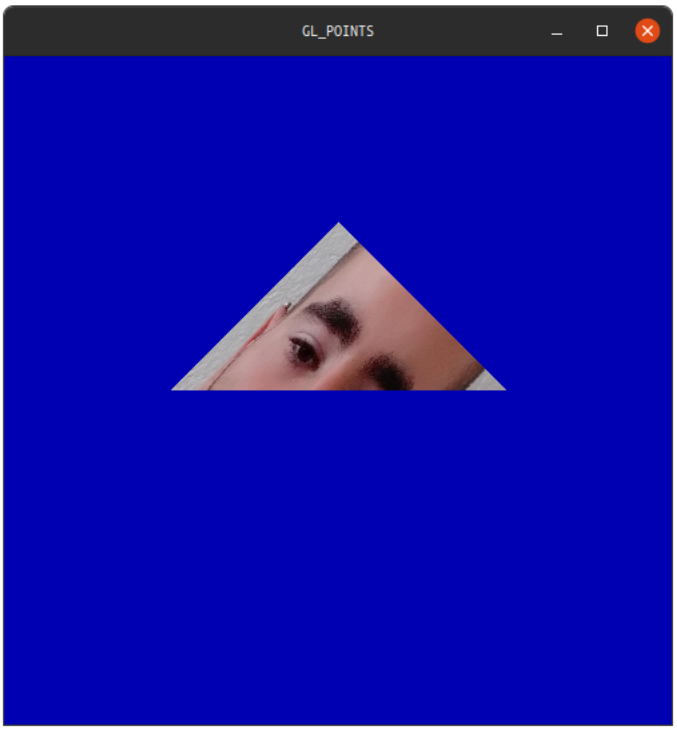
4 Casos de prueba

Para comprobar que la aplicacion funciona de forma correcta ante todos los casos posibles (o la mayoria de ellos) se han definido diferentes triángulos para probar con la ejecución. Con ello se comprueba el funcionamiento de cada una de las funciones con sus casos particulares como triangulos definidos por un tres puntos coincidentes o una linea, lados horizontales y/o verticales, puntos en los bordes del plano, evitar las divisiones entre 0, etc... A continuación algunos casos de prueba destacables.

Para dichas pruebas se ha utilizado la siguiente imagen:

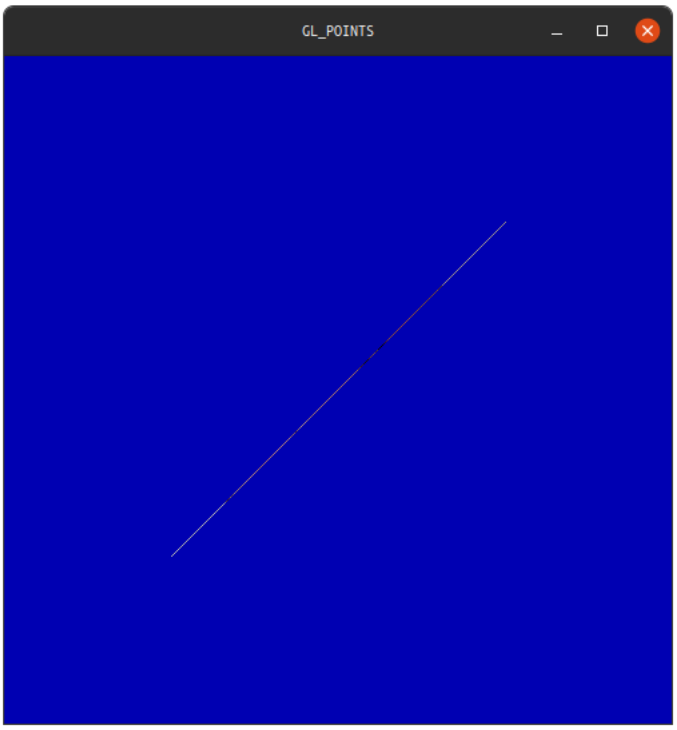


- t $A(250, 375, 0, 0.25, 0.75); B(125, 250, 0, 0.25, 0.5); C(375, 250, 0, 0.75, 0.75)$



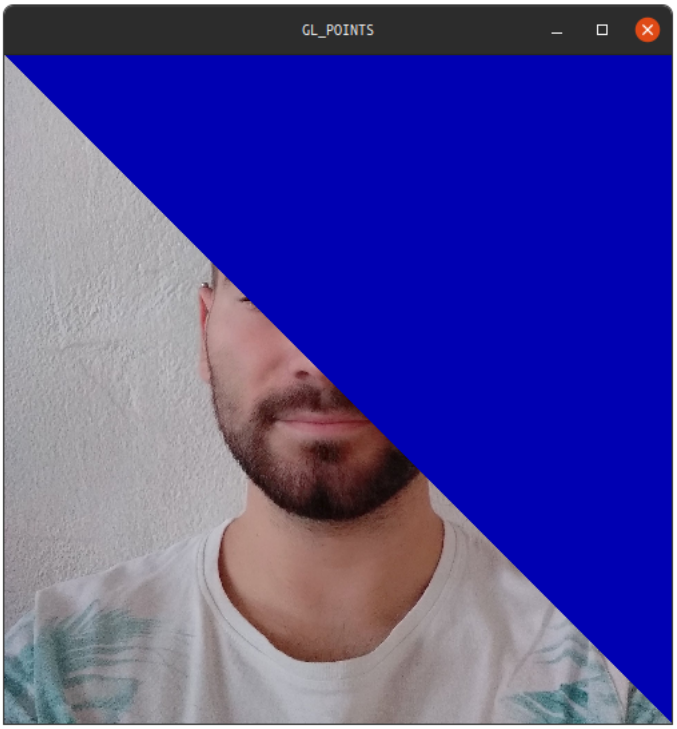
Triangulo estrechado y girado.

- t $A(125, 125, 0, 0.25, 0.25); B(250, 250, 0, 0.5, 0.75); C(375, 375, 0, 0.75, 0.25)$



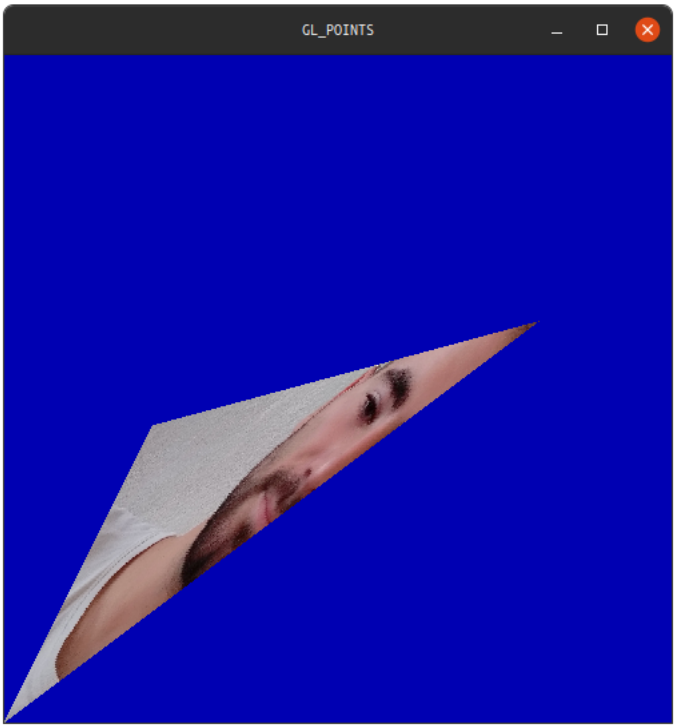
Triángulo definido por tres puntos en la misma recta.

- t $A(0, 0, 0, 0.0, 0.0); B(499, 0, 0, 0.99, 0); C(0, 499, 0, 0, 0.99)$



Triangulo con vertices en los bordes del plano y con arista horizontal y vertical.

- t $A(0, 0, 0, -0.5, 0.0); B(-600, 300, 0, 1.5, 1.9); C(111, 222, 700, 10.0, -13.7)$



Triángulo definido fuera del plano visible. Comprobacion de modulos y signos.