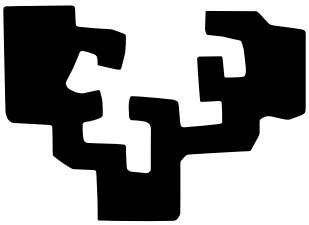
eman ta zabal zazu



Universidad del País Vasco

Euskal Herriko Unibertsitatea

Gráficos por Computador : Rasterizar Triángulo

Finalizado el Sábado, Septiembre 24, 2016

Sergio Tobal y Cristina Mayor

Contents

1.- Objetivos de esta práctica

El objetivo de esta práctica es introducirnos a OpenGL desde lo más básico (la creación del trazado externo de un triángulo pixel a pixel) hasta el rellenado del triángulo para entender como funciona y de este modo empezar a utilizar en un futuro los métodos que OpenGL incluye.

2.- Métodos de resolución

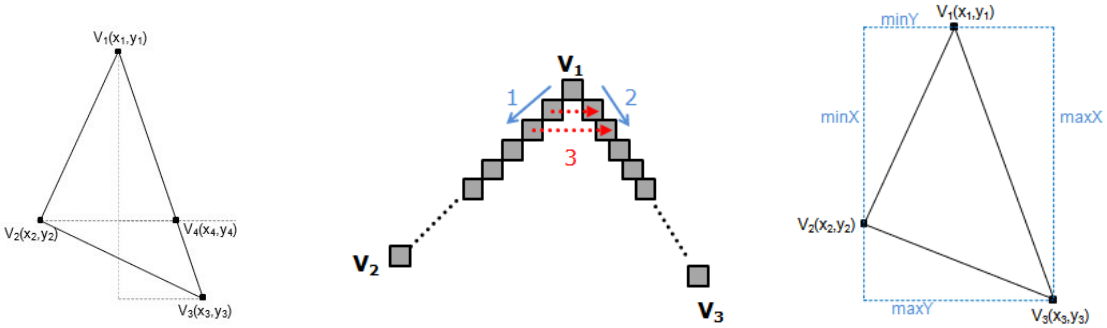
Haciendo investigación sobre el tema hemos descubierto 3 formas rápidas de rasterizar un triangulo, y 1 que esta pensando para cualquier tipo de polígono.

Para rasterizar o dibujar un triangulo tenemos:

1. **Algoritmo estándar**. En él dividimos un triangulo de 3 vértices en 2 triángulos mas reducidos, consiguiendo un triángulo con un vértice superior y el otro lado plano para realizar el rasterizado de manera mecánica.

2. **Algoritmo de Bresenham**. Este algoritmo explicado en clase esta pensado para dibujar una linea entre 2 puntos, pero podemos usarlo 2 veces desde un vértice hasta los otros 2, y mientras se dibujan las lineas entre los vértices, hacemos otra linea entre los puntos recién dibujados.

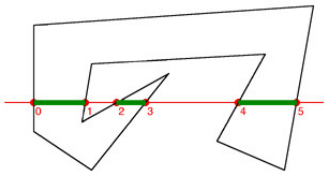
3. **Algoritmo baricéntrico**. La ventaja de este algoritmo es que no necesita ordenar los vértices, lo cual si necesitan los dos algoritmos anteriores; esto simplifica el escribir el algoritmo, pero a costa de un mayor trabajo del ordenador, que deberá calcular que puntos están dentro o fuera del triangulo.



(a) Estándar (b) Bresenham (c) Baricentrico

Los 3 algoritmos de rasterizacion expresados de manera visual.

Por otro lado tenemos el algoritmo de scanline rendering, que va dibujando lineas horizontales y mirando que puntos están dentro de un polígono. Para esto se debe ordenar los vértices por su posición en Y, ir moviendo la linea de escaneo desde el punto mas bajo al punto mas alto, y para cada linea sacar el punto de intersección del polígono con la linea para así dibujar desde el punto de entrada al punto de salida.



Para revisar los algoritmos se profundidad se puede visitar [triang-algs]

3.- Solución elegida

Para resolver el problema planteado, hemos usado el algoritmo estándar pero con variaciones.

Se ha creado una estructura llamada Vertice en la que metemos la posición (coordenadas X e Y) del punto tanto como la información correspondiente al mapeo del triángulo a dibujar (coordenadas U y V).

Para tener un mejor acceso a los puntos que se nos dan inicialmente, hemos creado 3 punteros que nos ayudan a gestionar mejor los vértices que se nos indican al principio. También tenemos un puntero que señala uno de los tres vértices(selectptoptr), gracias a él, el usuario podrá mover en cuatro direcciones (arriba, abajo, derecha e izquierda) el punto que se seleccione.

El método **main** es el punto de partida de todo programa openGL, donde se puede definir la posición, el tamaño o las características de la ventana, en nuestro caso hemos definido un buffer doble para poder intercambiar entre esos 2 buffers, evitando así los posibles errores de dibujo, y habilitado el multisampling o MSAA.

Para poder crear el triángulo, desde este método se llama al método **pedirCoordsUsuario**. Mientras no se escriban los 3 puntos correctamente (validando los parámetros del enunciado) estaremos en un bucle en el que nuestro programa va a estar continuamente pidiéndole al usuario que introduzca los números correspondientes a los campos de los tres vértices.

Este método también ordena los punteros de menor a mayor según la coordenada Y de cada vértice. El más alto irá en el puntero aptr y el más bajo el bptr. Guardaremos en el puntero selectptoptr por defecto el punto más bajo de todos.

Y por ultimo asignamos el nombre de los métodos que se encargan de las distintas funciones de openGL, como son el dibujado o el control del teclado.

El método **pedirCoordsUsuario** es el método implementado para que el usuario escriba por teclado los parámetros correspondientes a los tres vértices que forman el triángulo. Este método también se va a encargar de gestionar la validación de los parámetros que se introducen en cada vértice. Para validar los puntos se llamará al método **validarCoordenadas**.

Método **validarCoordenadas**. Este método se encarga de preguntar al método **puntoCorrecto** si cada punto es correcto, es decir, cumple las precondiciones iniciales.

Método **puntoCorrecto**. Se le pasa por parámetro el punto que queremos comprobar. Este método se encarga de comprobar que las coordenadas X e Y del punto sean valores entre el 0 y el 500 (ambos incluidos) y que el valor de los parámetros U y V oscilen entre el valor 0 y 1.

Método **draw**. Este es el método que gestiona el dibujar el triángulo.

Al iniciar el método dibujamos los puntos en 3 diferentes colores(RGB), en azul se dibujará el punto más alto, en verde se dibujará el punto del medio y en azul se dibujará el punto más bajo.

Tras dibujar los vértices, pasamos a definir los vértices c1 y c2 (con estructura Vertex). Dado que dibujamos de abajo hacia arriba, tanto c1 como c2 va a contener las coordenadas del vértice más bajo dado que estos son los puntos los vamos a utilizar como referencia para saber cuál sería la coordenada de entrada para empezar a dibujar el triángulo (c1) y la coordenada de salida para saber cuando dejar de dibujar el triángulo (c2).

A continuación procedemos a hacer la asignación de las variables slope. En el método hemos creado diferentes 'slope'. Estas pendientes son o o bien las pendientes que hay entre dos puntos:

1. **slope1**: pendiente que se produce entre el punto más bajo (bptr) y el punto más alto(mptr)
2. **slope2**: Esta pendiente va a tener dos valores. El triángulo se empieza a dibujar desde y=0, por tanto el primer valor que va a coger slope2 es la pendiente que hay entre el punto más bajo(bptr) y el punto del medio(mptr). Una vez se llega a la altura del punto del medio (y=bptr -> y) slope2 va a coger el valor de la pendiente que une el punto más alto(aptr) y el punto medio(mptr).

Por otro lado también tenemos las 'slope' correspondientes a las coordenadas de mapeo del triángulo:

1. **slopeU1**: Mide el cambio unitario de U sobre el cambio en la coordenada Y entre el punto más bajo y el punto más alto del triángulo.
2. **slopeU2**: Como pasaba con slope2, esta pendiente también tiene va a tomar dos valores a lo largo del método. Inicialmente slopeU2 va a medir el cambio unitario de U sobre el cambio en la coordenada X entre los puntos bajo y medio. Una vez se supere la coordenada Y del punto medio, slopeU2 tomará el valor unitario de U con respecto al cambio producido en la coordenada X entre el punto medio y el más alto.
3. **slopeV1**: Mide el cambio unitario de V sobre el cambio en la coordenada Y entre el punto más bajo y el punto más alto del triángulo.
4. **slopeV2**: Como pasaba con slope2 y slopeU2, esta pendiente también tiene va a tomar dos valores a lo largo del método. Inicialmente slopeV2 va a medir el cambio unitario de V sobre el cambio en la coordenada X entre los puntos bajo y medio. Una vez se supere la coordenada Y del punto medio, slopeV2 tomará el valor unitario de U con respecto al cambio producido en la coordenada X entre el punto medio y el más alto.

Una vez se declaran las variables, comprobamos si los puntos bptr y mptr están a la misma altura (sus coordenadas Y son iguales). En el caso de que sean iguales, cambiaremos el valor de c2 asignándole el valor del punto del medio.

Una vez finalizada esta asignación, procedemos a comprobar cual es el punto de entrada y el punto de salida. Dependiendo de si la pendiente slope1 o slope2 es más pronunciada, podremos saber si el punto del medio está a la derecha o a la izquierda de la recta pendiente entre el punto más bajo y el punto más alto, de este modo, si está a la derecha de la pendiente slope1, el puntero iptr (que marca el punto izquierdo, entrada al triángulo) tendrá asignado el punto c1 y el puntero dptr(que marca el punto derecho, salida al triángulo) tendrá asignado el punto c2. En caso contrario, las asignaciones serán al revés también.

Una vez hechas las comprobaciones, pasamos a iniciar el primer bucle for, que nos va a dibujar y texturizar el triángulo desde el vértice inferior hasta la altura del vértice del medio.

Este for va a actualizar los parámetros que tiene el vértice de entrada(c1) y el vértice de salida(c2) actualizándolo según las pendientes. Por cada punto de las líneas va a llamar al método dibujarLineas.

Tal como hemos ido comentando, a partir de que se supera la coordenada Y del punto medio(mptr->y) se cambian las pendientes que marcan el borde del triángulo. De este modo, al superar ese punto volveremos a entrar en otro for. Este bucle tiene la misma funcionalidad que el anterior pero va del punto medio al punto superior.

3.- Solución elegida continua en la siguiente pagina...

Método **dibujarLineas**. En este método se va a mirar el cambio unitario que hay en cada fila de las variables X e Y sobre el cambio que se hace en el eje de las X's. Ese dato lo vamos a utilizar para determinar el color que va a tener cada pixel de la línea. Vamos a proceder a ir dibujando pixel a pixel la línea(desde la coordenada X del punto de entrada hasta la coordenada X del punto de salida) y por cada pixel vamos a llamar al método **color** para calcular el color correspondiente.

Método **color**. Recibe por parámetro el valor U y V correspondiente al pixel del que queremos saber el color. Este método se va a encargar de devolver el color correspondiente que definen los parámetros.

Una vez que se seleccione un vértice del triángulo, podremos moverlos mediante el teclado. En el punto 5 del documento podremos ver las opciones que el usuario tiene sobre el movimiento de los puntos del triángulo y como puede interactuar con el.

4.- Código en C

Código efectuado en la práctica:

Código de la practica

```
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct {
    float x,y,u,v;
} Vertex;

Vertex v1, v2, v3;
Vertex *aptr, *bptr, *mptr; // punteros al alto, bajo y medio
Vertex *iptr, /*!< Puntero al punto de entrada de la linea a dibujar */
*dptr; /*!< Puntero al punto de salida de la linea a dibujar */
Vertex *selectptoptr; /*!< Puntero al vertice seleccionado por el usuario */

int msaa=0;
/**
 * Ordena los vertices usando 3 puntos, al final quedara en v1.y el mas bajo
 * y en v3.y el mas alto
 */
static void sortByY() {
    Vertex *auxptr;

    aptr = &v3;
    mptr = &v2;
    bptr = &v1;

    if (mptr->y > aptr->y) {
        auxptr = mptr;
        mptr = aptr;
        aptr = auxptr;
    }
    // aptr y mptr estan ordenados correctamente
```

```
    if (bptr->y > mptr->y) {
        auxptr = mptr;
        mptr = bptr;
        bptr = auxptr;
    }
    // el bptr es el más bajo fijo!
    if (mptr->y > aptr->y) {
        auxptr = aptr;
        aptr = mptr;
        mptr = auxptr;
    }
}
/**
 * Devuelve el color que le corresponde al pixel marcado por el pto (u,v)
 * @param u Coordenada de mapeado del eje horizontal
 * @param v Coordenada de mapeado del eje vertical
 * @return Numero correspondiente al color
 */
float color(float u, float v) {
    int x,y;
    x = u/0.125;
    y = v/0.125;

    if( (x % 2) == 0 ) {
        if ( ( y % 2 ) == 0 ) {
            return (0.0);
        } else {
            return (1.0);
        }
    }else {
        if( (y % 2) == 0 ) {
            return (1.0);
        } else {
            return (0.0);
        }
    }
}
/**
 * Dibuja desde un pto de entrada del triangulo a uno de salida
 * @param line Altura a la que se va a dibujar la linea
 */
void dibujarLineas(float line){
    float slopeUX, slopeVX, u, v, val;

    slopeUX = ((dptr->u-iptr->u)/(dptr->x-iptr->x));
    slopeVX = ((dptr->v-iptr->v)/(dptr->x-iptr->x));
    u = iptr->u;
    v = iptr->v;
    for (float x = iptr->x; x < dptr->x; x++) {
        u += slopeUX;
        v += slopeVX;

        val = color(u, v);
```

```
        glColor3f(val, val,val );
        glBegin(GL_POINTS);
        glVertex2f(x, line);
        glEnd();
    }
}
/**
 * Dibujado de los 3 vertices del triangulo
 */
static void dibujarVertices(){
    glPointSize(10.0f);
    glBegin(GL_POINTS);
    glColor3f(1.0f,0.0f,0.0f);
    glVertex2f(bptr->x,bptr->y);
    glColor3f(0.0f,1.0f,0.0f);
    glVertex2f(mptr->x,mptr->y);
    glColor3f(0.0f,0.0f,1.0f);
    glVertex2f(aptr->x,aptr->y);
    glEnd();
}

void reshape(int width, int height) {
    // we ignore the params and do:
    glutReshapeWindow(500, 500);
}

static void draw(void) {
    float slope1, slope2, slopeU1, slopeU2, slopeV1, slopeV2;
    float line;
    Vertex c1,c2;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 500.0, 0.0, 500.0,-250.0, 250.0);
    if (msaa) glEnable(GL_MULTISAMPLE_ARB);
    else glDisable(GL_MULTISAMPLE_ARB);
    dibujarVertices();

    glPointSize(1.0f);
    glColor3f(1.0f,1.0f,1.0f);

    c1.x = bptr->x; c1.u = bptr->u; c1.v = bptr->v;
    c2.x = bptr->x; c2.u = bptr->u; c2.v = bptr->v;

    slope1 = ((aptr->x-bptr->x)/(aptr->y-bptr->y));
    slope2 = ((mptr->x-bptr->x)/(mptr->y-bptr->y));

    slopeU1 = ((aptr->u-bptr->u)/(aptr->y-bptr->y));
    slopeU2 = ((mptr->u-bptr->u)/(mptr->y-bptr->y));

    slopeV1 = ((aptr->v-bptr->v)/(aptr->y-bptr->y));
    slopeV2 = ((mptr->v-bptr->v)/(mptr->y-bptr->y));
```

```
    if (bptr->y == mptr->y) {
        c1.x = bptr->x; c1.u = bptr->u; c1.v = bptr->v;
        c2.x = mptr->x; c2.u = mptr->u; c2.v = mptr->v;
    }

    // El pto de entrada(c1) sera donde la pendiente este mas baja
    if (slope1 < slope2) {
        iptr = &c1;
        dptr = &c2;
    } else {
        iptr = &c2;
        dptr = &c1;
    }

    // Dibujamos la parte inferior del triangulo, hasta que cambia la pendiente
    for (line = bptr->y; line < mptr->y; line++, c1.x += slope1, c2.x += slope2,
         c1.u += slopeU1, c2.u += slopeU2, c1.v+=slopeV1, c2.v+=slopeV2) {
        dibujarLineas(line);
    }

    //Cambiar slope2 que era la pendiente mas pequeña a la sig. pendiente
    slope2 = ((aptr->x-mptr->x)/(aptr->y-mptr->y));
    slopeU2 = ((aptr->u-mptr->u)/(aptr->y-mptr->y));
    slopeV2 = ((aptr->v-mptr->v)/(aptr->y-mptr->y));

    // Draw the upper side of the triangle
    for (line = mptr->y; line < aptr->y; line++, c1.x += slope1, c2.x += slope2,
         c1.u += slopeU1, c2.u += slopeU2, c1.v+=slopeV1, c2.v+=slopeV2) {
        dibujarLineas(line);
    }

    glutSwapBuffers();
}

static void keyboard (unsigned char key, int x, int y) {

    switch(key)     {
        case 98: // b
            selectptoptr = bptr;
            break;
        case 110: // n
            selectptoptr = mptr;
            break;
        case 109: // m
            selectptoptr = aptr;
            break;
        case 108: // l
            msaa = !msaa;
            if(msaa == 1)printf("msaa activado\n");
            else printf("msaa desactivado\n");
            glutPostRedisplay();
            break;
```

```
        case 27:  // <ESC>
            exit(0);
            break;
        default:
            printf("%d %c\n", key, key );
    }
    sortByY();
    glutPostRedisplay();
}

void SpecialKeys(int key, int x, int y) {
    switch (key) {
        case GLUT_KEY_LEFT:
            selectptoptr->x -= 20.0f;
            break;
        case GLUT_KEY_RIGHT:
            selectptoptr->x += 20.0f;
            break;
        case GLUT_KEY_UP:
            selectptoptr->y += 20.0f;
            break;
        case GLUT_KEY_DOWN:
            selectptoptr->y -= 20.0f;
            break;
        default:
            printf("El codigo ASCII de la tecla pulsada es %d\n", key );
    }
    sortByY();
    glutPostRedisplay();
}

bool puntoCoorrecto(Vertex v){

    if(v.x<0 || v.x>500 || v.y<0 || v.y>500 ||
          v.u>1 || v.u<0 || v.v<0 || v.v>1) return false;
    else return true;
}

bool validarCoordenadas(){
    bool pt1 = puntoCoorrecto(v1);    bool pt2 = puntoCoorrecto(v2);
    bool pt3 = puntoCoorrecto(v3);
    if(pt1 && pt2 && pt3) return true;
    else{
        printf("Hay algún valor erróneo. Mete los datos de nuevo\n");
        return false;
    }
}

bool pedirCoordsUsuario(){
    printf("Mete v1.x: "); scanf("%f", &v1.x);
    printf("Mete v1.y: "); scanf("%f", &v1.y);
    printf("Mete v1.u: "); scanf("%f", &v1.u);
    printf("Mete v1.v: "); scanf("%f", &v1.v);
```

```
    printf("Mete v2.x: "); scanf("%f", &v2.x);
    printf("Mete v2.y: "); scanf("%f", &v2.y);
    printf("Mete v2.u: "); scanf("%f", &v2.u);
    printf("Mete v2.v: "); scanf("%f", &v2.v);

    printf("Mete v3.x: "); scanf("%f", &v3.x);
    printf("Mete v3.y: "); scanf("%f", &v3.y);
    printf("Mete v3.u: "); scanf("%f", &v3.u);
    printf("Mete v3.v: "); scanf("%f", &v3.v);

    return validarCoordenadas();
}

int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_MULTISAMPLE);
    glutInitWindowPosition(50, 25);
    glutInitWindowSize(500,500);
    glutCreateWindow("OpenGL");

    bool coord = pedirCoordsUsuario();
    while(coord == false) coord = pedirCoordsUsuario();

    sortByY();
    selectptoptr = bptr;

    glutDisplayFunc(draw);
    glutKeyboardFunc(keyboard);
    glutSpecialFunc(SpecialKeys);
    glutReshapeFunc(reshape);

    glClearColor(0.0f, 0.5f, 0.5f, 1.0f);
    glutMainLoop();
    return 0;
}
```

5.- Ampliación a lo pedido

Como ampliación a lo que se nos pedía inicialmente hemos añadido las siguientes opciones en nuestro proyecto:

- **SpecialKeys**. A raíz de ver en clase mover un vértice con teclas hemos investigado como mover los 3 vértices con las teclas de dirección, así que ahora tenemos b, n y m para cambiar entre el vértice mas bajo(punto rojo), el medio(punto verde) y el mas alto(punto azul) respectivamente. Lo que nos costo un poco fue averiguar que las teclas de dirección se consideran teclas especiales y que necesitan de su propio método. Después de esto fue usar casi lo mismo que en el método de Keyboard.
- **Anti-Aliasing**. Al ver que con algunos ángulos se formaba demasiado aliasing, hemos investigado el como aplicar un método sencillo de anti-aliasing (MSAA). Debido a esto también hemos cambiado el [buffering].Con el buffer único, mandábamos la imagen generada por OpenGL a la tarjeta gráfica y al monitor tan pronto como se llamase al método *glFlush()*, pero, debido al tiempo de refresco de

la pantalla, podría haber frames que no se dibujaran, así que usamos un buffer doble y sustituimos *glFlush()* por *glutSwapBuffers()* para tener como dos lienzos en los que dibujar, en el que puedes ir dibujando y mostrando, de esta manera cuando uno se muestre, otro se dibuja, y así no se nota la desincronizacion.

- **Impedir el resize**. Hemos modificado el método de reshape para que la ventana siempre tenga un valor de 500x500, impidiendo así que el usuario cambie el tamaño a valores que no tenemos pensados.

6.- Conclusiones

Esta primera practica nos ha servido para revisar la importancia de los punteros en C, aprender el uso de Latex, y recordar algunas cosas olvidadas.

Hemos aprendido que un buen pensamiento, un buen planteamiento y la ordenación eficiente del código es una base importante para una implementación satisfactoria.

En un principio no teníamos las variables agrupadas, ni estaban declaradas ordenadamente, es por ello por lo que nos ha fallado el código en partes que, en el caso de haber sido ordenados inicialmente, no hubiera fallado.

Por otro lado hemos aprendido que la utilización de punteros (cosa que no habíamos visto hasta ahora) es una manera eficaz de implementar un código más claro y evitando que haya fallos o haya que hacer comparaciones no necesarias.

En cuanto a Latex, hemos comenzado a utilizarlo, saber como se usa y hemos creado también una plantilla para poder usarla sea cual sea el objetivo de su utilización.

Referencias

[triang-algs] Algoritmos de rasterizacion de triangulos

[scan-alg] Algoritmo scanline para rellenado de poligonos

[buffering] Diferencias entre Buffer unico y doble

Anexos

Anexo I: Manual del Usuario

Al ejecutar el programa el usuario deberá ir introduciendo los coordenadas de cada vértice, así como los valores de u y v, si algún valor no fuera introducido se considerara 0.

Una vez el programa compruebe que los datos son válidos, es decir, los valores X e Y que se den sean valores pertenecientes al rango 0-500 y los valores U y V que se den sean pertenecientes al rango 0-1, el programa procederá a dibujar lo que el usuario ha introducido.

Cuando el triángulo está dibujado, el usuario tiene la opción de interactuar con el programa. Hemos definido que al pulsar las teclas b, n y m se podrá seleccionar un vértice a mover, seleccionando con b el vértice mas alto, con n el que esté a media altura y con m el que esté en la posición más baja.

Una vez indicado el vértice que queremos mover, utilizando las teclas de dirección del teclado procederemos a mover los puntos hacia arriba, abajo, derecha o izquierda.

También existe la función experimental del antialiasing usando el método MSAA, aunque por defecto esta desactivada. Si se quiere activar se puede usar la tecla l, aunque como se ha indicado puede dar errores, así que se recomienda usar con prudencia.