Primer Parcial

Primer Cuatrimestre 2024

Normas generales

- El parcial es INDIVIDUAL
- Una vez terminada la evaluación se deberá completar un formulario con el *hash* del *commit* del repositorio de entrega. El link al mismo es: `https://forms.gle/U4fTCfQcJvDvwpWH9`.
- Luego de la entrega habrá una instancia coloquial de defensa del trabajo

Régimen de Aprobación

- Para aprobar el examen es necesario obtener cómo mínimo **60 puntos**.
- Para conservar la posibilidad de promocionar es condición necesaria obtener como mínimo **80 puntos**.

Compilación y Testeo

El archivo `main.c` es para que ustedes realicen pruebas básicas de sus funciones. Sientanse a gusto de manejarlo como crean conveniente. Para compilar el código y poder correr las pruebas cortas implementadas en main deberá ejecutar `make main` y luego `./runMain.sh`.
En cambio, para compilar el código y correr las pruebas intensivas deberá ejecutar `./runTester.sh`. El programa puede correrse con `./runMain.sh` para verificar que no se pierde memoria ni se realizan accesos incorrectos a la misma.

Pruebas intensivas (Testing)

Entregamos también una serie de *tests* o pruebas intensivas para que pueda verificarse el buen funcionamiento del código de manera automática. Para correr el testing se debe ejecutar `./runTester.sh`, que compilará el *tester* y correrá todos los tests de la cátedra. Luego de cada test, el *script* comparará los archivos generados por su parcial con las soluciones correctas provistas por la cátedra. También será probada la correcta administración de la memoria dinámica.

Ej. 1 - (50 puntos)

La famosa consola de videojuegos *Orga2-station* se diseñó de forma que para dibujar los gráficos, recorra una lista enlazada llamada *Display List* que contiene información de cada elemento a renderizar. Cada nodo de dicha lista, llamado `nodo_display_list_t`, contiene las coordenadas `x,y,z` (siendo `z` la profundidad en la escena a renderizar) y un puntero a la función a utilizar para determinar el valor de `z`.

Para dibujar una escena correctamente es necesario dibujar los nodos que están más lejos de la pantalla, esto es, dibujar primero aquellos con un valor en `z` más alto.

Para evitar recorrer la lista entera varias veces, se utiliza una *Ordering Table* o *Tabla de Ordenamiento*. Esta estructura mantiene referencias ordenadas por su valor de `z` a los nodos a dibujar de la *Display List*.

En otras palabras, una *Ordering Table* (OT a partir de ahora) es una lista enlazada diseñada para agrupar primitivas de gráficos 3D según su ubicación en el eje Z. Primero se dibujan los más lejanos y luego los más cercanos (es decir, se dibujan ordenados según `z` de mayor a menor).

La OT usa un array fijo como ancla para cada posible valor en Z, evitando así tener que recorrer la lista enlazada una vez para cada valor de Z. El tamaño del array determinará cuántos puntos en Z puede haber. Por ejemplo, si el array tiene 10 posiciones, entonces habrá 10 "profundidades" a dibujar. En esta consola, el tamaño es variable según el juego, por lo que no sabemos de antemano cuánta memoria utilizará.

Cada primitiva calcula la posición `z` a partir de las coordenadas `x,y`, además el valor resultante de Z se redondea para utilizarse como índice del OT.

Si todo fue calculado correctamente, con llamar una sola vez a la función para dibujar en pantalla, el motor de dibujo de la *Orga2-station* renderizará sin errores.

Cada nodo de la lista enlazada *Display List* tiene la siguiente forma:

```
typedef struct {
    // Puntero a la función que calcula z (puede ser distinta para cada nodo):
    uint8_t (*primitiva)(uint8_t x, uint8_t y, uint8_t z_size);
    // Coordenadas del nodo en la escena:
    uint8_t x;
    uint8_t y;
    uint8_t z;
    //Puntero al nodo siguiente:
    nodo_display_list_t* siguiente;
} nodo_display_list_t;
```

Por otro lado, la OT está definida como:

```
typedef struct {
    uint8_t table_size;
    nodo_ot_t** table;
} ordering_table_t;
```

y los nodos de la OT:

```
typedef struct {
    nodo_display_list_t* display_element;
    nodo_ot_t* siguiente;
} nodo_ot_t;
```

Un esquema general de las estructuras mencionadas se muestra en la figura 1

Se pide:

Implementar en `asm`:

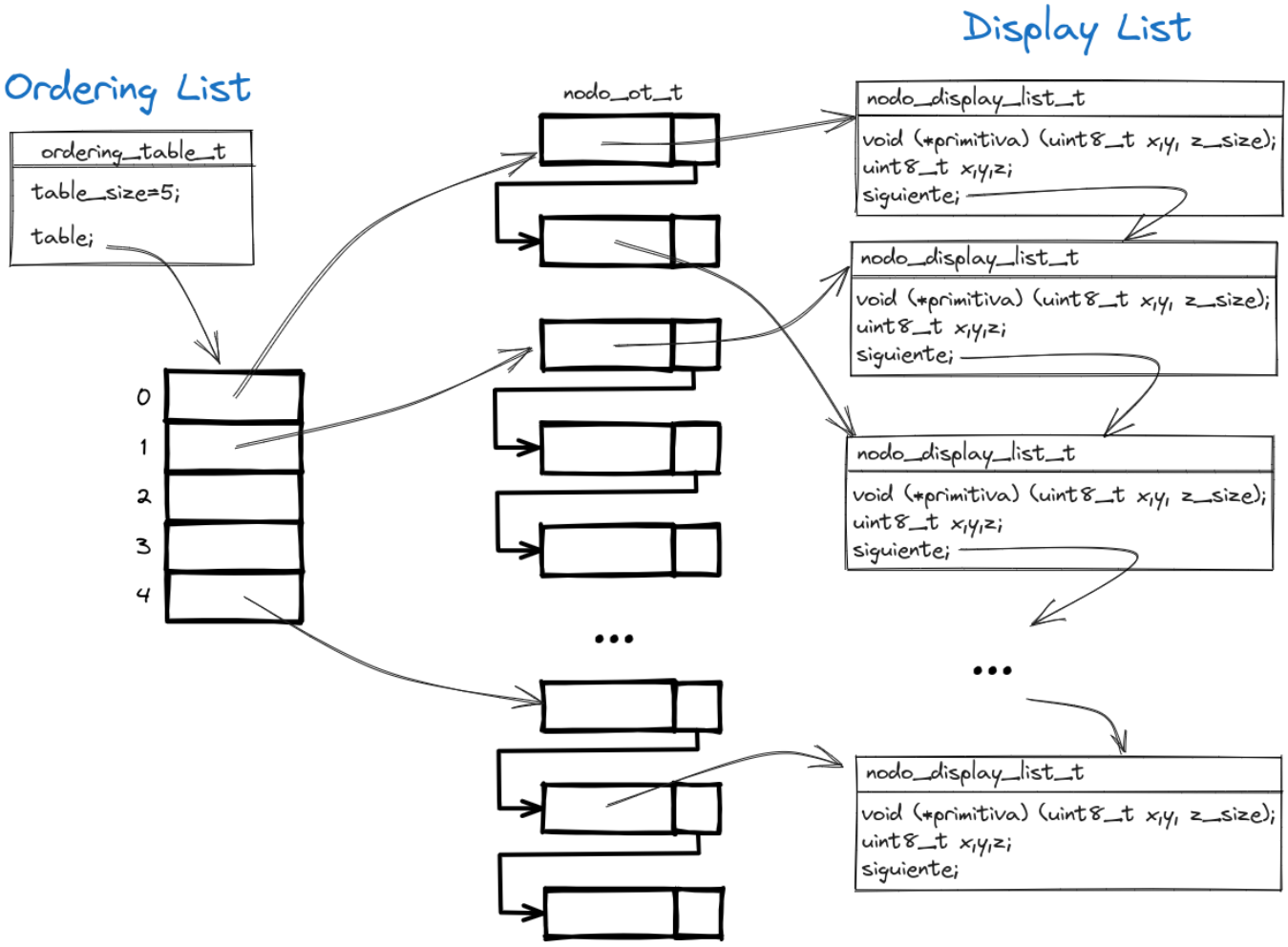

Figura 1: Esquema de estructuras para el motor de dibujo de la consola Orga2-station.

1. La función que inicializa la Ordering Table. La aridad de la función es:

   ```
   ordering_table_t* inicializar_OT_asm(uint8_t z_size);
   ```

2. Una función llamada `calcular_z` que complete la coordenada `z` de cada nodo en función de su primitiva.

   La aridad de la función es:

   ```
   void* calcular_z_asm(nodo_display_list_t* nodo,
                        uint8_t z_size);
   ```

   Aclaración: *Los parámetros son:*

   - *`nodo_display_list_t* nodo`; un puntero al nodo.*
   - *`uint8_t z_size`; tamaño del array de OT o "cantidad de profundidades" (utilizado para calcular `z`).*

   La función deberá llamar a la primitiva de ese nodo (`z=nodo.primitiva(x,y,z_size);`) y completar el campo `z` de dicho nodo.

   Aclaración 2: *Recordar que el `z` calculado va a ser menor al tamaño de la OT y (hint) puede ser utilizado como índice.*

3. La función que coloca en la OT las referencias de la Display List. La aridad de la función es:

   ```
   void* ordenar_display_list_asm(ordering_table_t* ot,
                                  nodo_display_list_t* display_list);
   ```

   La función deberá calcular el `z` para dicho nodo, ubicar en la OT la lista correspondiente y agregar el nodo al final de la misma.

Ej. 2 - (50 puntos)

Tenemos fotos sacadas de una cámara web, pero lamentablemente el formato de esta no es **RGBA** sino **YUYV**. Este formato es conocido como YUV `4:2:2`, dónde la información de intensidad de luz es individual para cada pixel, pero la información de color no lo es. Es decir, los datos se agrupan de la siguiente forma:

```
typedef struct{
    int8_t y_1;
    int8_t u;
    int8_t y_2;
    int8_t v;
} yuyv_t;
```

donde `y_1` es la "luz" del pixel 1, `y_2` es la luz del pixel 2 y `u,v` el "color" de ambos píxeles.

Se pide implementar en ASM usando SIMD una función que dada una imagen en este formato, la convierta a **RGBA**.

Como el bloque **YUYV** representa dos píxeles (uno por cada **Y**), les recomendamos que al procesar un bloque Y_1UY_2V lo separen en Y_1VU e Y_2VU siendo cada uno:

```
typedef struct{
    int8_t y;
    int8_t u;
    int8_t v
} yuv_t;
```

La función de conversión de un pixel **YUV** a **RGBA** es:

```
rgba_t YUV_to_RGBA(yuv_t in) {
    int Y = in.y;
    int U = in.u - 128;
    int V = in.v - 128;
    return (rgba_t) {
        .R = Y + 1.370705 * V,
        .G = Y - 0.698001 * V - 0.337633 * U,
        .B = Y + 1.732446 * U,
        .A = 255
    };
}
```

Adicionalmente que algunas imágenes se capturaron con errores y por lo tanto algunos valores de **U** y **V** son de 127 (`0x7F`) simultáneamente.

Se necesita que para detectar estos errores, se coloque un color fijo en el pixel de destino: `RGBA(127,255,0,255)`.

Notar que:

- Se recibe un puntero al espacio de memoria donde debe guardarse el resultado, es decir, esa memoria ya fue pedida (o reservada)

- La imagen de salida tiene el doble de tamaño (horizontalmente) a la de entrada