CLASE 13: más sobre memoria

Introducción a la Programación de Gráficos - Image Campus

Una de las actividades de la clase pasada consistió en una ejercitación sobre memory leaks.

El objetivo de esta actividad era profundizar en el entendimiento de la memoria. Entender aspectos de su funcionamiento es clave para poder trabajar con gráficos.

Para ilustrar esto, podemos recurrir a algunas de las instrucciones que vimos en el ejemplo de OpenGL...

Ya en el comienzo de nuestro programa teníamos esto:

```
GLFWwindow* window = glfwCreateWindow(640, 480, "OpenGL", NULL, NULL);
```

Fíjense como esta función devuelve un puntero a la ventana creada. Si no entendemos de qué se trata eso, ya nos encontraremos con problemas. Al interior de la función **`glfwCreateWindow`** se hace un new de una **`GLFWwindow`** y como retorno de dicha función se devuelve el puntero a esa ventana alojada en el heap.

De esta manera, podemos pasar dicho puntero a otra función que creemos o manipularlo sin estar creando copias de la ventana en la memoria, sino siempre teniendo un simple “señalador” a la misma.

Ahora bien, ¿en dónde liberamos esa memoria? ¿alguno agregó algo para hacerlo cuando hizo el programa de OpenGL? Si no hacemos nada más estaremos cerrando el programa sin liberar esa memoria. Peor aún, sin darnos cuenta podríamos estar haciendo operaciones como cerrar una ventana y abrir otra, y en el camino estar generando leaks. Por lo tanto, es una buena práctica (incluso si el programa va a terminar) hacer el proceso contraria que se hizo en la etapa de creación antes del ingreso al loop.

En la etapa de cierre (posterior al loop), en el caso de GLFW, deberíamos usar la función **`glfwDestroyWindow`** y pasarle por parámetro el puntero a la ventana. Dentro, la función se encarga de hacer el delete correspondiente.

Como agregado, luego de esto también deberíamos hacer un llamado a la función **`glfwTerminate`**, ya que así como se hizo **`glfwInit`** también será necesario liberar los recursos que esta inicialización estableció.

Fíjense, entonces, como en este simple ejemplo de inicialización de una ventana con GLFW ya tenemos un montón de implicancias sobre el funcionamiento de la memoria y su correcta administración para generar programas sin errores.

Para reforzar todavía más la importancia del ejercicio planteado y del entendimiento de la memoria en general, tomo otra pieza de código comentada la clase pasada:

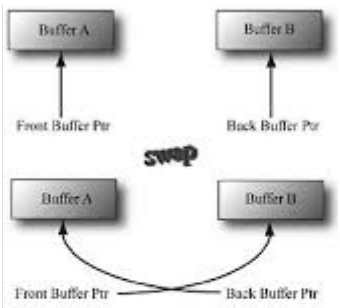
```
glfwSwapBuffers(window);
```

glfwSwapBuffers era una función muy importante que se llamaba dentro del loop de dibujado, hacia el final del mismo. Se necesitaba para mostrar algo en pantalla puesto que lo que se dibujaba con funciones como **glDrawArrays** se hacía en el back buffer y lo que el monitor mostraba era el front buffer. Por lo tanto, para actualizar el frame y "traer adelante" el dibujo, se debían intercambiar (swap) los buffers (las porciones de memoria que tenían el array de colores del frame actual y del próximo). Al hacer esto, entonces, se mostraba el dibujo ya que el back buffer pasaba a ser el front buffer y el ahora back buffer se disponía para ser dibujado.

Si este proceso queda claro, hay una pregunta interesante que podemos hacernos relacionada con la memoria: ¿este proceso es costoso? es decir, ¿lleva mucho tiempo y cuesta muchos ciclos de procesador que afectan la performance? ¿es por eso que debemos optimizar nuestros juegos, por culpa de este proceso de intercambio? ¿se debe copiar toda la data de la memoria al otro, necesitando un pedazo de memoria equivalente auxiliar, como cuando tenemos 2 variables (A y B) y hacemos que sus datos se intercambien? ¿qué creen?

Si pensamos en que los dos buffers son porciones de memoria y entendemos lo que son los punteros, podremos darnos cuenta de que este proceso es en realidad muy simple y no consume muchos recursos: el intercambio de buffers es simplemente un cambio de punteros.

Un puntero apunta al front, otro al back, y no es necesario copiar grandes cantidades de memoria a ningún lado. Con solo hacer que el puntero front ahora apunte al back y viceversa, se hizo el cambio en dos simples variables que sólo guardan una dirección de memoria (podrían ocupar 64 bits entre las 2, por ejemplo).



Pero vayamos todavía más allá, veamos las últimas dos piezas de código que me interesa destacar, ahora ya de la API de OpenGL y no de GLFW, para seguir dando cuenta de la importancia de entender sobre la memoria.

glGenBuffers es una función que nos crea un buffer (luego lo usábamos para pasarle los vértices a dibujar al GPU):

```
GLuint vbo;
glGenBuffers(1, &vbo);
```

Por otra parte, también teníamos esta función:

```
glVertexAttribPointer(posAttrib, 2, GL_FLOAT, GL_FALSE, 0, 0);
```

Para entender los parámetros de estas funciones y lo que hacen debemos comprender aún más aspectos de la memoria: su relación con los arrays y lo que se denomina aritmética de punteros.

Empecemos por entender algunos aspectos de los arrays en relación a la memoria que son importantes.

Arrays

Un array es un espacio contiguo de variables en la memoria.

En C++ podemos crear el array haciendo, por ejemplo:

```
int nums[] = { 1, 2, 3 };

int masNums[5];
```

Si hacemos esto dentro del main, el array se creará en el stack. El problema con esta forma de crearlo es que debemos establecer la cantidad de elementos de forma anticipada. Es decir, no podemos determinarlo en tiempo de ejecución. Esto es poco útil para arrays de los que no sabemos su tamaño de antemano. ¿Qué pasa si el usuario quiere elegir cierta cantidad de elementos, o si por ejemplo tenemos enemigos pero su cantidad se determina acorde a la dificultad o avance del jugador?

Para solucionar esto podemos crear un array determinando el tamaño dinámicamente si lo creamos en el heap. Podríamos hacerlo de esta manera:

```
int* nums = new int[]{ 1, 2, 3 };

int* masNums = new int[5];
```

Si hacemos esto, no necesitamos saber la cantidad de antemano, podemos dejar que el usuario lo ingrese durante la ejecución del programa:

```
int size;
cin >> size;
int* nums = new int[size];
```

Ahora bien, cuando hacemos esto, estamos pidiendo un espacio contiguo en la memoria (el heap específicamente) de determinada cantidad de elementos. Por lo tanto, así como pedimos, necesitamos liberar dicha memoria. Tratándose de arrays, la sintaxis del delete es algo diferente. No es como un delete de cualquier variable única, sino que se debe realizar con los corchetes entre el delete y el nombre de la variable:

```
delete[] nums;
```

Teniendo esto en cuenta, realizar el ejercicio tras el siguiente aviso...

Importante:

1) Revisar el documento de la clase anterior si quedó alguna duda de punteros (sin arrays) o memory leaks.

2) Si las dudas persisten (o surgieron nuevas), escribirlas en el documento presentado al comienzo de esta clase.

3) Este ejercicio requiere atención y en muchos casos pensar una respuesta sobre algo que no se comentó antes, sino en base a lo que se ve y a un análisis del código. El objetivo es que se tomen su tiempo y lean detenidamente intentando encontrar una explicación de por qué creen cada cosa.

https://forms.gle/CN9bYEUjYuAHY1y17