Capítulo 3. Desarrollo del Motor de Juegos

3.1 Análisis

Las características principales de un Motor de Juegos es que sea modular, re-usable y que aproveche los recursos de hardware, por lo que se va a crear una estructura en la cual los tres sub-Motores que van a constituir este Motor de Juegos son totalmente independientes. Estos tres sub-Motores van a tener la misma estructura y diseño para que el usuario final al saber o conocer como utilizar e implementar un sub-Motor, intuitivamente sepa como manejar y utilizar los tres sub-Motores. También debido a la estructura y diseño del Motor de Juegos se podrá utilizar para el desarrollo y programación de diferentes aplicaciones 3D y videojuegos sin necesidad de cambiar o modificar partes del Motor. Otra ventaja que se pretende con esta estructura, es que si se desean agregar nuevas funciones o métodos al Motor se pueda hacer con facilidad y se podrán hacer actualizaciones del Motor sin necesidad de reescribir código en las aplicaciones 3D o videojuegos creados anteriormente con el Motor. Finalmente para un máximo aprovechamiento de recursos de hardware se distribuirá el trabajo al CPU y al GPU con ayudad de lenguajes especializados en cada unidad de procesamiento como lo son SSE y HLSL, ambos implementados desde la plataforma de programación C++.

3.2 Diseño

Como cualquier otro proyecto en esta rama de los videojuegos, una parte importante y primordial para diseñar un motor de juegos es ponerle un nombre o título. Algunos pensarían que es algo irrelevante, pero es el primer paso en la elaboración de un proyecto. Este motor de juegos se llamará SIÓN. Este nombre me gustó ya que en la trilogía de Matrix, Sión es la última ciudad que no es dominada por las máquinas y en donde se encuentran y habitan los humanos que han despertado de la Matrix. Es la última ciudad o sociedad humana que queda en todo el mundo. Por lo tanto ya que este Motor de Juegos se iba enfocar más en las partes técnicas y a programar casi directamente con el hardware, el hombre dominará y manejará a las máquinas, y no al revés, por lo que el nombre es en honor al triunfo de Sión, los hombres, sobre las máquinas.

Algunos requisitos necesarios en la elaboración de un motor de juegos, son que este pueda Renderear de manera eficiente escenas y modelos 3D, que se puedan administrar sonidos ambientales o de acciones, y que tengan entradas ya sea del Mouse, Teclado o Joystick Debido a estas características esenciales en el diseño de un motor de juegos es que SION cuenta con un Motor Gráfico, un Motor de Audio y un Motor Input.

Los aspectos en que nos enfocamos para la elaboración de este Motor eran que fuera reutilizable, modular y que aprovechara lo mejor posible el hardware de las computadoras. Con esto mencionado antes es que cada sub-Motor de SION es modular y autónomo por lo que pueden ser utilizados por separado o juntos, dependiendo de las necesidades de los diseñadores y programadores de video juegos. Esto es posible ya que cada sub-Motor esta diseñado en base a una interfaz en una Librería Estática (.lib) que es implementada por una Dynamic Link Library (.dll). La Librería Estática se encarga de cargar el DLL. Podríamos decir que este motor también es independiente del API, ya

que la interfaz es independiente del API, y solamente necesitaríamos cambiar la implementación de la interfaz con el API deseado, pero utilizando la estructura de la interfaz. También estos sirve por si existen actualizaciones del API, o se desea cambiar el API, el video juego que fue desarrollado en base a SION no necesita ser reprogramado ni necesita cambios en el código, simplemente se tendría que reprogramar o hacer cambios en la implementación de la interfaz, en el DLL. Los API's que usa este motor de juegos son Direct3D, DirectInput, DirectMusic y DirectSound de DirectX por lo que estamos limitando la plataforma a solamente Microsoft Windows. Se podría hacer independiente de plataforma si se utilizará otro API que fuera no solamente para Windows, y como antes lo mencionaba con este diseño se puede cambiar el API. También con este diseño se podrán cargar y compilar los archivos shaders en el lenguaje HLSL sin necesidad de recompilar todo el código, simplemente guardando los cambios en estos archivos.

Con la siguiente figura se pretende explicar un poco mejor como va a estar diseñado o cual va a ser la estructura de este Motor de Juegos, en donde cada sub-motor va a ser independiente el uno del otro y van a poder ser utilizados por separado. Otro aspecto a considerarse es que con este diseño, solamente se necesitará agregar a la carpeta en donde se este desarrollando un video juego o una aplicación 3D los DLL's y una carpeta con las librerías y los headers necesarios. Así si existen cambios en alguno de los sub-Motores simplemente se tendrá que actualizar el DLL, reemplazando al viejo por el nuevo.

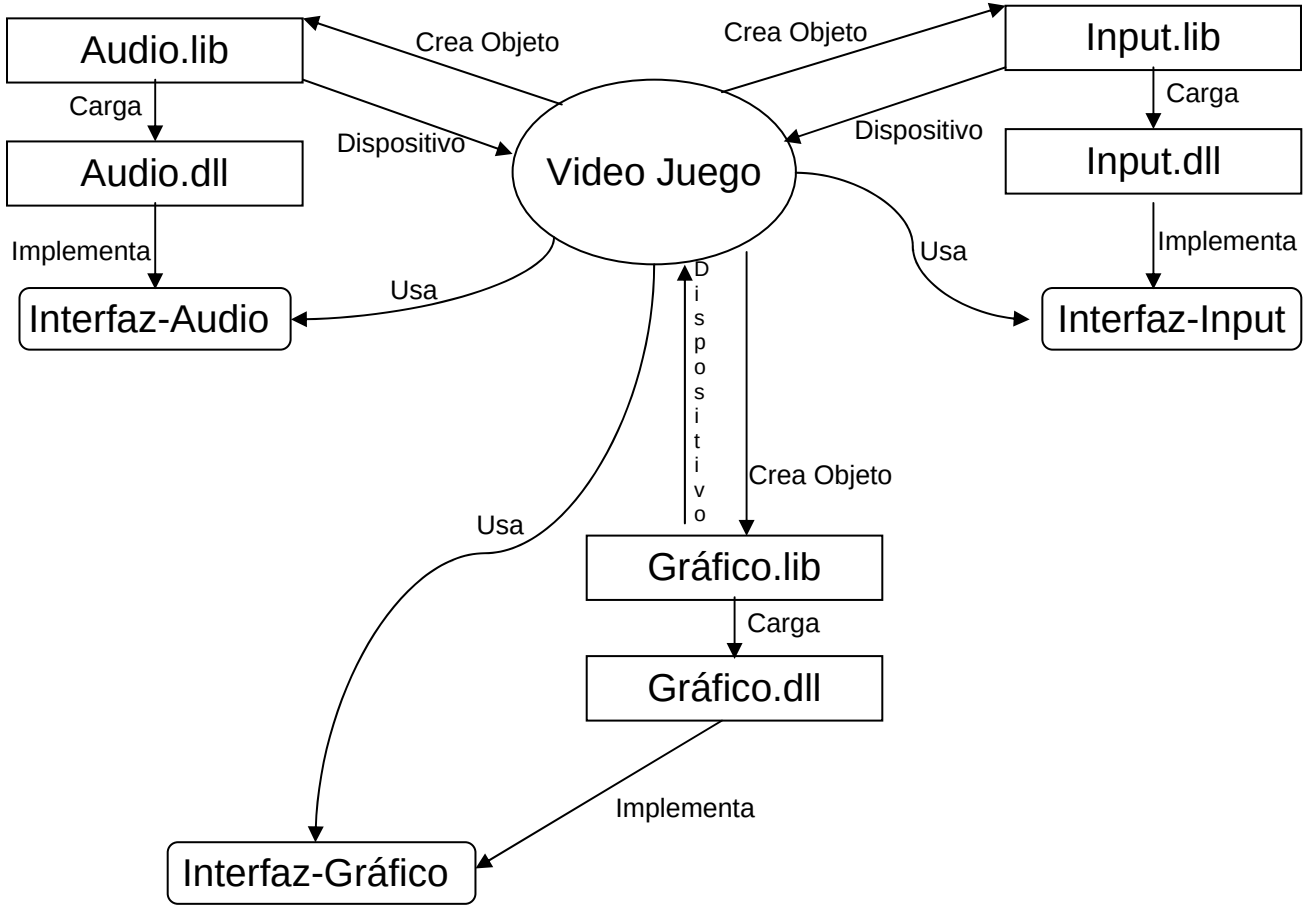

Figura 8 Diseño Motor de Juegos SION

3.3 Motor Gráfico

El motor gráfico es la parte más compleja y más interesante de SION. Este sub-Motor es el encargado de todo lo relacionado al dispositivo render. El nombre de la interfaz de este sub-Motor es SIONDispositivoRender. El DLL encargado de implementar esta interfaz es SIONDirect3D, ya que el API utilizado es Direct3D, y finalmente la Librería Estática encargada de cargar el DLL y en donde se encuentra la interfaz se llama SIONInterfazRenderer.

Para poder crear el ambiente de trabajo de cada sub-motor en Visual C++ .NET es muy sencillo. Lo único que hay que hacer es abrir el IDE, crear un nuevo proyecto como Win32 Project y en Application Settings especificar Librería Estática o DLL. Para el sub-Motor Gráfico primero se creó una Librería Estática la cual se encargaba de cargar el DLL y en la cual se encontraba la interfaz que iba a ser implementada. Para declarar esta interfaz, se añadió a la Librería Estática SIONInterfazRenderer un archivo "Header" llamado SIONDispositivoRender, en el cual está definida la interfaz como una clase abstracta. Primordialmente se debe definir bien la interfaz y que funciones deben ser declaradas en esta, ya que todo el Motor de Juegos va a estar conectado de alguna manera a esta interfaz. Estas funciones a definir no deben ser implementadas aquí, solamente declaradas como virtuales con la palabra utilizada en C++ "virtual" y con la expresión "=0" al final de la declaración para hacerla puramente virtual. Como lo podemos ver en las siguientes líneas de código de la interfaz.

```
virtual SIONAdministradorSkin* GetAdministradorSkin(void)=0;
virtual SIONAdministradorVertexCache* GetAdministradorVertex(void)=0;
virtual bool IsModoVentanas(void)=0;
virtual void GetResolucion(POINT*)=0;
```

Todas las demás funciones que utilicen funciones de un API específico deben ser omitidas dentro de esta definición, para que así la interfaz sea independiente del API. Al final de este archivo se encuentra una parte en donde se especifica que las funciones CrearDispositivoRender( ) y DestruirDispositivoRender( ) no serán implementadas aquí, pero se necesitara trabajar con ellas en esta Librería Estática. Esto es posible declarándolas como externas, con la palabra “extern” ya que estas funciones serán implementadas en el DLL:

```
extern "C" {
  HRESULT CrearDispositivoRender(…);
  typedef HRESULT (…)(…);

  HRESULT DestruirDispositivoRender(…);
  typedef HRESULT (…)(…);
  }
```

En el proyecto SIONDirect3D es donde finalmente se implementan todas las funciones declaradas en la interfaz. Todo lo relacionado a las funciones que deben ser inicializadas, lo que tiene que ver con la enumeración de los dispositivos, adaptadores, resoluciones y el buffer, y lo que se necesite en tiempo de corrida. También es aquí donde se implementa el administrador de materiales, texturas, skins y memoria, el administrador del buffer y el rendereo y donde se crea el dispositivo render y lo relacionado a los shaders. También en este proyecto se agrega un dialogo, en donde va a ser desplegada toda la información del adaptador de video para que el usuario final pueda seleccionar las diferentes características y modos que soporta la computadora en donde se va a desarrollar el video juego.

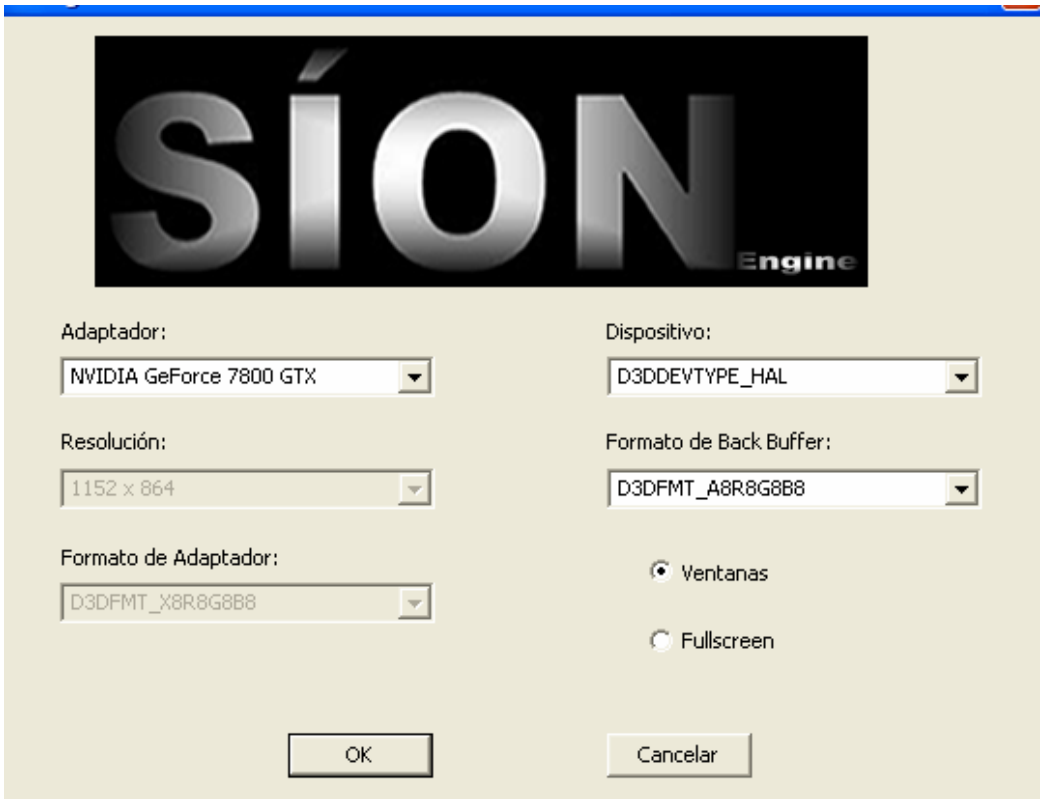

Figura 9 Dialogo Inicial de SION donde se seleccionan las características disponibles

Finalmente, dentro de SIONDirect3D se encuentra un archivo .def, en donde se especifican las funciones que van a ser exportadas del DLL, y que habíamos declarado como externas en la interfaz SIONDispositivoRender para que la Librería Estática pueda trabajar con ellas:

```
;Archivo SIONDirect3D.def
;FUNCIONES QUE VAN A SER EXPORTADAS DE ESTE DLL

LIBRARY   SIONDirect3D
EXPORTS
  CrearDispositivoRender
  DestruirDispositivoRender
```

En el archivo “header” SION.h, es donde se declaran las estructuras que va a utilizar este motor gráfico, como lo son los tipos de vertex, los tipos de enumeración, y las estructuras específicas de SION. Por ejemplo la estructura Material, Skin, Color, Textura y Luz.

```
typedef struct SIONSKIN_TYPE {
  bool  bAlpha;
  UINT  nMaterial;
  UINT  nTextura[8];
  } SIONSKIN;

typedef struct SIONCOLOR_TYPE {
  union {
    struct {
      float fR;
      float fG;
      float fB;
      float fA;
      };
    float c[4];
    };
  } SIONCOLOR;
```

También aquí se especifican algunos tipos de errores, para poderlos identificar bien cuando ocurra alguno:

```
// Errores específicos
#define SION_CREARAPI                0x82000002
#define SION_CREARDISPOSITIVO        0x82000003
#define SION_CREARBUFFER             0x82000004
#define SION_PARAMETROINVALIDO       0x82000005
#define SION_IDINVALIDO              0x82000006
#define SION_TAMAÑOBUFFER            0x82000007
#define SION_BUFFERLOCK              0x82000008
#define SION_NOCOMPATIBLE            0x82000009
#define SION_OUTOFMEMORY             0x8200000a
#define SION_ARCHIVONOENCONTRADO     0x8200000b
#define SION_ARCHIVOINVALIDO         0x8200000c
#define SION_NOSOPORTASHADERS        0x8200000d
```

El administrador de skins es la implementación encargada de todo lo relacionado a los materiales, texturas, luz, manejo de imágenes y transparencia de las figuras. En esta clase la estructura principal es un SKIN, que cuenta con un Material, hasta 8 Texturas por material, y el manejo de transparencia. Las funciones principales de este administrador es cuidar que no se carguen texturas o materiales iguales, y si existen solamente se cargue una y se guarde una referencia sobre esta para que materiales

iguales y texturas iguales no alteren el rendimiento del motor y la memoria en una forma negativa a la hora de cargarlos varias veces. En la estructura de Material, se guardan estructuras de tipo SIONColor que son los flotantes correspondientes a RGBA, y en los cuales se puede especificar que tipo de luz existe, como lo pueden ser luz diffuse, ambient, specular y emissive. También esta clase es la encargada de poder cargar archivos de DirectX como lo son los .x files y renderearlos. Estos archivos .x son cargados, y los materiales y texturas que poseen, también son agregados al administrador de skins.

El administrador de vertex cache, es donde se hacen las operaciones relacionadas a las proyecciones, ortogonales y perspectiva. Es aquí donde se manejan las matrices encargadas de la vista y del posicionamiento, al igual que sus proyecciones y la orientación de la cámara. Otra parte importante de esta implementación es la de los estados del renderer, los vertex buffer e index buffers, y es esta la encargada de renderear todos los primitivos y vértices. En si es esta parte la encargada de presentar en la pantalla las imágenes o los modelos a los usuarios.

Dentro de la clase llamada SIONDirect3D_varios, es donde se implementa lo relacionado a los shaders. Para esto se necesitan declarar antes los tipos y estructuras vertex, en los headers SION.h y SIONDirect3D.h:

```
typedef struct PVERTEX_TYPE {
  float          x, y, z;
  } PVERTEX;

typedef struct VERTEX_TYPE {
  float          x, y, z;
       float  vcN[3];
  float  tu, tv;
  } VERTEX;
```

```
typedef struct LVERTEX_TYPE {
  float          x, y, z;
      DWORD  Color;
  float  tu, tv;
  } LVERTEX;
```

Ahora si para poder utilizar los shaders primero se verifica si el hardware soporta shaders y que versión de estos, para poder crear y activar posteriormente un shader:

```
if (d3dCaps.VertexShaderVersion < D3DVS_VERSION(1,1) ) {
    Log("warning: Vertex Shader Version < 1.1 => no support");
    m_bCanDoShaders = false;
    return;
    }

if (d3dCaps.PixelShaderVersion < D3DPS_VERSION(2,0) ) {
    Log("warning: Pixel Shader Version < 1.1 => no support");
    m_bCanDoShaders = false;
    return;
    }
```

En este motor de juegos, los shaders utilizados son High Level Shading Language de DirectX y pueden ser cargados desde un archivo no compilado o desde un archivo compilado. Tanto los píxel y vertex shaders son soportados por SION. Las funciones que cargan y compilan los shaders son CrearHLSLPixelShader( ) y CrearHLSLVertexShader( ), que a la vez llaman a las funciones del API D3DXCompileShaderFromFile( ), y luego son activados con la función SetVertexShader( ) y SetPixelShader( ). Ahora si nuestro motor esta listo para poder dar efectos impresionantes programando directamente el GPU con la ayuda de los shaders. El ejemplo más sencillo de un vertex shader y un píxel shader en HLSL son los siguientes:

```
//vertex_shader.vsh
float4x4 worldViewProj;

struct VS_INPUT
{
	float3 position  : POSITION;
	float4 color0    : COLOR0;
	float2 texcoord0 : TEXCOORD0;
};

struct VS_OUTPUT
{
	float4 hposition : POSITION;
	float4 color0    : COLOR0;
	float2 texcoord0 : TEXCOORD0;
};

VS_OUTPUT main( VS_INPUT IN )
{
	VS_OUTPUT OUT;

    float4 v = float4( IN.position.x, IN.position.y, IN.position.z, 1.0f );
   OUT.hposition = mul( v, worldViewProj );
   OUT.color0    = IN.color0;
   OUT.texcoord0 = IN.texcoord0;
   return OUT;
}

////////////////////////////////////////////////////////////////////////////////////////////////////
//pixel_shader.psh
struct VS_OUTPUT
{
	float4 hposition : POSITION;
	float4 color0    : COLOR0;
	float2 texcoord0 : TEXCOORD0;
};
struct PS_OUTPUT
{
	float4 color : COLOR;
};

sampler testTexture;

PS_OUTPUT main( VS_OUTPUT IN )
{
	PS_OUTPUT OUT;

	OUT.color = tex2D( testTexture, IN.texcoord0 );
	return OUT;
}
```

Dentro de este sub-motor, también tenemos otra librería estática llamada SIONCamara que se encarga de todos los movimientos y tiempos de la cámara, al igual que su administración. Para la cámara libre se utilizan los Quaterniones que nos ayudan a la prevención del Gimbal Lock. Esta cámara libre cuenta con 6 grados de libertad, mientras que la cámara en primera persona es más restringida ya que la rotación solo es en los ejes del mundo. Para prevenir el Gimbal Lock en este último tipo de cámara solo se puede rotar 80º para arriba y abajo, y dar un poco mas de realidad al solo poder mover un poco la cabeza para arriba y para abajo. En el modo cámara en primera persona se utilizaran ángulos Euler. Para esta librería, el usuario final puede asignar nuevos valores de rotación, dando una mayor funcionalidad y flexibilidad a los movimientos de la cámara.

Otra pequeña librería estática que también pertenece al sub-Motor gráfico es la llamada SIONMatematicas3D. En esta otra librería, se lleva a cabo todo lo relacionado a las matemáticas para los gráficos en 3D. Debido a que SION es un motor en el que se trata de aprovechar al máximo los recursos para poder presentar imágenes en 3D lo más reales posibles, también influyen las matemáticas sobre los gráficos 3D, cuyas operaciones deben ser lo más rápidas posibles y aprovechando al máximo las capacidades de la computadora. Es por esto que no se quiere sobrecargar el trabajo al GPU, sino también aprovechar el CPU, y explotarlo para lo que esta hecho, operaciones matemáticas. De esta manera el GPU se encarga de los shaders y el rendering, mientras el CPU esta haciendo operaciones matemáticas y así distribuir el trabajo en toda la computadora. Para esta librería matemática se utilizó el lenguaje SSE (Streaming SIMD Extensions). En esta librería es donde se evalúa si la computadora en al que se esta

trabajando, soporta este tipo de lenguaje, y sino simplemente utiliza operaciones matemáticas en C++:

```
// Función que verifica el soporte de SSE en el Sistema Operativo
bool OSSupportsSSE() {
  __try
    {
    _asm xorps xmm0, xmm0
    }
  __except(EXCEPTION_EXECUTE_HANDLER) {
    if (_exception_code() == STATUS_ILLEGAL_INSTRUCTION)
      return false;  // sse no soportado por SO
    return false;     // excepción ocurrida
    }
  return true;
  }

// Función que verifica el soporte de SSE en el CPU
INFORMACIONCPU GetCPUInfo() {
  INFORMACIONCPU info;
  char *pStr = info.vendor;
  int n=1;
  int *pn = &n;

  memset(&info, 0, sizeof(INFORMACIONCPU));
  __try {
    _asm {
      mov  eax, 0         // eax=0 => CPUID returns vendor name
      CPUID               // perform CPUID function
      mov  esi,    pStr
      mov  [esi],   ebx    // first 4 chars
      mov  [esi+4], edx    // next for chars
      mov  [esi+8], ecx    // last 4 chars
      mov  eax, 1         // EAX=1 => CPUID returns feature bits
      CPUID               // perform CPUID (puts feature info to EDX)
       test edx, 02000000h  // test bit 25 for SSE
      jz   _EXIT1        // if test failed jump
      mov  [info.bSSE], 1  // set to true
_EXIT1:  // nothing to do anymore
      }
    }
  __except(EXCEPTION_EXECUTE_HANDLER) {
    if (_exception_code() == STATUS_ILLEGAL_INSTRUCTION)
      return info;        // cpu inactive
    return info;          // unexpected exception occurred
    }
  return info;
  }
```

Existen también operaciones matemáticas que no vale la pena implementar en SSE, ya que este lenguaje no es muy rápido cuando se trata de copiar registros, como por ejemplo la suma, resta, multiplicación y división de vectores las implementaremos en C++. Las operaciones en las que usaremos SSE son cuando normalicemos un vector, cuando se utilice alguna raíz cuadrada, cuando se multiplique un vector por una matriz, cuando se multipliquen dos matrices, o cuando se necesite la inversa de una matriz.

```
// Función que obtiene la longitud de un vector utilizando una raíz cuadrada
__asm {
    mov    esi,   this          ; vector u
    movups xmm0,  [esi]         ; first vector in xmm0
    mulps  xmm0,  xmm0          ; mul with 2nd vector
    movaps xmm1,  xmm0          ; copy result
    shufps xmm1,  xmm1, 4Eh     ; shuffle: f1,f0,f3,f2
    addps  xmm0,  xmm1          ; add: f3+f1,f2+f0,f1+f3,f0+f2
    movaps xmm1,  xmm0          ; copy results
    shufps xmm1,  xmm1, 11h     ; shuffle: f0+f2,f1+f3,f0+f2,f1+f3
    addps  xmm0,  xmm1          ; add: x,x,f0+f1+f2+f3,f0+f1+f2+f3
    sqrtss xmm0,  xmm0          ; sqroot from least bit value
    movss  f, xmm0          ; move result from xmm0 to edi
    }
```

Es también aquí donde se implementara una clase relacionada a los Quaterniones, cuya funcionalidad facilita el cálculo de diferentes transformaciones. El Motor gráfico de SION sólo utilizará la multiplicación de estos.

3.4 Motor Input

Otra parte del Motor de Juegos SION, es su sub-Motor Input. Este sub-Motor es muy importante, casi tan importante como el sub-Motor Gráfico. Un video juego sin ningún tipo de entrada por el usuario no sería juego, sería mas como un película. A SION se le agrega este sub-Motor con la misma estructura del sub-Motor Gráfico, que consta con una interfaz, una librería estática y un DLL. Esto se hace ya que otro objetivo de SION es que sea fácil de utilizar e implementar por los usuarios finales, y de este modo si el usuario ya sabe crear e inicializar un dispositivo render, por lo tanto sabrá como crear e inicializar un dispositivo input.

A esta librería estática se le llama SIONInterfazInput, en donde se encuentra el archivo header llamado SIONDispositivoInput en el cual se encuentra definida la interfaz para el sub-Motor Input. En los archivos SIONInterfazInput.h y .cpp se encuentra la implementación de esta librería estática y en donde se manda cargar el DLL para este sub-Motor llamado SIONDirectInput. Dentro del DLL es donde se implementa la interfaz y todo lo relacionado al manejo del Teclado, el Mouse y los Joysticks. Cabe mencionar que para el manejo y monitoreo del Mouse se utilizará la técnica de BufferedData y para el teclado y el joystick se utilizará la técnica de nonBufferedData. Dentro de este DLL se maneja y se administran los dispositivos de entrada antes mencionados, en donde se checa cuantos dispositivos se encuentran conectados y cuales. Para el teclado se podrán utilizar 256 teclas, para el ratón 3 y para el joystick dependiendo de cuantos botones se encuentren en este. Para todos los dispositivos se podrán implementar las funciones de Pressed y Release. Algunos detalles importantes sobre el dispositivo de entrada Joystick son que el motor solo podrá

identificar el primero que se encuentre en la enumeración, no se soportará el modo Feedback y se podrán utilizar ya sea cualquier tipo de joystick o joypads.

El API utilizado para este sub-Motor será DirectInput. Como podemos ver, al seguir con la misma estructura del sub-Motor Gráfico, es que este sub-Motor puede ser reemplazado por otro que utilice otro API, y al ser totalmente independiente de los otros sub-Motores, no se afectarían en nada. También cabe recalcar que otra ventaja de tener esta estructura es que para poder utilizar este sub-Motor solo basta con poner en la misma carpeta en donde se este desarrollando alguna aplicación o video juego la librería estática, el DLL y los archivos SIONDispositivoInput.h y SIONInterfazInput.h.

3.5 Motor de Audio

Al igual que los otros dos sub-Motores, en este se utiliza la misma estructura de la interfaz, la librería estática y el DLL. Como ya lo había mencionado antes, esto es para que al usuario final se la faciliten o se acostumbre a una misma arquitectura en todos los componentes de SION. Si el usuario sabe crear e iniciar un dispositivo render, o un dispositivo input, sabrá como hacerlo con un dispositivo de audio. Para este sub-Motor de Audio, la librería estática se llama SIONInterfazAudio, y la interfaz lleva el nombre de SIONDispositivoAudio. El DLL que implementa esta interfaz se llama SIONDirectAudio.

La función principal de este sub-Motor es la administración y control de archivos de sonido. Ya que para este se utiliza el API DirectMusic y DirectSound, se pueden cargar y tocar muchos formatos de audio. También este sub-Motor necesita de la pequeña librería SIONMatemáticas3D del sub-Motor Gráfico, ya que otra función importante es el poder tener efectos de 3D Sound, para poder dar un poco de realidad en el sonido, y necesitamos de los vectores y la posición o vista de la cámara. También se pueden cargar muchos archivos de sonido y poderlos mezclar para poder dar efectos más reales al video juego o aplicación en la cual estemos utilizando SION. Al igual que los Skins y las Texturas en el sub-Motor Gráfico, aquí se administran los sonidos para que no haya un desperdicio de memoria cuando se este utilizando el mismo archivo varias veces. La implementación de todas estas funciones con respecto al cargar y abrir archivos de sonido y dar efecto 3D se nos facilitó mucho debido a los API’s utilizados.