FACULTAD DE INGENIERIA – INSTITUTO DE COMPUTACIÓN

INTRODUCCIÓN A LA COMPUTACIÓN GRÁFICA

OBLIGATORIO Nº1

MATIAS FABRIZIO PÉRES – GERMÁN RUIZ
MAYO DEL 2012

1 ÍNDICE

2 INTRODUCCIÓN

En el presente trabajo se describe el proceso de resolución del obligatorio número uno de la materia Introducción a la Computación Gráfica del Instituto de Computación

La realización del trabajo se ha enfocado de manera tal que sirva como guía para el lector, no solo como solución inmediata de los clausulas, sino necesario, relevante, mediante el cual se pueda lograr una comprensión total y detallada de las cláusulas establecidas y de las soluciones.

Los objetivos son: cumplir con las cláusulas, ahondar en la utilización de los conceptos aprendidos, lograr un mejor manejo del entorno de desarrollo y ser lo más descriptivos posible de forma de lograr un desarrollo coherente, explicando las decisiones tomadas en cada sección del laboratorio.

En el capítulo 3, se realiza un acercamiento al marco teórico de las distintas herramientas utilizadas para la resolución del obligatorio. Entre estas destaca, el uso de Display Lists, Vertex Arrays y Texturas. Sus funciones y como las desempeñan. Intentamos focalizar la mirada en los conceptos tratados más adelante.

En el capítulo 4 se describe la resolución del obligatorio. Se realizan las observaciones necesarias en cuánto a detalles técnicos, siguiendo las bases establecidas en la teoría e introduciendo cambios y decisiones. Es importante señalar que en ningún momento se introducen cambios ajenos a las cláusulas establecidas por el obligatorio.

El trabajo se encuentra respaldado por material bibliográfico confiable que ha sido seleccionado con atención, a efectos de que la información en la cual nos basemos para la realización del mismo sea certera, lo que brinda constantemente una seguridad en la lectura.

2.1 CLAUSULAS DEL OBLIGATORIO

El obligatorio consiste en desarrollar una variante en 3D del popular juego *Lunar Lander*, publicado por *Atari Inc.* en el año 1979 (aunque la primera versión es de diez años antes). Es un juego estilo *ARCADE* en donde el jugador debe guiar una nave por el espacio, activando sus cohetes, con el objetivo de que alunice con cuidado sobre una superficie horizontal. Se cuenta con

"combustible" limitado. El conjunto de requerimientos se detallan en la letra del obligatorio. A continuación, se presentan los más relevantes:

- El juego está gobernado por las flechas del teclado (o teclas direccionales). Es posible rotar el juego para cambiar el punto de vista de observación a través del uso del mouse, mientras se presiona el botón derecho.
- **Ajustes (settings)**: Se debe disponer de una interfaz para ajustar los siguientes parámetros:
    - Velocidad del juego; wireframe (on/off); facetado/interpolado; texturas (on/off).
    - Dirección y color de por lo menos una luz.
- Sobre cargado y renderizado de texturas y modelos 3D (por lo menos una textura y un modelo). No es mandatorio que los modelos cargados formen parte de la simulación del juego (pueden formar parte del fondo o de algún efecto visual). **La programación del "render" del modelo 3D debe ser realizada por los estudiantes.**
- Se debe generar el terreno a partir de una malla regular donde la altura y ubicación cada vértice corresponde con un píxel de un mapa de altura.
- Se implementará al menos un tipo de enemigo.
- Se contemplará que la nave se encuentra bajo los efectos del campo gravitatorio lunar, por tanto debe "simular" un comportamiento físicamente realista. Cuando ocurra una colisión con la superficie lunar, debe controlarse a qué velocidad ocurre y qué orientación tiene la nave y el terreno. Si la velocidad es inferior a cierto umbral, y la nave y el terreno están en posición aproximadamente horizontal, entonces el alunizaje es exitoso.
- Game HUD básico dibujados con una proyección ortogonal.

3 CONCEPTOS PREVIOS

3.1 OPENGL

3.1.1 Display Lists

Una *display list* es simplemente un grupo de comandos GL y argumentos que se guardan para sus siguientes ejecuciones. Cuando se procesa una *display list* se puede proveer un número que especifica esta lista de forma única. Llamando a la *display list* con este identificador causa que los comandos dentro de la lista se ejecuten como si fueran llamados normalmente, con la excepción en los comandos que dependen del estado del cliente. Cuando un comando se guarda en una *display list*, el estado del cliente en efecto en ese momento es el que se aplica en los comandos. Es decir que si en la lista se guarda un comando que referencia a una variable que se encuentra en la memoria de la aplicación y luego esa variable cambia, el comando de la lista no se ve afectado. Solo el lado del servidor se ve afectado cuando se ejecuta un comando de una *display list.*

Para crear una *display list* se utiliza el siguiente comando:

void ***NewList****(uint n, enum mode);*

Donde *n* es un entero positivo que se le asigna a la *display list* que se está creando, es decir es el identificador de la lista a crear. El segundo parámetro *mode* es una constante que le indica a OpenGL el comportamiento durante la creación de la *display list*. Si *mode* es COMPILE, entonces los comandos no son ejecutados a la hora de ser puestos en la lista. Si es COMPILE_AND_EXECUTE, los comandos son ejecutados cuando son encontrados y después se les coloca en la *display list.*

Todos los comandos GL después de *NewList* son guardados en la lista hasta que una llamada a este comando ocurre:

void ***EndList****(void);*

Cuando esta llamada ocurre la *display list* especificada es asociada con el identificador pasado en *NewList.*

Luego de crear la *display list* se puede llamar a la misma para que se ejecuten los comandos guardados con el siguiente comando:

void ***CallList****(uint n);*

El parámetro *n* siendo el identificador de la *display list*.

También hay otro comando muy útil para la utilización de *display list*, este es:

uint ***GenLists****(sizei s);*

Este comando retorna un entero n tal que los índices n, n + s – 1, no han sido usados antes para otras *display lists.* Ademas crea una *display list* vacía para cada uno de los índices, de esta manera estos índices quedan como usados.

3.1.2 Vertex Arrays

Los comandos para especificar vértices comunes aceptan información en muchos formatos, pero su uso requiere muchas ejecuciones de distintos comandos de GL para especificar hasta simples geometrías. La información de los vértices de la geometría a dibujar también se puede guardar en el espacio de direccionamiento del cliente. Bloques de información guardados en estos arreglos que contienen la información de los vértices, pueden ser usados para especificar múltiples primitivas de la geometría con un solo comando GL. De esta forma usando vertex arrays se reduce la cantidad de llamadas a funciones y uso redundante de vértices en común, por lo tanto se incrementa la performance de forma considerable.

El cliente puede especificar muchos arreglos con los datos de los vértices de la geometría, entre ellos están: las coordenadas de los vértices, normales, colores, colores secundarios, índices de colores, coordenadas de la niebla, coordenadas de textura y banderas de los limites (edge flags). Para habilitar el uso de estos arreglos hay que llamar al comando de OpenGL:

void glEnableClientState(GLenum cap);

Donde *cap* es el tipo de arreglo que queremos habilitar. VERTEX ARRAY, NORMAL ARRAY, COLOR ARRAY, SECONDARY COLOR ARRAY, INDEX ARRAY, FOG COORD ARRAY, TEXTURE COORD ARRAY, o EDGE FLAG ARRAY, para el arreglo de vértices, normales, colores, colores secundario, índices de color, coordenadas de niebla, coordenadas de textura, o banderas de límites, respectivamente.

Luego de habilitarlos se tiene que especificar la localización y organización de los arreglos con unos comandos especiales. Para el arreglo de

vértices, normales y texturas, que son los que utilizamos en nuestra aplicación, se utilizan los comandos:

void ***VertexPointer****(int size, enum type, sizei stride, void *pointer);*

void ***NormalPointer****(enum type, sizei stride, void *pointer);*

void ***TexCoordPointer****(int size, enum type, sizei stride,void *pointer);*

El parámetro *type* especifica el tipo de datos de los valores guardados en el arreglo. Luego el parámetro *size* indica el número de valores por vértices que están guardados en el arreglo. Como las normales siempre se especifican con tres valores el comando NormalPointer no tiene el argumento *size*. El parámetro *stride* indica el número de bytes de offset para el siguiente dato. Luego el último *pointer* indica la localización o dirección del arreglo, el cual se encuentra del lado del cliente, es decir en la memoria misma de la aplicación.

Luego de habilitar e indicar la localización y organización de los distintos arreglos a usar ya se está listo para poder usar las ventajas ofrecidas. En nuestra aplicación usamos un comando para dibujar toda la geometría del terreno.

El comando de OpenGL que usamos es:

void DrawElements(enum mode, sizei count, enum type, void *indices);

El primer parámetro *mode* corresponde al tipo de primitiva que se desea dibujar. El segundo *count* indica la cantidad de elementos del arreglo a dibujar. El parámetro *type* indica el tipo de datos de los valores guardados en índices. El ultimo parámetro *índices* guarda los índices de los vértices que se usan para dibujar las primitivas. El efecto de este comando es el mismo que la siguiente secuencia de comandos:

```
if (mode, count, or type es invalido )
        generar un error apropiado
else {
        int i;
        Begin(mode);
        for (i=0; i < count ; i++)
                ArrayElement(indices[i]);
        End();
}
```

Al llamar a *ArrayElement(int i)*, el i-esimo elemento de cada arreglo habilitado es transferido al estado de OpenGL. Por ejemplo si están habilitado los arreglos de vértices y normales y se llama a *ArrayElement(0)*, es como llamar al comando de especificación de vértice y normal con los datos que se encuentran en la posición 0 de cada arreglo.

3.1.3 Texturas

La incorporación de texturas en la aplicación consiste en mapear una parte de una o más imágenes especificadas en cada primitiva de la geometría donde se tiene habilitado el uso de texturas. Este mapeo se logra usando el color de una imagen en el lugar indicado para modificar el color primario del fragmento de la primitiva que se le está aplicando la textura.

Para empezar, primero se necesita un nombre para la textura, que es esencialmente un número que usa OpenGL para identificarla. Para ello se utiliza el siguiente comando:

```
void GenTextures( sizei n, uint *textures );
```

Se retornan *n* nombres de texturas sin usar previamente en *textures*. Estos nombres son marcados como usados, pero recién adquieran estado y dimensiones cuando son referenciadas por primera vez con el comando:

void ***BindTexture****(enum target, uint texture);*

Donde *target* es TEXTURE 1D, TEXTURE 2D, TEXTURE 3D, o TEXTURE CUBE MAP. El parámetro *texture* corresponde a el nombre de la textura o identificador. Este comando no se usa solo para crear este objeto textura, sino que también se usa para decir a OpenGL con que textura queremos trabajar. Si queremos cambiar de textura simplemente se llama a esta función nuevamente con el identificador de textura deseado y el tipo de textura a usar.

Para especificar una imagen textura de dos dimensiones se utiliza el siguiente comando:

void ***TexImage2D****(enum target, int level, int internalformat, sizei width, sizei height, int border, enum format, enum type, void *data);*

En *tarjet* se pueden pasar varias constantes simbólicas pero en el caso de nuestra aplicación solo se utiliza TEXTURE_2D. Los parámetros *format, type* y *data* especifican el formato de los datos de la imagen, el tipo de esos datos y un puntero a los datos de la imagen guardada en memoria. El parámetro *level*

corresponde al nivel de detalle de la textura e *internalformat* el el numero de componentes de color. Luego el *width* y *height* corresponden al ancho y alto de la textura, y *border* a la anchura del borde.

En nuestra aplicación usamos SDL_Image para pasar la imagen que queremos usar como textura a un tipo SDL_Surface. Luego de este tipo de datos sacamos toda la información de los datos de la imagen textura para poder especificarla.

Existen varios parámetros para controlar como el arreglo de texturas es tratado cuando es especificado o cambiado y aplicado a un fragmento. Los comandos para especificar estos parámetros son de la forma:

void ***TexParameter{if}****(enum target, enum pname, T param);*

void ***TexParameter{if}v****(enum target, enum pname,T params);*

Donde *target* es TEXTURE 1D, TEXTURE 2D, TEXTURE 3D, o TEXTURE CUBE MAP. El segundo parámetro es una constante simbólica que indica el parámetro a tratar. En el primer comando *param* corresponde a solo un valor para asignarle al parámetro en cuestión. En el segundo comando es un arreglo de valores para los parámetros cuyo tipo depende en los parámetros a asignar.

Los distintos *pname* que se utilizan son: TEXTURE_WRAP_S, TEXTURE_WRAP_T, TEXTURE_MAG_FILTER y TEXTURE_MIN_FILTER. Para los dos primeros parámetros los valores que se le pueden asignar son: CLAMP y REPEAT. Si usamos el modo CLAMP el tratamiento de las coordenadas de la textura quedan en el rango [0,1], mientras que si usamos REPEAT se ignora la parte entera de las coordenadas de textura usando solo la parte fraccional.

Luego los siguientes parámetros especifican los filtros de ampliación o reducción de la textura. Algunos de los valores que se les puede asignar son: NEAREST y LINEAR. El primer valor agarra el *mipmap* más cercano mientras que el otro filtra el original.

También se puede especificar cómo aplicar la texturas sobre los pixeles, para esto se asigna un estado al ambiente de la textura. El comando utilizado es el siguiente:

void ***TexEnv{if}****(enum target, enum pname, T param);*

El *target* puede ser TEXTURE_ENV o TEXTURE_FILTER_CONTROL, en *pname* puede ir TEXTURE_ENV_MODE o TEXTURE_ENV_COLOR, y en *param* DECAL, BLEND, MODULATE o REPLACE. Si el objetivo es TEXTURE_ENV y se asigna DECAL al parámetro TEXTURE_ENV_MODE se aplica como color el que se lee de la textura, pero si el canal alpha no es 1, se hace *blending*. Con REPLACE es lo mismo ignorando el color alpha. Si se pone MODULATE el color del pixel se escala con el valor de la textura y con BLEND se combinan el color de la textura y el pixel.

Finalmente para que se puedan ver las texturas hay que decirle a OpenGL que las habilite. Para esto se utiliza el comando genérico *Enable* pasándole una de las siguientes constantes simbólicas: TEXTURE 1D, TEXTURE 2D, TEXTURE 3D, o TEXTURE CUBE MAP. Por el contrario para deshabilitarlas se utiliza el comando *disable* con la constante correspondiente a la textura que se quiere deshabilitar.

3.1.4 Iluminación

La iluminación requiere que primero se preparen los modelos. Setear, actualizar el material y proveer por cara o vértice, normales.

1. Habilitar luz; opcional, habilitar por vértice/cara color y sombra suave.

```
glEnable(GL_LIGHTING);
glEnable(GL_COLOR_MATERIAL);
glShadeModel(GL_SMOOTH);
```

2. Habilitar hasta 8 fuentes de luz:

```
glEnable(GL_LIGHTn) // (e.g: GL_LIGHT0)
```

3. Configurar las Fuentes de luz: común a todos los tipos, setear el color para los componentes de luz “ambient”, “diffuse” y “specular”. Ambiente permite iluminar cada punto en una escena. Diffuse permite iluminar objetos alrededor y Specular añade una macha brillante a los modelos.

```
glLightfv(GL_LIGHTn, GL_AMBIENT, color4f );
glLightfv(GL_LIGHTn, GL_DIFFUSE, color4f );
glLightfv(GL_LIGHTn, GL_SPECULAR, color4f );
```

4. La luz puede ser direccional o posicional:

glLightfv(GL_LIGHTn, GL_POSITION, vector4f);

vector4f es (x,y,z,w). Si w= 0 entonces la luz es direccional (similar a la luz solar, con dirección x,y,z. Si w= 1 entonces la luz es posicional como una bola de fuego.

5. Foco de luz

```
glLightfv(GL_LIGHTn, GL_POSITION, vector4f );
glLightfv(GL_LIGHTn, GL_SPOT_DIRECTION, vector3f );
glLightf(GL_LIGHTn, GL_SPOT_CUTOFF, angle); // angle entre 0 to 180
glLightf(GL_LIGHTn, GL_SPOT_EXPONENT, exp); // exponent varia entre 0 y 128
```

valores de exponente altos hace la luz más fuerte en el medio del cono.

6. Atenuación: la atenuación hace que la luz pierda fuerza cuando el modelo se encuentra más lejos de la fuente de luz. Esto puede enlentecer el renderizado.

glLightf(GL_LIGHTn, attenuation, value)

donde atenuación puede ser:

GL_CONSTANT_ATTENUATION
GL_LINEAR_ATTENUATION
GL_QUADRATIC_ATTENUATION

3.2 SDL

Simple DirectMedia Library es una librería multimedia multiplataforma que ofrece funcionalidad para manejo de ventanas, lectura de teclado, reproducción de sonido, y demás características.

Presenta varios sub-sistemas que pueden ser inicializados de forma independiente. En el ejemplo se presenta la inicialización utilizada :

```
if (SDL_Init(SDL_INIT_EVERYTHING) < 0) {
      return false;
  }
```

Para configurar el modo de video y resolución se debe utilizar la función ***SDL_SetVideoMode***. Se puede configurar a SDL para dibujar con OpenGL en vez de usar las primitivas de SDL.

```
if ((Surf_Display = SDL_SetVideoMode(anchoPantalla, altoPantalla, 32, SDL_HWSURFACE
| SDL_OPENGL)) == NULL) {

    return false;

}
```

SDL_Quit(): Libera todos los recursos usados por SDL.

SDL_PollEvent(&evento): se fija si ha sucedido algún evento y devuelve inmediatamente. En **evento.type** está el tipo de evento sucedido.

Uint8* SDL_GetKeyState(NULL): devuelve el estado completo del teclado en un array. Se pueden utilizar constantes de SDL para especificar las posiciones en dicho array. [13]

3.3 NETBEANS Y MINGW

Para la resolución del obligatorio se utilizó el ide NetBeans versión 7.1.1 con los compiladores y herramientas de MinGW.

Ver [14] y [15] para más información y configuración.

4 RESOLUCIÓN

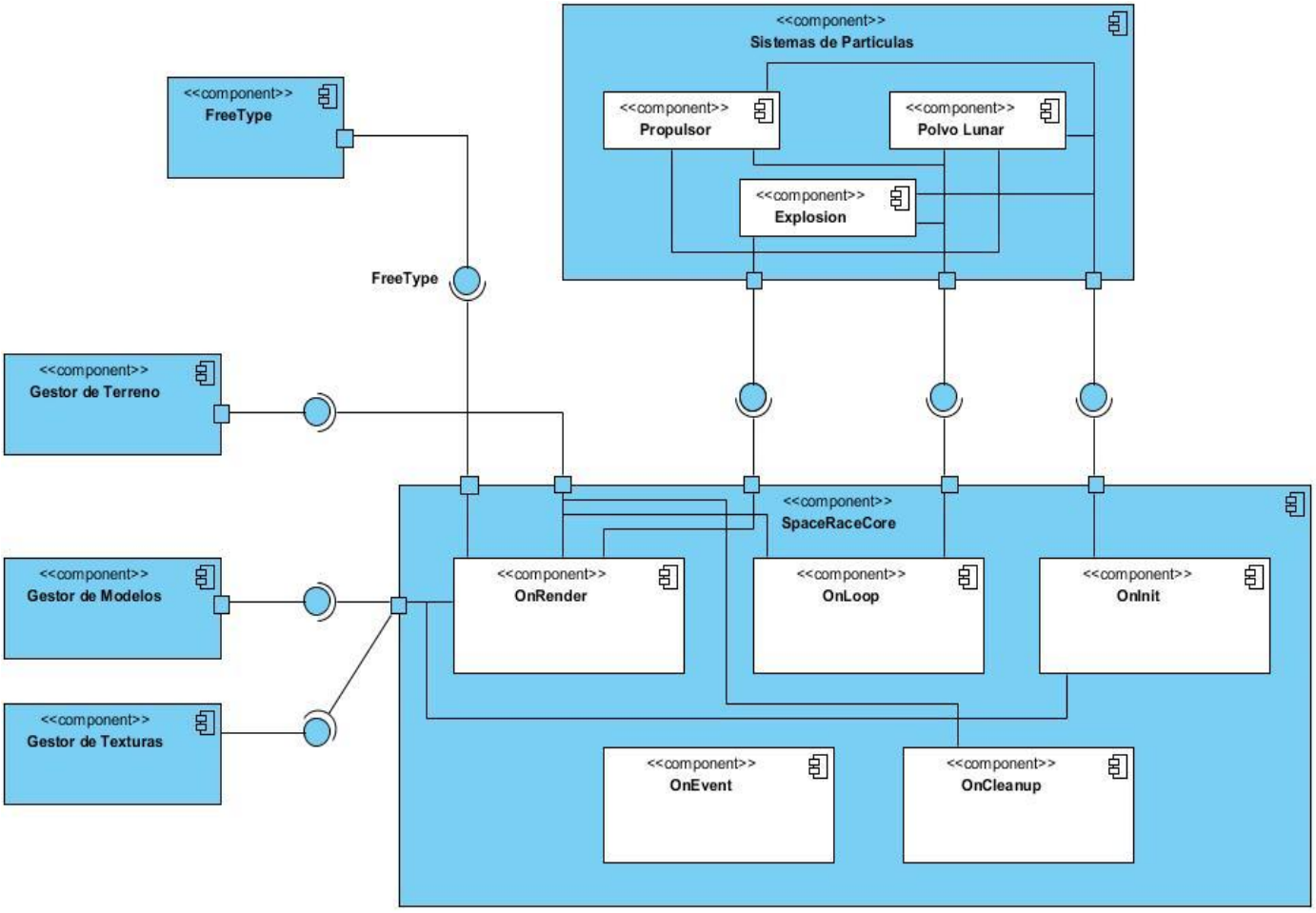
4.1 ARQUITECTURA

Para comenzar la resolución del obligatorio, es necesario tener en cuenta la típica secuencia de ejecución de un videojuego. Dicha secuencia comienza con la inicialización de todas las variables relevantes del sistema. Se abre la ventana y se configuran las librerías. Se inicializa el estado de OpenGL y se procede con una entrada al loop principal del programa. Este es, en definitiva, el que da lugar a las imágenes, el movimiento y la interacción entre el usuario y el videojuego. En el loop principal se chequean los eventos y se toman las decisiones de acuerdo al estado de las variables. Se actualizan la variables y se redibuja la escena.

Una vez comprendido los pasos necesarios para dar lugar a la creación del video juego, desarrollamos una arquitectura capaz de soportar dicho comportamiento pero que a su vez, permitiera la separación entre distintas funcionalidades del sistema, facilitando el trabajo en equipo y el desarrollo seguro hacia una aplicación sólida y eficiente. Otro de los aspectos tenidos en cuenta fue el desarrollo de un código comprensible y coherente, cuyo testeo no imponga dificultades ajenas a la intuición.

Para la presentación de la arquitectura se ha optado por un diagrama de componentes y conectores. Cada elemento del diagrama, como es usual en un diagrama de componentes, se manifiesta durante la ejecución del programa (como ser objetos o librerías), consume recursos y contribuye con el comportamiento del sistema.

En la siguiente imagen se muestra el diagrama realizado. El diagrama muestra instancias, no tipos. Su realización contribuyo al desarrollo del obligatorio, ayudando a visualizar el camino de la implementación, permitiendo tomar decisiones tempranas. Se debe tener en cuenta que se trata de un diagrama primitivo, cuya semántica puede no estar perfectamente definida. En particular, esta vista ayuda a responder la pregunta de cuáles son las entidades más importantes durante la ejecución del videojuego y cómo interactúan.



A simple vista podemos notar dos grandes componentes, SpaceRaceCore y Sistemas de Partículas. SpaceRaceCore es el componente principal del videojuego, encargado del chequeo de eventos, la actualización de variables que almacenan el estado de la aplicación y la renderización en cada frame. SpaceRaceCore se describe a su vez, como una subarquitectura de componentes y conectores. Notar que los componentes interiores a SpaceRaceCore no se

comunican entre sí, sino que requieren interfaces (sockets) provistas por componentes externos (lollipops).

Entre los componentes externos, disponemos de un gestor de modelos, encargado de la administración de los modelos utilizados durante el juego. Dicho componente tiene la responsabilidad de cargar los modelos y disponer su uso cuando sea necesario renderizarlos. Con fin similar se dispone de un gestor de texturas, responsable de la administración de texturas permitiendo enlazarlas ("bind") cuando sea necesario.

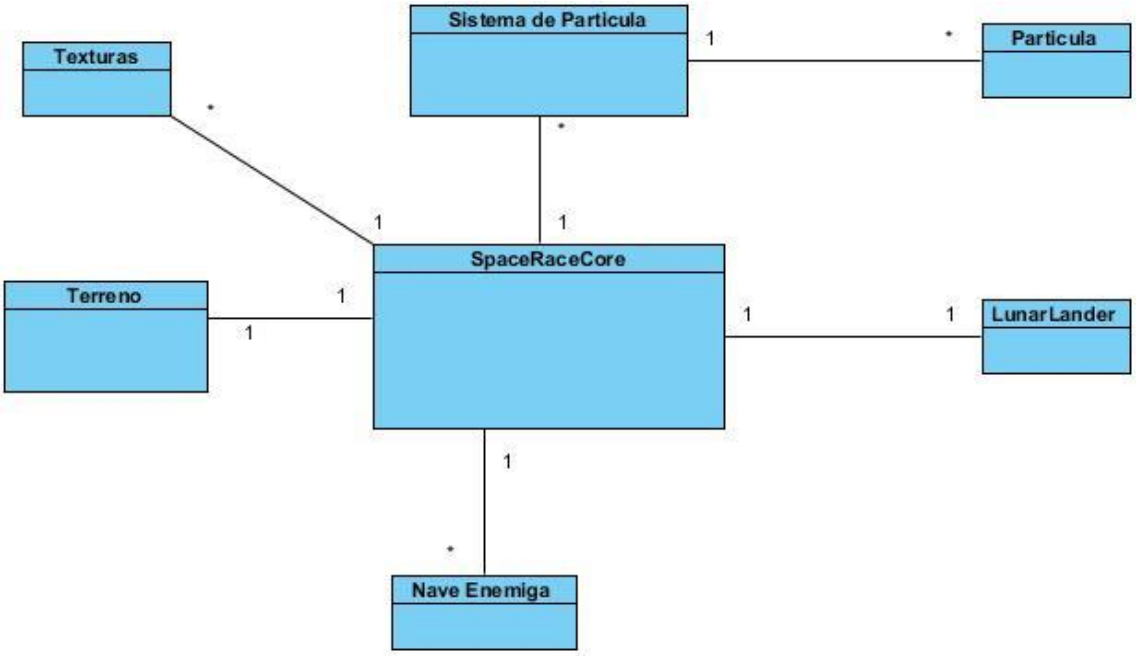
Los sistemas de partículas ponen en escena 3 de los efectos principales del videojuego. Estos son, el propulsor, el polvo lunar y la explosión del lunar lander. Más adelante veremos más detalles de este tipo de sistemas y su funcionamiento.

El componente Terreno, también detallado más adelante, es el encargado de la creación, visualización y eliminación del terreno de juego.

Por último, el componente FreeType, se presenta al momento de generar el GameHud del videojuego.

4.2 ANÁLISIS Y DISEÑO

Un primer modelo de dominio de la solución implico la tarea de discernir el mapeo entre los componentes presentados en la arquitectura a entidades del sistema. Sin embargo, este mapeo no es directo, y un componente puede ser implementado por una o más clases/objetos del sistema, así como varios componentes pueden ser implementados por una clase/objeto. Veamos en la imagen siguiente una versión prematura del dominio de la aplicación.

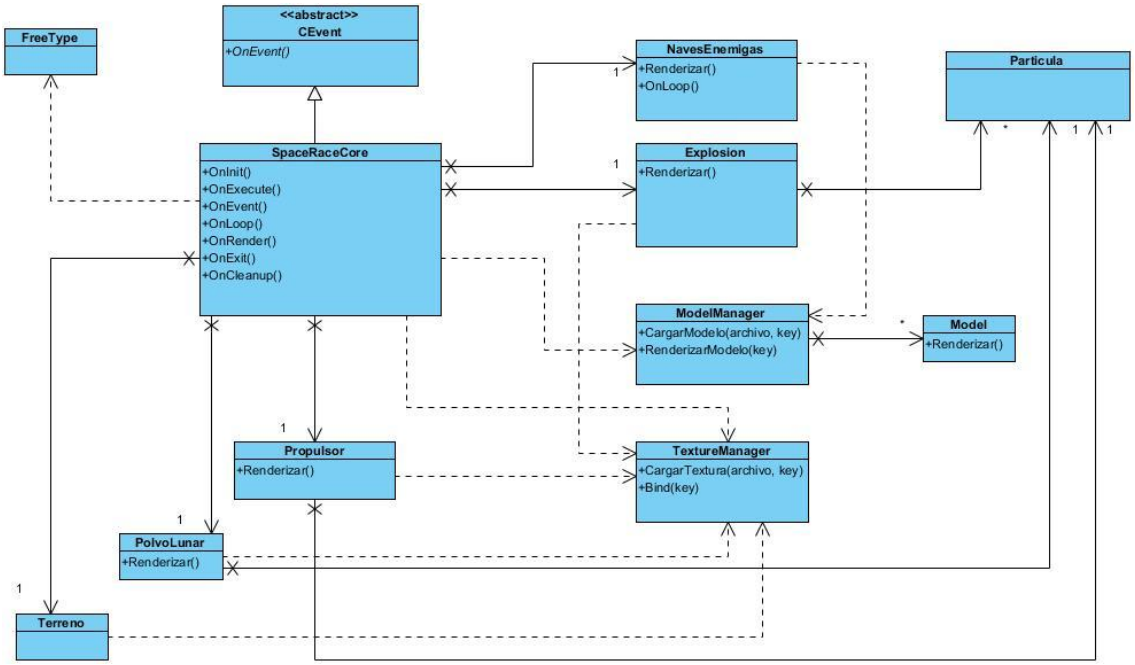


Este modelo de dominio presenta a grandes rasgos las entidades principales presentes en el sistema desde su comienzo. Resalta su baja cohesión, teniéndose una entidad principal en el centro (SpaceRaceCore), que se comunica con casi todas las demás entidades. Resulta evidente que no refleja las propiedades de la arquitectura introducida pero permite simplificar el entendimiento de la realidad.

Con este modelo de dominio se comenzó a trabajar en un código propenso a modificaciones. Casi inmediatamente surgió la necesidad de poseer un verdadero gestor de texturas y modelos. Los sistemas de partículas se hicieron complejos y con características propias demasiado integradas como para tratar los sistemas de forma genérica. Es por ello que se tomo la decisión de poseer una clase por cada sistema de partículas, que, actualmente son 3.

Una vez discutidas las decisiones analíticas de la solución se procedió a diseñar un esquema que soportara la arquitectura presentada en el capítulo anterior y que introdujera las ideas surgidas en la codificación. A pesar de que este no es el orden de trabajo correcto, codificar y luego diseñar, fue necesaria una aproximación, por mínima que fuera de manera de focalizar nuestra atención en aspectos técnicos relevantes. Además, la poca experiencia en el desarrollo de videojuegos por nuestra parte, hace difícil discernir un diseño firme y sustentable desde el comienzo.

La siguiente imagen muestra un diagrama de clases de la aplicación.



En principio podemos observar una baja del acoplamiento al introducir los gestores de texturas y modelos (ModelManager y TextureManager). También resaltan los métodos *OnInit()*, *OnExecute()*, *OnLoop()*, *OnRender()*, *OnExit()* y *OnCleanup()* de la clase SpaceRaceCore y su derivación de la clase abstracta CEvent.

CEvent define el método *OnEvent()*, que es heredado por SpaceRaceCore. Dicho método tiene la siguiente estructura:

```
void CEvent::OnEvent(SDL_Event* Event) {
   switch (Event->type) {
      case SDL_ACTIVEEVENT:
      {
         switch (Event->active.state) {
            case SDL_APPMOUSEFOCUS:
            {
               if (Event->active.gain) OnMouseFocus();
               else OnMouseBlur();
               break;
            }
            case SDL_APPINPUTFOCUS:
            {
               if (Event->active.gain) OnInputFocus();
               else OnInputBlur();
               break;
            }
            case…
```

Lo que hacemos básicamente es, tomar un puntero a SDL_Event, y determinar su tipo. Luego, llamar a la función apropiada. SpaceRaceCore hereda de CEvent por lo que cuando queramos atrapar un evento simplemente debemos sobrecargar una función.

Este comportamiento fue imitado de [5] (donde se ejemplifica la captura de ciertos eventos) por inferir un orden cuidadoso en la implementación.

La clase SpaceRaceCore setea el estado de nuestro programa completo. La mayoría de los juegos consisten de 5 funciones que manejan como el juego se procesa. Los procesos son típicamente:

- Inicializar: esta función maneja todo el cargado de datos, ya sean texturas, mapas, modelos o lo que sea.
- Eventos: esta función maneja todos los eventos de entrada del mouse, teclado, joysticks o cualquier otro dispositivo.
- Loop: se gestionan todas las actualizaciones de las variables, como ser el movimiento de los personajes por la pantalla, la barra de vida, etc.
- Render: esta función maneja el renderizado de todo lo que se imprime en pantalla. No se realiza ninguna manipulación de datos.
- Cleanup: se liberan todos los recursos utilizados asegurando una salida limpia del juego.

Sin ser repetitivos en cuanto a lo mencionado anteriormente, es importante entender que los juegos son un loop gigante. Dentro de este loop encontramos eventos, actualizaciones de variables y renderizado de imágenes.

Disponemos de la siguiente estructura:

```
SpaceRaceCore:: SpaceRaceCore() {
   Running = true;
}

int SpaceRaceCore::OnExecute() {
   if(OnInit() == false) {
      return -1;
   }

   SDL_Event Event;

   while(Running) {
      while(SDL_PollEvent(&Event)) {
         OnEvent(&Event);
```

```
        }

        OnLoop();
        OnRender();
    }

    OnCleanup();

    return 0;
}

int main(int argc, char* argv[]) {
    SpaceRaceCore theApp;

    return theApp.OnExecute();
}
```

Primero inicializamos nuestro juego. Si la inicialización falla, cerramos el programa. Si todo está bien, continuamos con el loop. Dentro del loop del juego utilizamos *SDL_PollEvent* para chequear eventos y pasarlos de a uno por vez a *OnEvent*. Una vez terminamos con los eventos nos movemos a *OnLoop* para la actualización de variables. Luego renderizamos. Repetimos este ciclo indefinidamente. Si el usuario sale del juego, procedemos con *OnCleanup* liberando todos los recursos.

La variable *Running* es nuestra bandera de salida del loop del juego. Cuando se setea en falso, termina nuestro programa. Cuando el usuario presiona la tecla escape, por ejemplo, podríamos setear esta variable en false para salir. No es el comportamiento adoptado.

4.2.1 Velocidad de simulación

Para independizar la aplicación del Frame Rate, cada movimiento realizado, ya sea por parte del lunar lander, de cada una de las partículas de los sistemas de partículas, o el satélite o las naves enemigas, se calcula la nueva posición de los objetos dependiendo del tiempo transcurrido entre dos render consecutivos. Para ello se mantienen las siguientes variables:

```
// Variables Tiempo
    int startTime;
    int prevTime;
    int currTime;
    float timeElapsed;
```

En cada iteración se calcula,

```
currTime = SDL_GetTicks();
timeElapsed = 0.001*(currTime - prevTime);
prevTime = currTime;
```

Cualquier cálculo necesario hace uso de la variable *tiemeElapsed*. *SDL_GetTicks()* devuelve el número de milisegundos desde la inicialización de la librería.

4.3 CÁMARA

Para el movimiento de la cámara disponemos de las siguientes variables:

```
GLfloat xrot_cam,yrot_cam,xtra_cam,ytra_cam,ztra_cam;
GLfloat maxyrot_cam, maxztra_cam, minztra_cam;
```

La variable *xrot_cam* indica cuánto rota la cámara respecto al eje (0,1,0). La variable *yrot_cam* cuanto rota la cámara respecto al eje (1,0,0). Con la "ruedita" del mouse es posible acercar o alejar. Dicho comportamiento se logra con los siguientes llamados, una vez que se modificó *ztra_cam*:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(ztra_cam, (GLfloat) anchoPantalla / (GLfloat) altoPantalla, 0.1, 100);
glMatrixMode(GL_MODELVIEW);
```

Un aspecto interesante es que, para permitir una mejor visualización e interacción, la cámara también percibe la rotación del lunar lander. Pero en forma disminuida, logrando así un movimiento general y no brusco por parte del lunar lander.

4.4 TERRENO

El terreno como dice en la letra del obligatorio es generado a partir de una malla regular donde la altura y ubicación de cada vértice corresponda con un píxel de un mapa de altura. El mapa de alturas utilizado en nuestra aplicación fue generado usando GIMP. Es una imagen de 1024x1024 pixeles en escala de grises y además *tileable*. Se explicara más adelante la razón de que imagen del mapa de alturas tenga esta propiedad.

En nuestra arquitectura tenemos una clase (Terreno) que representa el terreno usado en la aplicación. Nuestra clase principal tiene en uno de sus atributos un objeto de clase Terreno, el cual se crea cuando se inicializa la aplicación. Al constructor de esta clase se le pasa un factor de altura que es usado para darle la altura deseada al terreno.

Lo primero que hacemos en el constructor de esta clase es cargar la imagen del mapa de alturas usando una función de SDL_Image, la misma que usamos para cargar las texturas. Al tener la imagen en memoria como tipo SDL_Surface tenemos todos los datos de la imagen. En el objeto de Terreno tenemos todos los arreglos de datos sobre los vértices de la geometría del terreno.

A partir del ancho y alto de la imagen que lo sacamos de la SDL_Surface le damos el ancho y largo al terreno nuestro. Luego reservamos la memoria de todos los arreglos que vamos a usar. En nuestro caso tendremos un arreglo con las coordenadas de los vértices, las normales de cada vértice, las coordenadas de textura, y además un arreglo de índices que indicara el orden de cómo recorrer los arreglos.

Después de tener todos los arreglos con su memoria reservada recorremos todos los pixeles de la imagen cargada. La información de cada pixel la sacamos de la SDL_Surface cargada. Como la imagen esta en escala de grises cada pixel tiene valor en el rango de [0,255]. A partir del valor del pixel y el factor de altura pasado se calcula la coordenada Y de cada vértice, y con su ubicación en la imagen se calculan las coordenadas X y Z. Estas 3 coordenadas se van guardando en el arreglo de vértices creado previamente.

Para facilitar el juego se colocan unas cuantas zonas de aterrizaje en el territorio. A partir de unas coordenadas X y Z se crean estas plataformas. En esta función se calcula una altura promedio de los alrededores y se le asigna a todos los vértices pertenecientes a esta zona esa altura.

El siguiente paso es rellenar el arreglo de normales previamente creado. Este arreglo es muy importante ya que si no se especificaría la iluminación no actuaria sobre el terreno. El terreno va a estar formado básicamente por QUADS, o siendo más especifico por QUAD_STRIP. Para hallar las normales de cada vértice se hace el producto vectorial de dos vectores pertenecientes a cada quad. Uno de los problemas es que hay vértices que pertenecen a varios quads, en ese caso se hace el promedio de la suma de los vectores normales hallados.

Luego se rellena el arreglo de coordenadas de texturas de cada vértice. Para tener una mejor vista de la textura aplicada al terreno no se coloca la imagen de la textura entera en cada quad, sino que se coloca solo una fracción pequeña. La textura usada tiene sus parámetros de WRAP_S y WRAP_T en REPEAT, de esta manera se ignoran los límites de [0,1].

Teniendo estos tres arreglos rellenos con sus valores correspondientes se pasa a rellenar el arreglo de índices que va a indicar la forma en que se recorren estos arreglos. Para el dibujado del terreno se va a utilizar la primitiva QUAD_STRIP, por lo tanto el arreglo de índices de los vértices tiene que ser de manera que se dibuje esta primitiva. Con el ancho del terreno y altura del terreno se logra esta organización.

Finalmente se rellena una matriz que contiene la altura de cada vértice del terreno que se mapea de a acuerdo a las coordenadas X y Z de cada vértice. Esta matriz va a ser de utilidad a la hora de sacar la altura del terreno según unas coordenadas en el espacio de nuestra aplicación, en particular para sacar la altura del terreno a partir de la posición de la nave.

Al tener todos los arreglos en memoria del sistema es hora de activar los estados de cliente de los arreglos de vértices, normales y texturas. También se indican la localización e organización de estos arreglos. El siguiente es el código que realiza estos pasos:

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glVertexPointer(3, GL_FLOAT , 0, verticesTerreno);
glNormalPointer(GL_FLOAT, 0, normalesTerreno);
```

```
glTexCoordPointer(2, GL_FLOAT, 0, texturasTerreno);
```

Luego se genera un id para la *display list* que va a guardar los comandos que se van a ocupar de dibujar el terreno. En este fragmento de código se puede observar como se llama al comando *DrawElements* con el modo de QUAD_STRIP como ya se había mencionado antes. También se utiliza el arreglo de índices antes mencionado para indicar como dibujar el quad strip.

```
displayListID = glGenLists(1);
 glNewList(displayListID, GL_COMPILE);
 for(int i = 0; i < cantQuadStrips; i++)
{
        glDrawElements(GL_QUAD_STRIP, anchoTerreno * 2,
        GL_UNSIGNED_INT, &indicesTerreno[anchoTerreno*2*I ]);
 }
glEndList();
```

Notar que al usar quad strips como primitiva se hacen un total de llamadas a *DrawElements* igual al largo del terreno -1. Si se utilizarían simplemente quads habría un gran impacto en la performance, ya que en ese caso se harían un total de llamadas igual a (largo del terreno -1) x (ancho del terreno – 1). El id de la *display list* creada se guarda en un atributo del objeto terreno ya que esta lista será referenciada siempre en el momento de renderizado de la escena.

Luego llegamos a un problema cuando nos pusimos a probar el juego, el terreno no era tan grande como queríamos para nuestra aplicación. Una solución fue aumentar el tamaño del terreno aumentando los lados de cada quad de la geometría. Pero esta solución tenía unos problemas, entre ellos el terreno quedaba más plano, pero también uno más grave es que la precisión de las colisiones se veía afectada. Al ser mas grandes los quads las alturas del terreno que se dan en los vértices quedan más imprecisas a la hora de compararlas con la altura de la nave. Entonces como solución para agrandar el terreno lo que hicimos fue replicar el terreno varias veces en todas direcciones, por esta razón es que era importante que la imagen del mapa de alturas fuera tileable ya que sino en la unión de terrenos las alturas estarían desparejas.

4.5 SKYBOX

Se trata de una técnica donde encierras la escena dentro de un cubo al cual se le mapea una textura especialmente diseñada para dar la sensación de cielo. Esta skybox se dibuja siempre antes de toda la otra geometría en la escena con el buffer de profundidad desactivado.

En nuestra aplicación al principio cargamos las 6 texturas del skybox, una textura por cada cara del cubo. Luego un método se encarga de dibujar este skybox. Su dibujado es sencillo, se deshabilitan el test de profundidad, la iluminación (no queremos que afecte las texturas del skybox), y también la transparencia. En el siguiente código se puede observar bien:

```
glPushAttrib(GL_ENABLE_BIT);
glDisable(GL_DEPTH_TEST);
glDisable(GL_LIGHTING);
glDisable(GL_BLEND);
```

El primer comando guarda en un stack especial el estado de los bits de enable, luego para volver al estado anterior se hace un pop del mismo atributo.

También para mantener el skybox de acuerdo a las rotaciones de la cámara, antes de dibujarla le aplicamos las mismas rotaciones que a la cámara.

Luego se dibuja el cubo utilizando la primitiva QUAD para cada cara y llamando a la textura correspondiente a cada una de ellas. También estos comandos se guardan en una *display list* ya que su dibujado es siempre el mismo.

4.6 MODELOS 3D

Como es posible imaginar, el proceso de especificar la posición y otros atributos para cada vértice en un objeto no es una tarea escalable. Un cubo, una pirámide o una superficie tildada no involucran demasiada complejidad. Sin embargo, cuando se trata de un rostro humano, o una nave espacial, no es una tarea simple. En el mundo real de los juegos y las aplicaciones comerciales, el proceso de creación de mallas es realizada por artistas que utilizan programas de modelado como ser Blender, Maya y 3ds Max. Estas aplicaciones proveen herramientas avanzadas que ayudan al artista a crear modelos extremadamente sofisticados. Cuando el modelo esta complete, se guarda en un archivo en uno

de muchos formatos. El archive contiene la definición geométrica completa del modelo. Puede ser luego cargado luego en el juego y su contenido ser usado para llenar los buffers de vértices e índices para renderizado.

Saber como parsear la definición geométrica y cargar modelos profesionales es crucial para llevar nuestra programación 3d al siguiente nivel.

Desarrollar el parser por nosotros mismos podría requerir demasiado tiempo. Si quisiéramos cargar modelos de distintas fuentes deberíamos estudiar cada formato y desarrollar un parser específico. Algunos de los formatos son simples pero otros demasiado complejos y acabaríamos desperdiciando demasiado tiempo en algo que no es exactamente programación 3d. Es por ello que nuestra aproximación será utilizar una librería externa que se encargue de parsear y cargar modelos de archivos.

La "*Open Asset Import Library*" o *Assimp*, es una librería de código abierto que maneja muchos formatos 3d, incluyendo los más populares. Es portable y disponible para Linux y Windows.

La clase ModelManager representa la interfaz entre Assimp y nuestro programa. ModelManager es una clase Singleton. La única instancia de ModelManager toma un nombre de archivo y una clave, usa Assimp para cargar el modelo y almacena en un diccionario interno un objeto Modelo. Al crear un objeto Modelo, ModelManager le asigna el objeto aiScene, que Assimp usa para representar el modelo cargada. El objeto aiScene es el punto raíz de donde acceder todos los distintos data types que un archivo de modelo puede contener. En [7] se describe cómo interpretar estos datos.

Dentro del aiScene tenemos pequeñas entidades llamadas nodos "*aiNode*", que tienen un lugar y una orientación relativa a sus padres. Empezando por el nodo raíz de la escena, todos los nodos pueden tener de 0 a x nodos hijos, formando así, una jerarquía. Los nodos forman la base sobre la cual se construye la escena.

Un nodo puede referirse a 0..x mallas. Las mallas no se guardan dentro del nodo, sino que se guardan en un array de "*aiMesh*" ("*mMeshes*") dentro del aiScene. Un nodo solo se refiere a sus mayas por su índice en el array. Esto significa que múltiples nodos pueden referirse a la misma malla.

Una malla usa un solo material en todos lados. Si partes del modelo usan un material diferente, estas partes son movidas a una malla separada en el mismo nodo. La malla se refiere a su material de la misma forma que el nodo se

refiere a sus mallas: los materiales se guardan en un array dentro del aiScene. La malla guarda solo el índice en ese array.

Un aiMesh se define con una serie de canales de datos. La presencia de esos canales de datos se define por el contenido del archivo importado. Por defecto hay solo aquellos canales de datos presentes en la malla que fueron encontrados también en el archivo importado. Los canales cuya presencia está garantizada son aiMesh::mVertices y aiMesh::mFaces. Testearemos la presencia de cualquier otro canal al que queramos acceder comparando contra null.

Para la resolución del obligatorio se eligieron tres modelos de [8]. Uno para representar al lunar lander, otro para un satélite que acompaña al lunar lander de cerca y otro para representar a las naves enemigas. Ninguno de estos modelos posee texturas, por lo que no se desarrollo el trozo de código correspondiente para cargar texturas.

Observando el código, en primer lugar, para cargar un modelo utilizamos la siguiente función,

```
bool ModelManager::LoadMesh(const char* path, const unsigned int texID) {

   m_texID[texID] = this->loadasset(path);

}
```

que a su vez llama a loadasset,

```
Modelo* ModelManager::loadasset(const char* path)
{
   const aiScene* scene = aiImportFile(path,…);

   Modelo* nuevo_modelo = new Modelo(scene); // Creamos el objeto Modelo

   …

   GLuint scene_list = glGenLists(1);      // Generamos una display list para renderizar
   glNewList(scene_list, GL_COMPILE);

//Comenzamos en el nodo raíz de los datos importados

   recursive_render(scene, scene->mRootNode);
   glEndList();

   nuevo_modelo->SetDisplayList(scene_list);
```

```
    return nuevo_modelo;
}
```

Luego tenemos el método *recursive_render*,

```
void recursive_render(const struct aiScene *scene, const struct aiNode* nd)
{
  …

  // dibujamos todos los meshes del nodo
  for (; n < nd->mNumMeshes; ++n) {

    const struct aiMesh* mesh = scene->mMeshes[nd->mMeshes[n]];

    // recorremos todas las caras de cada mesh
    for (t = 0; t < mesh->mNumFaces; ++t) {
      const struct aiFace* face = &mesh->mFaces[t];
      GLenum face_mode;

      // Determinamos la primitive a dibujar según el número de índices para el array de
      //vértices

      switch (face->mNumIndices) {
        case 1: face_mode = GL_POINTS;
          break;
        case 2: face_mode = GL_LINES;
          break;
        case 3: face_mode = GL_TRIANGLES;
          break;
        default: face_mode = GL_POLYGON;
          break;
      }

      glBegin(face_mode);

      for (i = 0; i < face->mNumIndices; i++) {
        int index = face->mIndices[i];

       // chequeamos pues no esta garantizado que tengamos colores ni normales
        if (mesh->mColors[0] != NULL)
          glColor4fv((GLfloat*) & mesh->mColors[0][index]);
        if (mesh->mNormals != NULL)
```

```
                glNormal3fv(&mesh->mNormals[index].x);
            glVertex3fv(&mesh->mVertices[index].x);
          }

          glEnd();
        }

    }

    // Recordemos que se trata de una jerarquía de nodos
// por lo que ejecutamos el método recursivamente para cada nodo hijo.

    for (n = 0; n < nd->mNumChildren; ++n) {
        recursive_render(scene, nd->mChildren[n]);
    }

  …
}
```

En [6] puede encontrarse un código de ejemplo en el cuál se baso nuestra implementación. Entre otras cosas no mencionadas, se determina un factor de escalamiento del modelo y sus dimensiones.

4.7 LUNAR LANDER

El renderizado del lunar lander junto con sus variables, conforman la esencia de la aplicación. Es por ello, que se decidió mantener los datos dentro de SpaceRaceCore. Para coordinar su movimiento comenzaremos por comentar las variables utilizadas.

Se mantienen tres vectores que representan su posición, su velocidad y su aceleración en *x*, *y* y *z* en cada instante.

```
Vector3f pos_lander;
Vector3f vel_lander;
Vector3f acl_lander;
```

Además, es posible rotar el lunar lander tanto de acuerdo al eje (1,0,0) como al eje (0,0,1). Por lo que se tienen variables que mantengan este movimiento:

```
GLfloat xrot_lander,yrot_lander;
GLfloat maxxrot_lander, maxyrot_lander;
```

Nuestro objetivo es determinar la próxima ubicación del lunar lander en cada iteración. Por tanto, surge de forma inmediata la necesidad de saber cuáles son las fuerzas que actúan sobre el lunar lander. En la implementación llevada a cabo por el grupo, solamente se tienen en cuenta dos fuerzas, la fuerza peso y la fuerza que se imprime sobre el lunar lander al encender el propulsor. La fuerza peso es constante y siempre está presente. Sin embargo, la fuerza causada por el propulsor no. Esto implica que haya una variación de fuerza en el tiempo, y por tanto una variación de la aceleración. Razonamiento trivial si recordamos la ecuación fundamental de la mecánica clásica:

$$\sum \vec{F} = m\,\vec{a}$$

En cada ciclo, las 3 variables, *pos_lander*, *vel_lander* y *acl_lander* serán actualizadas siguiendo las ecuaciones que describen el movimiento con aceleración constante y la segunda ley de newton. Buscando simplificar la realidad y los cálculos, en cada llamado a *OnLoop()* se tendrán dos posibilidades. Que el propulsor este activado o no. Dependiendo del estado del propulsor, se asigna a *acl_lander* el valor correspondiente.

Es muy común encontrar el movimiento con aceleración constante. Los objetos que caen cerca de la superficie terrestre o de los automóviles que frenan son muy representativos. Pero no olvide que las siguientes ecuaciones se aplican sólo cuando la aceleración es constante.

Las ecuaciones que nos permiten analizar el caso especial de cinemática unidimensional con aceleración constante, son las siguientes:

$$v_x = v_{0x} + a_x t$$

$$x = x_0 + v_{0x} t + \frac{1}{2} a_{0x} t^2$$

Pero en lugar de estas ecuaciones necesitamos ecuaciones para movimiento en tres dimensiones. En este caso tenemos:

$$\vec{v} = \vec{v_0} + \vec{a}\,t$$

$$\vec{r} = \vec{r_0} + \vec{v_0} t + \frac{1}{2} \vec{a}\, t^2$$

La segunda ley de newton en forma vectorial abarca las tres ecuaciones de las componentes:

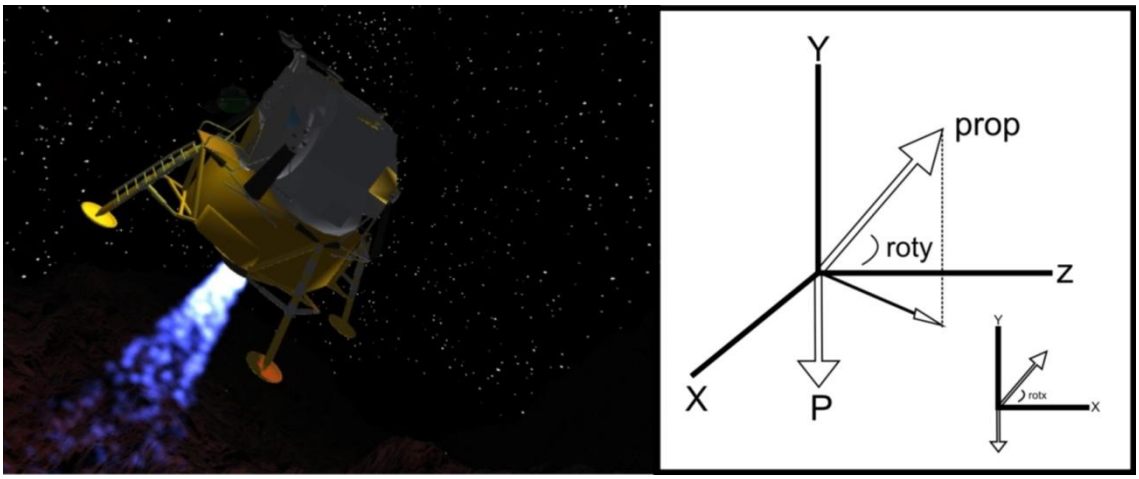
$$\sum F_x = m\, a_x$$

$$\sum F_y = m\, a_y$$

$$\sum F_z = m\, a_z$$

Es necesario satisfacer las tres simultáneamente cuando apliquemos la segunda ley.

El módulo lunar (L.E.M.) **era un vehículo espacial construido de aluminio** y de unos 6.98 metros de altura por 9.45 de ancho. Su peso era aproximadamente de 15,061 KG. Supondremos esa masa para el lunar lander. La gravedad en la luna es de aproximadamente 1,62 m/s^2. Adoptaremos ese número para la g de la luna. Al encender el propulsor se generará una fuerza de magnitud igual a 2 veces la magnitud de la fuerza peso. Se supondrá que esta fuerza opera sobre el centro de masa del lunar lander y que se encuentra rotada según la rotación del lunar lander.



Tenemos por tanto, los siguientes cálculos en el método *OnLoop*:

```
if(propulsor) // propulsor activado
{
        // calculamos las componentes de la fuerza
        // Debemos convertir los ángulos a radianes
        Fx = fuerzaPropulsor*(sinf(xrot_lander*M_PI/180));
```

```
            Fz = -fuerzaPropulsor*(sinf(yrot_lander*M_PI/180));
            Fy = sqrt(fuerzaPropulsor*fuerzaPropulsor - (Fx*Fx) - (Fz*Fz));

            // Aceleración calculada a partir de la segunda ley
            acl_lander.Set(Fx/masaLunarLander,(-fuerzaPeso + Fy)/masaLunarLander,
            Fz/masaLunarLander);

            // posición calculada según las ecuaciones de aceleración constante
            pos_lander+=vel_lander*(timeElapsed)+
            acl_lander*(0.5*timeElapsed*timeElapsed);

            // velocidad calculada según las ecuaciones de aceleración constante
            vel_lander += acl_lander*(timeElapsed);
        }
        else
        {
            acl_lander.Set(0,-gLuna,0); // solo fuerza de gravedad


            pos_lander+=vel_lander*timeElapsed+
            acl_lander*(0.5*timeElapsed*timeElapsed);
            vel_lander += acl_lander*timeElapsed;
        }
```

A pesar de haber simplificado bastante la realidad, el movimiento obtenido es bastante realista. Posteriormente se añadieron detalles como ser un turbo, cuyo efecto es incrementar la fuerza del propulsor aplicada al lunar lander.

Al momento de renderizar el lunar lander, se renderizan 2 luces frontales rojas, 1 luz trasera amarilla y una luz azul que simula ser la iluminación causada por el propulsor. Esta última, se enciende al presionar el propulsor. Todas son posicionales y para cada una de ellas se configuran los valores GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_POSITION, GL_SPOT_DIRECTION, GL_SPOT_CUTOFF y GL_SPOT_EXPONENT.

4.8 SISTEMAS DE PARTÍCULAS

Para simular la mayoría de los efectos del juego, se investigaron varias técnicas posibles. Una primera aproximación a lo que finalmente se utilizó en la implementación de los sistemas de partículas del videojuego, es el material disponible en [9]. En dicha fuente puede encontrarse información y código y

código fuente para el renderizado de fuego y humo en tiempo real. La técnica utilizada es bastante compleja y extremadamente realista. Su implementación hubiera desviado nuestra atención de aspectos más importantes a tener en cuente en el obligatorio. Es por ello, que preferimos continuar en la búsqueda de una solución más simple y en conformidad con nuestros objetivos.

Una técnica bastante distinta, desarrollada en [11] consiste en pensar en cada partícula como un objeto individual que responde al entorno que la rodea. A cada partícula se le otorga vida, envejecimiento aleatorio, color, velocidad, aceleración, y más. De esta forma, es posible recrear escenas complejas con miles de partículas controladas por las ecuaciones de movimiento, conformando un efecto particular como ser fuego, humo, polvo, o lo que se nos ocurra.

Observemos la estructura de una partícula,

```
typedef struct {
   int active;        /* Si se encuentra activa o no */
   float life;        /* Vida de la partícula   */
   float fade;        /* Velocidad de desaparición      */
   float dt;          /*  Tiempo transcurrido desde su creación */

  /* Color      */
   float r; float g; float b;

/* Posicion       */
   float x; float y; float z;

/* Velocidad      */
   float xi; float yi; float zi;

/* Aceleración      */
   float xg; float yg; float zg;

} Particle;
```

Sin entrar en más detalles describiremos la implementación del sistema de partículas que da vida al Propulsor del lunar lander. Dicha implementación nos permitirá comprender cuáles son los factores a tener en cuenta al momento de diseñar un sistema de partículas similar a los desarrollados.

En primer lugar, cada instancia de Propulsor posee un array de partículas:

```
Particle particles[MAX_PARTICLES];
```

Al momento de renderizar el propulsor se recorre cada una de las partículas previamente inicializadas. Para lograr un efecto de transparencia debe previamente ejecutarse,

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_ONE);
```

Luego, como estamos usando GL_BLEND, las cosas deben dibujarse de atrás hacia adelante. Es por ello que nos definimos un vector

```
vector<Particle*> ps;
```

y ordenamos las partículas según la posición sobre el eje z.

```
for(int i = 0; i < MAX_PARTICLES; i++) {
  ps.push_back(particles + i);
}
sort(ps.begin(), ps.end(), compareParticles);
```

Luego recorremos cada una de las partículas

```
for(unsigned int i = 0; i < ps.size(); i++) {
        Particle* p = ps[i];
if (p->life > 0) { // si su vida es mayor que cero

   // Otorgamos color a la partícula y según su vida, la transparencia.
   // A medida que decrece su vida, aumenta la transparencia, lo que brinda un
   // efecto de desaparición
   glColor4f(p->r * (p->life), p->g * (p->life), p->b * p->life * 3, p->life);

        // Dibujamos la particular
   glBegin(GL_TRIANGLE_STRIP);
        …
   glEnd();


  // Calculamos su nueva posición  y velocidad según las ecuaciones de movimiento con
  // aceleración constante

   p->x += p->xi * p->dt + p->xg * (0.5f) * p->dt * p->dt;
```

```
    p->y += p->yi * p->dt + p->yg * (0.5f) * p->dt * p->dt;
    p->z += p->zi * p->dt + p->zg * (0.5f) * p->dt * p->dt;

    p->xi += p->xg * p->dt;
    p->yi += p->yg * p->dt;
    p->zi += p->zg * p->dt;

    p->yg -= p->yg * (fabs(p->y) / 10); // gravedad


    /* Reducimos la vida de la particular según la velocidad de envejecimiento */
    p->life -= p->fade;

   // Incrementamos el tiempo que transcurrió desde su creación.
    p->dt += timeElapsed;
  // Si la particular murió,  la volvemos a crear */
    if (p->life <= 0.0f){

       p->life = 1.0f;
       p->fade = (float) (rand() % 100) / 1000.0f + 0.01f;
       p->dt = 0;

       p->x = ((float) (rand() % 500)) / 7000.0f;
       p->y = ((float) (rand() % 500)) / 7000.0f;
       p->z = ((float) (rand() % 500)) / 7000.0f;

       p->xi = 0;
       p->yi = 0;
       p->zi = 0;

       p->xg = (p->x - 0.035)*3000;
       p->yg = -1000.0f + 0.01 * fabs(p->xg) ;
       p->zg = (p->z -  0.035)*3000;

       }
     }
  }
```

Los parámetros tenidos en cuenta para determinar la aceleración de la partícula son muchos. En el caso del propulsor nos propusimos lograr un efecto de potencia y fuerza, al estilo de un cohete. Es por ello que la aceleración en Y debe ser muy grande y en sentido negativo. A su vez, la aceleración depende de la posición de la partícula. Suponiendo que solo viéramos el fuego de frente,

quisiéramos que las partículas que se encuentran en los extremos se abran, formando un abanico.

No entraremos en detalles sobre el desarrollo del polvo lunar y el fuego. Pero es importante señalar que su implementación se basa en los mismos principios presentados, variando la asignación de la velocidad y aceleración de las partículas para lograr el efecto deseado.

4.9 COLISIONES

4.9.1 Colision entre el terreno y la nave

Cuando se carga el modelo 3D de la nave se guardan las medidas de la misma, como la distancia desde el centro de la nave hacia lo más bajo de la nave o hacia los costados de la nave. Estas medidas se obtienen al hacer la caja que la recubre (bounding box). Con estas distancias y los ángulos de rotación de la nave calculamos la colisión con el terreno.

Para la nave de nuestra escena tenemos su posición que corresponde al centro de la misma. A partir de las coordenadas X y Z de esta posición podemos mapear la nave en el terreno. Para determinar si hay colisión hallamos las 5 alturas correspondientes con las 4 patas de aterrizaje de la nave y el centro también. Además con las distancias a los bordes de la nave y con los ángulos de rotación de la nave se calculan de forma más precisa estas alturas. Luego estas alturas se comparan con las alturas del terreno correspondientes según las coordenadas X y Z de cada pata de la nave y su centro. Si alguna de estas alturas del terreno es mayor o igual a su correspondiente altura de la nave entonces sabemos que hay colisión.

Al saber que existe colisión se desea saber si es un aterrizaje con éxito o fallido. Para calcular esto vemos que los ángulos de rotación de la nave no sean muy grandes, deben ser próximos a 0 para que la nave aterrice derecha. Además se controla que las velocidades en X, Y, Z no sean muy altas (próximas a 0). Como último requerimiento para aterrizaje exitoso es comparar que las alturas halladas previamente sean más o menos iguales, de esta forma nos aseguramos que la nave esta aterrizando en una de las zonas de aterrizaje colocadas en el terreno. Si no se cumple alguno de estos requerimientos entonces es un aterrizaje fallido.

4.9.2 Colisión con las naves enemigas

Las naves enemigas son 3. Se posicionan en forma aleatoria cerca del lunar lander con velocidad aleatoria. Cuando se encuentran demasiado lejos, son reubicadas cerca del lunar lander. Cada una de ellas posee una ubicación por lo que en cada iteración se comprueba la distancia entre las naves enemigas y el lunar lander. Si se detecta que esta distancia es menor a 4, entonces se dice que hay una colisión. Al lunar lander se le imprime la velocidad de la nave enemiga con la que colisionó y es expulsado al suelo.

Esta parece una forma demasiado trivial para resolver la colisión, sin embargo, los resultados obtenidos son aceptables.

4.10 GAME HUD

Para el renderizado de texto del game hud y del menú utilizamos una clase (extraída de un tutorial online) que se encarga de crear un tipo de fuente de texto que nosotros elijamos, también se encarga de limpiar esta fuente de la memoria del sistema, y lo más importante proporciona un método para imprimir en pantalla en la posición que se requiera el texto de una manera muy simple.

Cuando se crea la fuente hay que pasarle el nombre del archivo donde se guarda la fuente a utilizar, este archivo debe ser un archivo de fuente TrueType (.ttf). Además se le pasa la altura deseada de la fuente a crear. Esta función utiliza funciones de la librería *freetype* para poder leer los datos de este archivo. Se crean *display lists* para cada carácter de la fuente a utilizar, además se guardan los identificadores de estas listas y las texturas para poder utilizarlas a la hora de dibujar en pantalla. Este es el encabezado del método:

*void init(const char * fname, unsigned int h);*

La función que usamos para dibujar el texto en pantalla es la siguiente:

*void print(const font_data &ft_font, float x, float y, const char *fmt, ...) ;*

Los parámetros *x, y* corresponden a la posición en pantalla dada en pixeles, y el primer argumento corresponde a la fuente a usar. Esta función permite la impresión con formato, al igual que la función printf de C.

Lo primero que se realiza en esta función es pasar a una vista ortogonal en 2D de la pantalla. Para realizar este cambio se ejecuta la siguiente funcion:

```
inline void pushScreenCoordinateMatrix() {
        glPushAttrib(GL_TRANSFORM_BIT);
        GLint   viewport[4];
        glGetIntegerv(GL_VIEWPORT, viewport);
        glMatrixMode(GL_PROJECTION);
        glPushMatrix();
        glLoadIdentity();
        gluOrtho2D(viewport[0],viewport[2],viewport[1],viewport[3]);
        glPopAttrib();
}
```

Lo primero que hace es guardarse el estado del bit de transformación para luego con un simple pop retornarlo al estado anterior. Luego se hace un get de el *viewport*. Este get retorna un arreglo de 4 GLint teniendo en la dos primeras posiciones el x, y del *viewport* que se inicializan a 0, 0. En las siguientes posiciones están el ancho y alto de la pantalla. A continuación se cambia a la matriz de proyección ya que hay que cambiar de vista en perspectiva a vista ortogonal. Se llama al comando especial de glu para pasar a vista ortogonal en 2D, la especificación del mismo es:

```
void gluOrtho2D( GLdouble left, GLdouble right, GLdouble bottom, GLdouble top );
```

Donde left, right, bottom, top corresponden a las distancias al plano izquierdo, derecho, de abajo y de arriba respectivamente. Por lo tanto en la llamada a este comando se asigna que las distancias al plano izquierdo, derecho, de abajo y de arriba son 0, ancho de la pantalla, 0 y alto de la pantalla respectivamente. De esta forma el mundo de las coordenadas de nuestros objetos quedan idénticas a las coordenadas de la pantalla, haciendo mucho más fácil el dibujado del texto en la posición que se requiera de la pantalla.

Para volver a la vista en perspectiva que se tenía antes se utiliza la siguiente función:

```
inline void pop_projection_matrix() {
        glPushAttrib(GL_TRANSFORM_BIT);
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glPopAttrib();
}
```

Simplemente pone el modo de matriz en proyección y hace pop de la matriz previamente guardada antes de pasar a vista ortogonal en 2D.

El resto de la función se encarga de imprimir el texto con formato pasado en el tercer parámetro, utilizando los identificadores de la display list y texturas guardadas en el objeto con todos los datos de la fuente.

Todo el dibujado del hud y de los diferentes menús se realizan luego de dibujar toda la escena del juego. En el hud se muestra información que le ayuda al jugador, como la velocidad de la nave en sus tres ejes en cada momento, además de la altura al terreno, el combustible disponible y si se puede utilizar el turbo de la nave. Otros datos son de información como el nivel que se encuentra y el puntaje que tiene. Luego están los datos sobre los controles del juego, como poner pausa, ir al menú, salir del juego, luces de la nave, controles sobre la nave.

En el menú del juego se tiene la opción de un nuevo juego reiniciando el puntaje y empezando del nivel 1. Luego está la opción de ajustes del juego que se verá en la siguiente sección. También una opción para ver los créditos del juego y una opción que permite salir del juego desde el menú. El manejado de los menús se realizo utilizando variables que indican en que opción está parado el jugador y en que menú. Luego mediante eventos se controla los movimientos y realizados por los menús del jugador.

4.11 SETTINGS

Para el manejo de settings del juego utilizamos varias variables que nos indican el estado de cada setting del juego. Tenemos para la velocidad del juego, modo wireframe, facetado/interpolado, texturas, iluminación, color de la luz que ofrece el satélite y posición de esta luz, y una más que nos indica el tipo de terreno (plano, normal o montañoso).

El usuario puede cambiar todas estas opciones desde el menú de ajustes, y desde la aplicación se cambian las variables correspondientes. De acuerdo a estas variables al momento de dibujar la escena se habilitan/deshabilitan/cambian los diferentes estados de OpenGL para cumplir con lo pedido por el usuario.

5 CONCLUSIÓN

Para finalizar el informe sobre la realización del obligatorio número uno, es importante mencionar los objetivos cumplidos.

El proceso hacia la formalización, ha tenido como propósito, según lo mencionado en ocasiones anteriores, ser coherentes en el desarrollo del informe, a diferencia de las ocasionales contradicciones que pueden encontrarse en piezas de información varias. Se pretendió fijar la atención en aspectos importantes y esenciales para la comprensión del desarrollo.

Durante la preparación del trabajo los cambios y variantes, tanto de detalles como de perspectiva se hicieron numerosos y de múltiples estratos, sin embargo, en busca de una lectura confiable y coherente, se basaron las notas, observaciones y conclusiones en los textos bibliográficos.

Antes de la resolución del obligatorio, se intentó introducir al lector al aspecto teórico, y definir en forma precisa el vocabulario utilizado en etapas posteriores. Se expusieron imágenes a modo de esclarecer la situación y agilizar la lectura.

Debemos mencionar que las decisiones tomadas a lo largo de la implementación surgieron como respuesta a los resultados obtenidos hasta el momento.

6 BIBLIOGRAFÍA

[1] Dave Shreiner – Mason Woo – Jackie Neider – Tom Davis
OpenGL programming guide. The official guide to learning OpenGL Version 2.1. 6th edition (2007).

[2] The OpenGL Graphics System: A Specification (Version 1.5)

[3] The OpenGL Graphics System Utility Library (Version 1.3)

[4] http://nehe.gamedev.net/tutorial/ (Codigo de FreeType extraido del tutorial 43 - FreeType Fonts in OpenGL)

[5] http://www.sdltutorials.com/sdl-events

[6] http://assimp.sourceforge.net/main_doc.html

[7] http://assimp.sourceforge.net/lib_html/data.html

[8] http://www.celestiamotherlode.net/

[9] http://graphics.ethz.ch/teaching/former/imagesynthesis_06/miniprojects/p3/index.html

[10] http://ogldev.atspace.co.uk/www/tutorial22/tutorial22.html

[11] nehe.gamedev.net/tutorial/particle_engine_using_triangle_strips/21001/

[12] http://www.oogtech.org/content/tag/gllightfv/

[13] http://www.fing.edu.uy/inco/cursos/compgraf/Clases/2012/OpenGL2012_1.pdf

[14] http://netbeans.org/

[15]http://netbeans.org/community/releases/69/cpp-setup-instructions.html#compilers_windows