Introducción a la Computación Gráfica

Obligatorio 1

Videojuego en OpenGL - Crossy Road

Integrantes

Nombre	CI	Correo
Emiliano Botti	4.670.479-1	emiliano.botti@fing.edu.uy
Renzo Gambone	5.155.486-4	renzo.gambone@fing.edu.uy
Germán Ouviña	4.823.566-1	german.ouvina@fing.edu.uy

Índice

1. Introducción

En el presente informe se tratan los aspectos generales del proceso de desarrollo del **Obligatorio 1**, utilizando la plantilla otorgada por el cuerpo docente como base estructural.

En la presente sección, se aborda el problema como tal y el análisis realizado previo a la construcción de la solución.

En la sección **Desarrollo** 2, se detalla tanto la arquitectura de la solución, como los aspectos técnicos de desarrollo e implementación, poniendo especial énfasis en los puntos relacionados al manejo de gráficos y de OpenGL. En un segundo plano, se tratan la dinámica del juego y otros aspectos generales.

En la sección **Conclusiones** 3, se realiza un análisis posterior a la construcción de la solución, repasando conceptos aprendidos y puntos donde hubo dificultades.

Finalmente, en la sección **Trabajo futuro** 4, se mencionan todos los puntos que quedaron pendientes en el proceso de desarrollo y no fueron realizados por limitaciones de tiempo.

1.1. Definición del problema

La solución debe emular el comportamiento del juego Crossy Road, el cual consiste en un jugador que avanza por un escenario esquivando distintos tipos de obstáculos. El juego cuenta con múltiples dinámicas, pero se hace énfasis en las funcionalidades básicas:

1. Manejo del jugador.
2. Manejo del escenario (y generación dinámica del mismo).
3. Interacción entre jugador y escenario.
4. Manejo de progreso dentro del juego.
5. Computación gráfica de todo lo anterior.

En las siguientes secciones se baja a tierra el desarrollo de cada requisito, no entrando ahora en detalle sobre su definición, ya que esta es tratada en la consigna del obligatorio.

1.2. Análisis del problema

El problema en cuestión incluye múltiples aristas, ya que es necesario modelar tanto el comportamiento y la lógica del juego, así como la generación de gráficos 3D para el mismo. A su vez, se agregan ciertos requisitos relacionados a la parte gráfica en cuestión, los cuales no están relacionados con el funcionamiento del juego. Esto se hace con el motivo de poder analizar las implementaciones gráficas en tiempo de ejecución y también ha de ser tenido en cuenta a la hora de desarrollar la solución.

De esta manera, se decidió encarar el problema de manera iterativa, generando una arquitectura base lo más modularizada posible y construyendo a partir de ahí. Inicialmente, se intentó separar la lógica gráfica de la lógica del juego. No obstante, en diversos escenarios resultó de utilidad mantener cierta relación entre ambas partes, por lo que en la versión final se cuenta con componentes tanto de un área como de ambas, manteniendo la visibilidad cruzada al mínimo.

2. Desarrollo

2.1. Arquitectura

Teniendo en mente la separación de computación gráfica y lógica del juego, se separó el código en diversos módulos, asignando distintas responsabilidades a cada uno y estableciendo un patrón de conexión entre ellos. De esta manera, los módulos principales son los siguientes:

- `Game` - El módulo principal, donde se implementa el bucle de juego. Dicho módulo es encargado de inicializar, centralizar y ejecutar al resto de módulos. El código que ejecuta puede ser dividido en la siguiente etapas:
  1. **Inicialización:** Como primera etapa se inicializan todas las variables, flags, módulos y objetos necesarios para ejecutar el juego y el componente gráfico.
  2. **Bucle principal:** Se ejecuta infinitamente hasta que el usuario decida salir del juego. Dentro del mismo, se ejecutan las siguientes etapas en orden para cada iteración:
     a) **Pre renderizado de componentes:** Limpieza de buffers y matrices de OpenGL.
     b) **Cálculo del tiempo:** Determinar el tiempo transcurrido entre iteración e iteración, utilizandose dicho valor para todas aquellas acciones que se ejecuten de manera continua y no discreta a través del tiempo.
     c) **Manejo de eventos:** Detección de eventos de teclado y mouse, delegando a distintos componentes la decisión de que hacer en base al evento en cuestión.
     d) **Actualización de componentes:** Actualización de cada componente y sus valores internos en base a las acciones del usuario, los eventos disparados y el tiempo transcurrido.
     e) **Renderizado de componentes:** Manejo de la luz y del renderizado de cada modelo del programa.
     f) **Post renderizado de componentes:** Swap de la ventana.
  3. **Limpieza final:** Destrucción de objetos residuales y otros aspectos de interfaz gráfica.
- `Models` - Módulo donde se definen representaciones de los objetos del juego. Si bien se expanden en la sección 2.2.3, por lo pronto se puede mencionar que existen tres modelos principales para los objetos del juego:
  - `Player` - Objeto de instanciación única, representando al jugador. Pertenece al módulo `Game`.
  - `Lane` - Objeto de instanciación múltiple, representando un único carril del escenario, con el mismo ancho que el jugador. Pertenece al módulo `GameManager`.
  - `ScenarioObject` - Objeto de instanciación múltiple, representando un único objeto del escenario, dentro de un carril dado. Pertenece al modelo `Lane`.

  Se destaca que todos los modelos cuentan con métodos `draw` y `update` para su renderizado y actualización en el bucle principal.
- `Managers` - Módulo donde se definen distintos manejadores en base a diversas responsabilidades:
  - `GameManager` - Encapsula el control de la lógica del juego, teniendo responsabilidad sobre la generación dinámica de `lanes`, el seguimiento del avance del `player` (guardándolo en `score` y `coins`) y el escalamiento en dificultad a medida que `score` crece (implementando una variable `level` que es utilizada para hacer más difícil el juego a medida que crece).
  - `CameraManager` - Encapsula el control de los distintos tipos de cámara, sus posiciones, ángulos y distancias.
  - `DrawManager` - Encapsula el manejo del renderizado en pantalla de diferentes figuras del juego. Esto permite generar una única instancia para carga de texturas, y centralizar métodos genéricos de dibujo (por ejemplo, `DrawMultiplePoints`).

  - `UIManager` - Encapsula el manejo de las operaciones de interfaz tales como el `HUD`, su dibujado y posterior limpieza.
  - `FrustumManager` - Centraliza el uso del *Frustum Culling* (actualmente desactivado debido a errores).

  Se destaca que todos los manejadores son de única instanciación, es decir, implementan el patrón de diseño *singleton* y por lo tanto son de acceso global siempre y cuando el módulo incluya las dependencias necesarias.

- `helpers` - Módulo donde se encapsulan diversos métodos de funcionalidad específica y utilizados en diversas áreas del programa, siendo los mismos:
  - `FileHelper` - Responsable de la carga de texturas (`load_texture`) y carga de modelos 3D (`load_obj`).
  - `MathHelper` - Responsable de encapsular cálculos matemáticos a usar en el resto de la aplicación, como por ejemeplo, pasar de grados a radianes (`degree_to_radian`), u obtener un número aleatorio en un rango (`get_random(limit)`.
  - `WindowHelper` - Responsable de encapsular métodos de control sobre la ventana, tales como incializar SDL y cambiar a pantalla completa.
- `structures` - Módulo similar al de modelos, pero donde se definen estructuras auxiliares de uso extendido por toda la aplicación. Se expande su descripción en 2.2.3.
- `constants` - Módulo de determinación de constantes, las cuales son de uso extendido por toda la aplicación y buscan sustituir un comportamiento configurable. Se expande su descripción en 2.3.1.

2.2. Implementación

2.2.1. Entorno de desarrollo

- **Sistema Operativo:** Windows 10 x64
- **Lenguaje:** C++11
- **IDE:** Visual Studio C++
- **Compilador (Release):** `/O2` (Maximum Optimization (Favor Speed)), `/Oi` (Enable Intrinsic Fucntions) `/GL` (Whole Program Optimization)

2.2.2. Bibliotecas utilizadas

Principalmente, se emplearon las siguientes bibliotecas:

- **OpenGL 1.3 x86:** Empleada como biblioteca principal para el manejo de gráficos.
- **GLU x86:** Empleada en adición a OpenGL para efectuar el render.
- **FreeImage:** Empleada para leer y manipular las imágenes que se emplean para las texturas).
- **SDL2-2.0.12 x86:** Empleada para la creación de la ventana de la aplicación, así como la lectura de dispositivos de entrada.
- **SDL2_ttf-2.0.15 x86:** Empleada para la generación de texturas a través de texto durante tiempo de ejecución (a modo de mostrar el puntaje y las monedas en el HUD).

A su vez, gran parte de las funcionalidades desarrolladas usaron como referencia de implementaciones de terceros. Dichas funcionalidades fueron adaptadas a la arquitectura y contexto de la solución, no obstante, resulta de interés mencionarlas:

- **Proyección Isométrica:** Para la proyección en el tipo de cámara isométrica. [1]
- **Cámara Tercera Persona:** Para el control de los atributos de la cámara en tercera persona. [2]
- **Cámara Libre:** Para el control de los atributos de la cámara en primera persona. [3]
- **Colisiones AABB - AABB:** Para la detección de colisiones entre el jugador y los objetos de la escena. [5]
- **Carga de objetos 3D de archivos .obj:** Para la implementación del parser modelos `.obj` [7] y [8]

Por último, también se utilizaron fragmentos de código de terceros de manera directa, siendo los mismos:

- **FrustumGeometric:** Algoritmo de *Frustum Culling*, el cual fue integrado pero no funciona correctamente. Se mantuvo en la versión final de la solución, pero su uso está desactivado. [4]
- **Plane:** Clase que define un plano en el espacio tridimensional e implementa métodos de operaciones entre vectores y un plano. Es usado únicamente por `FrustumGeometric`. [4]
- **Vector3:** Clase que define un vector en el espacio tridimensional e implementa métodos de operaciones vectoriales. Es usada a lo largo de todo el código cuando se trabajan con posiciones, así como por `FrustumGeometric`. [4]

2.2.3. Estructuras de datos

A continuación, se detallan las estructuras de datos utilizadas en la solución. Se destaca que se separan en dos grupos según lo que representan:

Modelos:

- **Player**
  Modelo que representa al jugador dentro de la lógica del juego. Expone métodos para controlar la dirección del movimiento, y se encarga de actualizar su posición y ángulo en un estado interno. Se destaca que está implementado como un *singleton* ya que jamás se tendrá más de una instancia del mismo.
- **Lane**
  Modelo que representa un carril unitario dentro del escenario del juego. Dicho escenario se compone de un número variable de `lanes`, las cuales van agregandose al final del escenario a medida que el jugador avanza. Existen varios tipos de `lanes`, las cuales son representadas por modelos que heredan de la clase padre:
  - **Grass**: Representa una zona donde el jugador se encuentra fuera de peligro, siendo el tipo de `lane` más básico. En la misma se renderiza pasto de fondo y se generan aleatoriamente árboles, los cuales son obstáculos que tanto decoran el escenario como dificultan el juego al jugador.
  - **Street**: Representa una zona donde el jugador puede chocar con obstáculos móviles, representados por autos, "enemigos" del jugador, estableciendo una zona de peligro. Tiene la particularidad de que dichos autos tienen una tasa de generación aleatoria, con un respectivo retraso predeterminado, utilizado para no superponer autos. El conjunto de autos generados en una `Street` dada comparte la misma velocidad y dirección de desplazamiento, ambas determinadas aleatoriamente. Se destaca que la velocidad y el retraso de generación aumenta y disminuye respectivamente en base al nivel de dificultad al momento de creación de la calle.
  - **Wall**: Usada para delimitar el escenario donde el jugador puede desplazarse, poniendo una muralla por detrás. Al generar nuevas `lanes` y borrar las anteriores, se agrega siempre una `lane` de este tipo en el inicio del escenario, la cual no puede ser atravesada por el jugador (causando una colisión y un comportamiento de rebote).

  Se destaca que todos los tipos de `lane` tienen control sobre los objetos del escenario que se crean en ellas, modelados por `ScenarioObject`. Teniendo esto en cuenta, ellas son las encargadas de generar y controlar dichos objetos. De hecho, implementan un método `update` que retorna las colisiones ocurridas en ese carril.
- **ScenarioObject**
  Modelo que representa los objetos del escenario, siendo los mismos controlados por la `lane` en la que son creados. Existen diversos tipos de objetos, dependiendo de su función y del tipo de `lane` en el que aparecen, siendo los mismos:
  - **Tree**: Los árboles son generados de forma aleatoria y sin repetir posiciones (en el carril `Grass`) y una vez generados no varían su posición dentro de la linea. La cota superior de árboles generados es configurable por una constante, siendo en la última versión de la solución, la mitad del largo de `lane`. Si el jugador colisiona contra ellos "rebota", invirtiendo la dirección de su movimiento para volver a la celda anterior.
  - **Car**: Los autos se generan aleatoriamente durante el tiempo de ejecución (en el carril `Street`), se mueven con la velocidad y dirección que el carril establezca. Si el jugar colisiona con ellos, el juego es terminado. Si un auto sale de los límites del carril, es destruido.
  - **Coin**: las monedas se generan tanto en las calles (`Street`) como en el pasto (`Grass`). Al colisionar con el jugador son "recolectadas", incrementando en uno el contador de `Coins` del juego (situado en el `HUD`).
  - **Border**: Todos los carriles (`Lanes`) cuentan con bordes a los extremos. Son paredes que impiden el paso del jugador (delimitando el movimiento del jugador para que no se salga del escenario).

Estructuras auxiliares:

- **HUDComponent**
  Estructura que representa los componentes dinámicos del HUD. La estructura está destinada a contener atributos para realizar la renderización del componente que representa (como por ejemplo, `Score` que es determinado por la distancia recorrida en el eje `z`, o `Coins` que almacena la cantidad de monedas recolectadas).
- **Vector3**
  Representación de vectores de tres dimensiones. Implementa métodos de operación entre vectores tales como producto interno, producto escalar, entre otros. Originalmente implementado para realizar optimizaciones con `Frustum Culling` [4], y posteriormente extendido a toda la aplicación.
- **Plane**
  Representación de un plano, haciendo uso de `Vector3` para su implementación. Tiene propiedades de plano tales como punto, normal, y métodos como distancia de un punto al plano, entre otros. Originalmente implementado para realizar optimizaciones con `Frustum Culling` [4], y salvo por este mismo, ningún otro componente de la aplicación hace uso de este modelo.

2.2.4. Estados de la aplicación

El manejo de todos estos estados es controlado por el módulo `Game` y el comportamiento del usuario. De esta manera, se definen distintos grupos de estados, de acuerdo a que parte de la aplicación hacen referencia:

Estados del juego:
La lógica del juego en sí cuenta con los estados más basicos.

- **Inicial:** El usuario controla una gallina que se encuentra posicionada en un bosque mirando hacia las carreteras. Inicialmente la cámara se encuentra en tercera persona mirando en la misma dirección que la gallina.

  Usando las fechas del teclado, el usuario puede mover al jugador por el escenario. Si se dirige hacia atrás tiene un límite establecido por una pared, pero por donde puede recolectar monedas si lo desea. Estas monedas están desperdigadas a lo largo del juego, cuya recolección incrementa la variable `Coins` situada en la esquina superior derecha de la pantalla (`HUD`).

  De todas formas, el objetivo del juego es el de avanzar lo más posible. Para ello, el jugador deberá moverse con cautela esquivando los autos que se mueven a diferentes velocidades a través de las carreteras que se despliegan horizontalmente al norte del juego. Cuánto más avance el usuario con su jugador, más puntaje obtendrá (incrementándose la variable `Score` en la esquina superior izquierda del `HUD`).
- **Inmortal:** Tras apretar la tecla `X` el juego entrará **modo inmortal**, desactivando las colisiones del jugador con los autos y por lo tanto, imposibilitando su muerte. Todo el resto del comportamiento del juego se mantiene igual al estado **inicial**.
- **Pausa:** Tras apretar la tecla `P` el juego entrará en pausa, siguiendo en el bucle principal, pero omitiendo las llamadas a los distintos métodos `update` e ignorando los eventos del teclado que desencadenen movimiento del jugador. En este estado de juego, el usuario aún podrá hacer uso de los ajustes del juego si lo desea (por ejemplo, rotación y cambio de cámaras, cambio de texturas, etc.). Para deshacer la pausa basta volver a apretar la tecla `P`.
- **Game Over:** Donde el jugador sea interceptado por un vehículo, se terminará el juego desplegando una pantalla de `Game Over`. La misma tiene el mismo comportamiento que la pausa, sólo que es imposible de remover. Si quiere seguir jugando, deberá cerrar el juego (`Q/ESC`) y volver a ejecutarlo.

Estados de la cámara:
El usuario puede optar por poder alternar diferentes tipos de cámaras con el fin de mejorar la jugabilidad.

- **Cámara en tercera persona** (`T`): Agrega la posibilidad de ver de cerca los objetos de la realidad. Además, el usuario puede hacer zoom in/out con la rueda del ratón. Inicialmente comienza en esta cámara.

- **Cámara isométrica** (`I`): Genera una perspectiva no centrada en el jugador como la tercera persona, y permite tener una mejor visualización de los objetos por venir.
- **Cámara libre** (`F`): Permite la exploración del escenario previo a avanzar, en caso se desee. Se destaca como la generación aleatoria del escenario es dependiente al avance del jugador y no a la posición de la cámara.

Estados gráficos:
Las teclas `F1, F2, F3, F4, F5, F11` permiten variar configuraciones en el juego, como el color de la iluminación, la velocidad del juego, el uso de texturas y otros comportamientos detallados en este informe. Se detalla más de los mismos en la sección 2.3.4

Por último el programa puede ser cerrado con las teclas `Q` y `ESC`, así como cerrando la ventana del mismo.

2.2.5. Velocidad de simulación

A modo de controlar la naturaleza de la simulación estrictamente se determina el tiempo transcurrido durante cada iteración del bucle principal (frame) a través de las bibliotecas **chrono** y **ctime** de C++.

Para cada iteración, el tiempo transcurrido desde la anterior iteración se multiplica por un coeficiente según la velocidad del juego (0.5 para lento, 1 para normal, 2 para rápido) y el resultado se envía a los métodos `update` de los objetos que se encargan de actualizar su estado en cada frame (alterando sus posiciones por ejemplo).

El tiempo transcurrido (multiplicado por el coeficiente de velocidad) se multiplica a otras constantes referentes a velocidades de traslación para conseguir los incrementos/decrementos en las posiciones de objetos de una forma agnóstica a la cantidad de frames transcurridos (al ser sensible al tiempo).

2.2.6. Uso de OpenGL

Se utilizó OpenGL, utilizando como base estructural el proyecto proporcionado por el cuerpo docente y los ejercicios prácticos. El programa principal se encarga de inicializar los elementos de **OpenGL**, **SDL**, **SDL_ttf** y **GLU**. Posterior a ello se inicia el bucle principal y el estado de estas bibliotecas sufre cambios según los flujos como se destacará a continuación.

Manejo del estado

- Se configura `SDL_SetRelativeMouseMode(SDL_TRUE)` a lo largo de la aplicación a fin de mantener el mouse en la ventana para recibir la entrada del mismo.
- Se configura `glPolygonMode(GL_FRONT_AND_BACK, GL_FILL)` como comportamiento por defecto, aunque con la tecla `F1` se alterna el valor `GL_FILL` con `GL_LINE` (a modo de mostrar solo los Wireframes).
- Se maneja el estado del `glMatrixMode` entre `GL_PROJECTION` y `GL_MODELVIEW` a la hora de hacer cambios en la proyección. Se cambia a `GL_PROJECTION` para aplicar el `glClearColor` las transformaciones de `glOrtho` y `gluPerspective`, y en el caso del último activar el `GL_DEPTH_TEST`. Luego se fija el estado en `GL_MODELVIEW`.
- Por defecto se usa el `glShadeModel(GL_SMOOTH)`, pero con la tecla `F3` dicha configuración se alterna con `glShadeModel(GL_FLAT)`.
- La matriz de OpenGL es reiniciada con `glLoadIdentity()` al inicio del bucle y al cambiar el tipo de cámara. Sus valores se almacenan y retiran del stack a lo largo del proceso de renderizado.

Se destaca que los módulos `WindowHelper`, `DrawManager`, `CameraManager` y `UIManager` aplican código que cambia los estados de OpenGL y el resto de las bibliotecas que se usan en conjunto.

Iluminación, color y materiales

Se empleó una única luz `GL_LIGHT0` teniendo la misma una **posición** determinada dinámicamente alrededor del jugador (aunque con las teclas `UHJK` es posible trasladarla alrededor del plano `x,z`). Dicha fuente emite sólo luz de **tipo** `GL_AMBIENT` y su **color** tiene tres posibles valores (para día, tarde y noche).

Se destaca que no se alteró el estado de `glMaterial` dado que no se le aplicó un material a los objetos cargados. De hecho, los objetos tienen todos un color asignado, pero en el caso de que se estén renderizando con sus texturas el color del mismo es cambiado a blanco. En el caso de que no se estén renderizando texturas (por haber apretado la tecla `F2`) se desactiva la iluminación, dado que se mantuvo la configuración por defecto de que la iluminación desactive el color de los objetos.

Display List

No se emplearon display lists en el desarrollo de la aplicación. La única optimización que se hizo en cuanto a disminuir la memoria de objetos a trabajar fue cargar las texturas y modelos 3d (.obj) una sola vez a través de la única instancia del `DrawManager`. A su vez, las coordenadas de objetos que se dibujan y no se cargan de un .obj se almacenaron como vectores constantes en el archivo `gl_constants.h` (con sus coordenadas centradas en el origen, dictando la posición del objeto a raiz de donde esté situado el origen en el sistema de coordenadas de OpenGL). Para dibujar objetos se implementó una función `DrawMultiplePoints` que va iterando mientras ejecuta comandos y datos con los que aplicar los mismos.

Vertex Arrays

No se emplearon vertex arrays en el desarrollo de la aplicación, debido a falta de tiempo.

Texturas

Todo objeto renderizado tiene una textura asociada. Las rutas de las mismas se almacenan en `constants.h` y son cargadas una sola vez a través de la única instancia del `DrawManager`. Para la carga de las mismas se encapsuló en una función el código proveniente del proyecto base brindado.

Modelos 3D

En la aplicación se cargan y emplean dos modelos 3D, `Chicken02.obj` y `Truck02.obj`. Los mismos se leen con el método `load_obj` implementado en `file_helper`, el cual se desarrolló tomando como base las implementaciones de [7] y [8]. Este método devuelve como resultado dos vectores por modelo, uno de comandos y otros de datos. Estos mismos son almacenados en el `DrawManager` y están en el formato indicado para ser pasados al método `DrawMultiplePoints` del mismo.

Ambos modelos empleados fueron realizados usando la herramienta Blender.

Cámara

Se implementaron tres tipos de cámara, sus directivas y comportamiento es gestionado a través del `CameraManager`, el cual ofrece una interfaz genérica para actualizar la cámara en base a distintos eventos y se encarga de llamar a `gluLookAt`, así como de reajustar las matrices de `GL_PROJECTION` y `GL_MODELVIEW` al cambiar la cámara seleccionada. A continuación se detallan los tipos de cámara implementados:

- **Cámara Isométrica:** Esta cámara realiza una proyección ortogonal con una distancia fija y ajusta los ángulos para que se asemeje a una proyección isométrica. Basada en el hilo [1].
- **Cámara Tercera Persona:** Esta cámara sigue y está centrada en el `Player`, permitiendo rotar alrededor de sí, así como hacer zoom in/out respecto al mismo. Implementada adaptando el código java del tutorial [2].
- **Cámara Libre:** Esta cámara permite recorrer la escena en primera persona, estando no anclada al `Player` y pudiendo trasladarse independientemente a través de las teclas `WASD`. Implementada basándose en el código del tutorial [3].

HUD

La aplicación posee un HUD dividido en componentes administrado por el `UIManager`. El mismo (empleando **SDL2_ttf**) [6] se encarga de generar las texturas (del texto) asociadas a cada posición del HUD (puntaje, monedas y game over), así como de realizar una transformación `glOrtho` con las resoluciones de la pantalla para que se puedan dibujar mapeando las coordenadas a los pixeles de la ventana.

El resto del HUD es dibujado por el `DrawManager`, el cual renderiza los componentes del HUD mencionados anteriormente en sus respectivas posiciones, así como un cuadrado negro que los une y un cuadrado con la textura de los controles en la parte inferior de la ventana.

2.3. Otros aspectos

Con el fin de optimizar la performance en tiempo de ejecución, se integró un algoritmo de *Frustum Culling* (usando distancias geométricas)[4] encapsulandolo en `FrustumManager`. Sin embargo, la implementación de la misma tiene un error, y el frustum queda configurado como un pequeño recuadro horizontal centrado en la pantalla.

La justificación principal para integrar dicha optimización fue la gran disminución de FPS tras haber agregado el modelo 3D de los autos. No obstante, tras realizar pruebas compilando la aplicación con la flag de **Release**, se encontró que dichas optimizaciones no eran necesarias y el número de FPS no disminuyó. De tal manera, se le sacó prioridad a la implementación del Frustum Culling (a favor de priorizar otros componentes de la aplicación) y no dió tiempo de terminarla.

La aplicación fue compilada con la constante `FRUSTUM_CULLING = false` (en el archivo `constants.h`), y por ende a la hora de renderizar objetos no se tiene en cuenta si están comprendidos o no dentro del Frustum.

2.3.1. Carga de Información

No hay información cargada de archivos estáticos que no se despliegue de forma gráfica. Se destaca que los archivos `constants.h` y `gl_constants.h`, almacenan configuraciones y datos que dictan el comportamiento de la aplicación y de esta manera se busca que dichas configuraciones sean fácilmente parametrizables.

2.3.2. Movimiento

El movimiento de los objetos es implementado dentro de la clase de cada objeto en cuestión, a través del método `update`. Dicho método, en base al estado del objeto y el tiempo transcurrido desde el último frame, reajusta el estado del mismo (para que luego al llamar al método `draw` se puedan dibujar con los datos respectivos al frame en cuestión).

Se destaca que, salvo la cámara, la mayoría del movimiento es sólo entre los ejes `x` y `z`. Otra excepción a esto es el `Player`, que cuando se está moviendo en cualquier dirección de los ejes `x,z` también realiza una pequeña traslación y distorsión vertical (en el eje `y`), a modo de simular un salto.

Otra particularidad del movimiento es cuando el jugador colisiona contra una superficie que obstruya el paso (`bounce`) como es por ejemplo los árboles, bordes y el carril `Wall`, tras este evento el jugador invierte la dirección de su movimiento, a modo de volver a la celda de donde inició el movimiento.

2.3.3. Colisiones

Para implementar las colisiones se adaptó el código de **colisiones AABB** (*axis-aligned bounding box*) de la siguiente referencia [5]. El algoritmo de colisiones es ejecutado en el método `update` de cada `Lane` siempre y cuando el `Player` se encuentre adyacente o dentro de la misma.

Tras determinar si ocurrió una colisión entre el `Player` y cada uno de los objetos dentro de la misma `Lane`, se retorna un `vector<OnCollision>` que contiene una lista de los eventos que el bucle principal del juego deberá resolver (*game over*, *bounce*, *coin*).

Se destaca que en el algoritmo AABB - AABB se usan hitboxes cuadradas (no se calculan colisiones en el eje `y`). Se tomó la decisión de que las hitboxes de los vehículos de la carretera (que causan game over) y del `Player` sean marginalmente más pequeñas que las dimensiones de los mismos, con el fin de favorecer al jugador en caso de una colisión con un vehículo.

2.3.4. Configuraciones

Dentro del bucle principal, al realizar el manejo de eventos, se escucha a las teclas `F1,F2,F3,F4,F5,F11`, las cuales permiten cambian el estado de OpenGL o de variables que influyen directamente en características de componentes del mismo (como el color de las luces).

Las configuraciones posibles son:

- **Wireframe** (`F1`): Alterna el `glPolygonMode` entre `GL_FILL` y `GL_LINE`.
- **Use Textures** (`F2`): Dicta si se deben asociar texturas a los modelos (en el caso contrario los modelos se renderizan con un color y se desactiva la iluminación para que se muestren los mismos).
- **Interpolated Lighting** (`F3`): Alterna el `glShadeModel` entre `GL_SMOOTH` (Gouraud) y `GL_FLAT`.
- **Lighting Color** (`F4`): Cambia los valores usados para el color de la luz de ambiente (oscilando entre día, atardecer y noche).
- **Game Speed** (`F5`): Cambia el coeficiente de velocidad en el juego (oscilando entre lento, normal y rápido),
- **Full Screen** (`F11`): Llama a `SDL_SetWindowFullscreen` y otros métodos necesarios para cambiar el estado a pantalla completa.

3. Conclusiones

A modo de conclusión se destacan los siguientes puntos:

- Se cumplió con el objetivo de desarrollar la aplicación solicitada empleando las técnicas y herramientas vistas en el curso.
- El esfuerzo inicial dedicado a modularizar la arquitectura brindó un beneficio importante a la hora de desarrollar, integrar y modificar los diferentes componentes de la aplicación.
- Gracias a que los objetos y modelos 3D utilizados tienen muy pocos polígonos, fue posible alcanzar muy buena performance (con las flags del compilador en Release) sin depender de técnicas de optimización avanzadas como Vertex Arrays y Frustum Culling.
- Se logró incluir alguno de los requisitos opcionales determinados en la consigna (junto con otras ideas que surgieron en el proceso de desarrollo), pese a ello, no fue posible incluir muchas de las funcionalidades opcionales contempladas en el alcance tentativo. Las mismas se destacan en la siguiente sección.

4. Trabajo futuro

A continuación se brinda una lista de funcionalidades no obligatorias que fueron contempladas dentro del alcance del obligatorio, pero que, por cuestiones de tiempo, no terminaron en la implementación brindada. Estas son:

- **Arreglar el algoritmo de Frustum Culling:** A modo de activarlo y mejorar la performance de tiempo de ejecución (aplicando un filtro en los objetos previo a dibujarlos).
- **Modelo 3D con mayor complejidad:** Integrar algún modelo más "esférico" como el de árboles con sus copas compuestas por triángulos que se asemejen a una esfera. Esto permitiría observar un mayor contraste respecto al algoritmo del shader de iluminación usado (Flat vs Gouraud). Debido a los problemas de performance obtenidos al correr en modo Debug, se le sacó prioridad ante el Frustum Culling. No fue hasta pocos días antes de la entrega que se observó que, al probar en modo **Release**, se alcanzó 12 veces más FPS y para ese entonces no dió tiempo de incluir esta característica.
- **Reiniciar:** Permitir reiniciar el juego tras un Game Over.
- **Tercera Persona, Movimiento relativo a la cámara:** Que el movimiento del jugador cambie según el ángulo con el que la cámara en tercera persona esté observando al mismo (en la implementación brindada la flecha UP siempre dirige al jugador en la dirección -z).
- **Skybox:** Agregar una textura de cielo y aplicarla al resto de la escena a modo de tener una textura en lugar de un color sólido de fondo.
- **Efectos de sonido:** Agregar un manejador que ejecute efectos de sonido, por ejemplo cuando el jugador se mueve, cuando se recoge una moneda, cuando ocurre un game over y cuando pasa un camión adyacente al jugador.
- **Vertex Arrays:** Cambiar el `DrawManager` para que cree y emplee Vertex Arrays a modo de evitar pasar múltiples veces los objetos a la memoria de la GPU y decrementar el tiempo de ejecución.
- **"Sombras":** A modo de simular que el jugador tiene una forma, se podría dibujar un polígono plano bajo los pies del mismo, de color gris y con transparencia.
- **Luces puntuales:** Implementar luces puntuales sin iluminación de ambiente, como por ejemplo, una luz cónica centrada en el jugador simulando una linterna.
- **Lane - Tunel:** Un tipo de `Lane` que tenga el mismo comportamiento que `Street`, pero que se cancele la luz de ambiente dentro de la misma y tenga paredes y techo, a modo que el jugador tenga que guiarse por una luz puntual que porte. El zoom de la cámara y la cámara isométrica se tendrían que ajustar para no quedar afuera del mismo.

- **Lane - Río:** Un tipo de `Lane` con el mismo comportamiento del juego Crossy Road, en donde si el jugador colisiona con la misma ocurre un game over, y para atravesarla se debe pasar por troncos que se trasladan en la misma y por lo tanto, afectan la posición del jugador.
- **Optimización - Generación de objetos:** En lugar de crear y borrar autos a demanda cuando se salen de sus límites, podría reutilizarse un conjunto fijo de autos por `Street` y trasladar los mismos a posiciones mutuamente exclusivas en el extremo opuesto de la carretera.
- **Diversificar Modelos:** Agregar distintos modelos 3D (como un modelo para el jugador tras game over) o distintas texturas para los mismos (como vehículos de distintos colores).

5. Referencias

[1] *Isometric Projection with OpenGL*
https://stackoverflow.com/questions/1059200/true-isometric-projection-with-opengl

[2] ThinMatrix *OpenGL 3D Game Tutorial 19: 3rd Person Camera*
https://www.youtube.com/watch?v=PoxDDZmctnU

[3] Learn OpenGL *Camera (Getting Started)*
https://learnopengl.com/Getting-started/Camera

[4] Lighthouse3d *View Frustum Culling*
http://www.lighthouse3d.com/tutorials/view-frustum-culling/

[5] Learn OpenGL *Collision detection (AABB - AABB collisions)*
https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection

[6] *Display text with SDL2-ttf*
https://stackoverflow.com/questions/30016083/sdl2-opengl-sdl2-ttf-displaying-text

[7] Wikibooks *Modern OpenGL Tutorial Load OBJ*
https://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Load_OBJ

[8] OpenGL Tutorial *Tutorial 7 : Model loading*
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-7-model-loading/

Planilla Obligatorio 1 - Introducción a la Computación Gráfica	
Integrantes del grupo	
- Emiliano Botti	
- Renzo Gambone	
- Germán Ouviña	

Requisitos obligatorios	**Si/No**
Ejecuta sobre Windows:	Si
Algún objeto tiene textura:	Si
Los objetos tienen normales y se aprecia la iluminación interpolada (smooth):	Si
Algún objeto fue cargado desde un archivo:	Si
Hay un game HUD dibujado en 2D o proyección ortogonal:	Si
Existen al menos dos modos/vistas posibles:	Si
La cámara rota mediante el mouse en alguna vista:	Si
La cámara puede moverse en alguna vista:	Si
El personaje principal se mueve utilizando las flechas del teclado:	Si
Se puede detener el juego (pausa):	Si
Se puede modificar la velocidad del juego independientemente del FPS:	Si
Se puede cambiar a modo Wireframe:	Si
Se puede cambiar a modo sin texturas :	Si
Se puede cambiar a modo facetado (flat):	Si
Se puede modificar la dirección y el color de una luz:	Si
Se entrega una carpeta ejecutable que funciona haciendo doble-click en el .exe:	Si

Requisitos opcionales	**Si/No**
El juego tiene más de un nivel:	Si
Algún objeto se mueve aleatoriamente:	Si
Se puede definir niveles mediante un archivo XML (o en otro formato de texto):	No
Existe un editor de niveles (dentro del juego o como una aplicación aparte):	No
Utilizan DrawArrays o DrawElements (OpenGL > 1.0):	No

Efectos Gráficos opcionales	**Si/No**
Implementan sistema de particulas:	No
Implementan sombras:	No
¿Implementan otros efectos gráficos?:	Si

¿Qué otros efectos gráficos implementaron? (descripción breve)	
1 - Distorsión y movimiento vertical durante el movimiento del jugador	
2 - Rebote del jugador tras una colisión con un elemento que bloquee paso	