

# **Introducción a la Computación Gráfica**

Obligatorio 1  
Curso: 2021

Francisco Aguirre	5.538.976-6
Santiago Calvo	5.055.578-2
Emiliano De Sejas	5.088.428-8

# Índice

<b>Índice</b>	<b>2</b>
<b>Introducción</b>	<b>3</b>
Definición del problema	4
<b>Análisis del problema</b>	<b>4</b>
<b>Diseño de la solución</b>	<b>4</b>
Arquitectura	4
<b>Implementación</b>	<b>5</b>
Descripción del entorno utilizado	5
Algoritmos y técnicas gráficas implementadas	5
Librerías utilizadas	5
Código desarrollado por terceros	5
Estados de la aplicación	5
Estructuras de datos	6
Velocidad de simulación	6
Uso de OpenGL	6
Iluminación, color y materiales	6
Vertex arrays	6
Texturas	6
Modelos 3D	7
Cámara	7
HUD	8
<b>Desarrollo del obligatorio</b>	<b>8</b>
Cargado de información	8
Movimiento	8
Colisiones	8
Settings	9
Características particulares de la solución	9
<b>Conclusiones</b>	<b>9</b>
<b>Trabajo futuro</b>	<b>9</b>
<b>Tabla de requerimientos</b>	<b>9</b>
<b>Referencias</b>	<b>11</b>

## Introducción

### Definición del problema

En esta tarea se solicitó la realización de una imitación del juego [Crossy Road](#) utilizando OpenGL, SDL principalmente, con la posibilidad de incluir librerías para cargado de objetos 3D y físicas. En el juego, el personaje (gallina) se deberá mover esquivando obstáculos como vehículos, trenes, ríos y recogiendo monedas para poder incrementar el puntaje. El objetivo es obtener el mayor puntaje posible antes de morir.

### Análisis del problema

Para esta etapa realizamos distintas sesiones de juego en las cuales intentamos identificar cuál era la esencia principal, es decir, cuales son los puntos que son imprescindibles a la hora de jugar para que un juego se “sienta” como el original [Crossy Road](#). De esto logramos destacar que lo principal es:

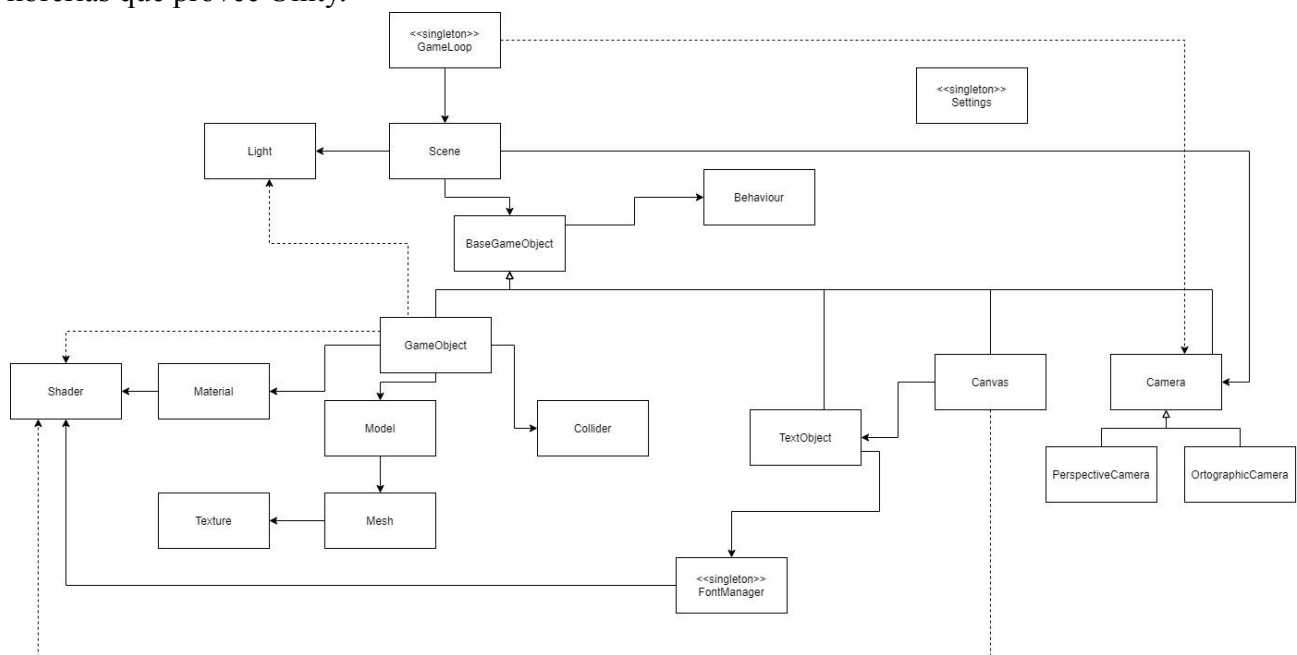
- Escenarios en filas, esto es que el escenario a través del cual la gallina corre es generado usando filas, teniendo calles, ríos y naturaleza en este formato.
- Que el personaje sea una gallina, parece algo simple pero toda la idea del juego se basa en que las gallinas normalmente cruzan calles de forma imprudente.
- Cámara isométrica.
- Que no tenga fin, es decir, no hay niveles ni dificultad a elegir, solo “correr”.

### Diseño de la solución

Para la solución se utilizaron herramientas más modernas de OpenGL pertenecientes a la versión 3.0 en adelante. Esto en particular incluye *Shaders* en vez de utilizar las rutinas estándar de OpenGL.

### Arquitectura

Para el desarrollo se tomó como referencia un sistema de componentes similar al encontrado en las librerías que provee Unity.



La principal abstracción es *BaseGameObject*, esta representa un objeto en el juego que posee *Behaviours*, comportamientos que se realizan en cada frame. De este se desprende *GameObject* que representa objetos con modelo que pueden ser dibujados en la escena, *TextObject* y *Canvas* que se utilizan para la renderización de texto en la pantalla y *Camera* que permite almacenar distintas cámaras en la escena.

## Implementación

### **Descripción del entorno utilizado**

El desarrollo se realizó por todos los integrantes en computadoras Windows con Visual Studio 2019 y el compilador MSVC.

Con el fin de compartir fuentes entre los integrantes se utilizó github.

Para los modelos se utilizó la herramienta Open Source Blender dado su accesibilidad y la familiaridad de algunos integrantes del equipo con ella.

### **Algoritmos y técnicas gráficas implementadas**

Dado el uso de *shaders* propios, decidimos utilizar el método de **Phong** para los modelos de luz (esto es el modelo de **Gouraud** aplicado en el fragment shader). De esta manera nos permitió mantener una pequeña cantidad de vértices manteniendo la calidad de la iluminación.

### **Librerías utilizadas**

Las librerías usadas son las siguientes:

- assimp - Utilizada para importar objetos 3D.  
Versión: 5.0.1  
Link: <https://github.com/assimp/assimp/releases/tag/v5.0.1>
- FreeImage - Utilizado para la carga de texturas e imágenes.
- SDL - Utilizada para crear la ventana, crear el contexto de OpenGL, obtener inputs del usuario.  
Versión: 2.0.14  
Src: <https://www.libsdl.org/download-2.0.php>
- glm - Utilizado para realizar todos los cálculos vectoriales y de rotaciones.  
Versión: 0.9.9.8  
Src: <https://github.com/g-truc/glm/releases/tag/0.9.9.8>
- glew - Librería de extensión de OpenGL.  
Versión: 2.1.0  
Src: <http://glew.sourceforge.net/>
- freeType - Utilizado para importar fuentes a OpenGL.  
Versión: 2.10.4  
Src: <https://github.com/ubawurinna/freetype-windows-binaries/releases/tag/v2.10.4>

### **Código desarrollado por terceros**

Se tomó como referencia <https://learnopengl.com/> para transformaciones, luces y renderización de texto.

### **Estados de la aplicación**

La aplicación solo tiene un estado que es el juego interminable. A medida que el personaje avanza se va generando el piso a sus pies. Los ajustes se cambian desde el juego usando ciertas teclas.

## **Estructuras de datos**

La escena se mantiene en una clase que tiene un vector (array dinámico) de todos los objetos de ella. A su vez tiene un map con las cámaras de la escena identificadas por su nombre. Los objetos pueden tener hijos que son almacenados en un vector del padre. Cada objeto tiene su modelo y textura.

## **Velocidad de simulación**

Para mantener la velocidad constante se tomó el tiempo de demora entre cada frame (llamado *deltaTime*) usando `SDL_GetTicks()` y tomando la diferencia entre un frame y el siguiente. Luego para cada valor de movimiento se multiplicó por la diferencia entre frames, normalizando la velocidad. Luego para aumentar o reducir la velocidad de la simulación se le agregó un multiplicador más a *deltaTime*.

## **Uso de OpenGL**

### **Iluminación, color y materiales**

Como fue especificado anteriormente, para mostrar efectos de color, iluminación y materiales se utilizó un *shader* genérico para todos los objetos. Mediante Assimp se obtienen los materiales y las texturas que un objeto necesita y así, al dibujar cada *mesh* (conjunto de vértices de un objeto 3D), se asignaron los *uniforms* del *shader* con los valores correspondientes de color, material y propiedades de iluminación. A su vez, en cada frame, al dibujar, se asignan las propiedades de la luz de la escena en el *uniform* correspondiente.

### **Vertex arrays**

Se utilizaron Vertex Array Objects (VAO), Vertex Buffer Objects (VBO) y Element Buffer Objects (EBO) para dibujar.

El VBO contiene todos los vértices de un objeto 3D con todos sus atributos (posición, normal y coordenadas de textura fueron usados en esta tarea).

El EBO contiene índices que indican, dentro del VBO, el orden en el cual deben ser dibujados los vértices. Esto facilita el no tener que duplicar vértices para generar los meshes.

El VAO es una estructura que contiene el VBO y el EBO. Se utiliza por conveniencia.

Al utilizar el VAO, el VBO y EBO que son cargados con vértices e índices respectivamente son asociados a dicho VAO.

De esta manera al momento de dibujar un mesh en particular lo único necesario es utilizar el VAO, este ya contiene toda la información que se precisa.

### **Texturas**

Se utilizó FreeImage para cargar las texturas necesarias.

Para los valores de configuración de estas en OpenGL se utilizó `GL_Repeat` en ambos ejes de la imagen y `GL_Linear` para las configuraciones de *mipmap* de la textura.

Luego se genera la textura a partir de los bytes provistos por FreeImage, se genera el *mipmap* y se desvincula la textura.

Por último, utilizando el identificador obtenido al generar la textura, se carga esta cada vez que se quiera imprimir un objeto que la requiera.

## Modelos 3D

Como fue mencionado anteriormente, para la carga de modelos se utilizó la librería Assimp. Assimp crea una estructura de una escena con nodos. Cada uno de estos nodos contiene muchos *meshes* y a su vez más nodos como hijos. Al cargar el modelo con assimp, se recorrió toda esta jerarquía, procesando los nodos y todas sus *meshes*.

Para procesar cada *mesh* se obtuvieron los vértices de cada una junto con sus normales y coordenadas de texturas. Luego se recorre la definición de las caras del objeto para crear el EBO. Finalmente se obtiene una textura y un material (que por razones de simplicidad se utilizó solo uno de ambos).

Con los vértices encontrados, los índices y la información del material de un objeto, se procede a crearlo. Para esto se genera un VAO que guarda de manera sencilla la información de los índices y vértices de un objeto. Luego se genera un VBO y un EBO, donde se guardarán los vértices (posición, normales, coordenadas UV) e índices respectivamente.

Luego se generan los punteros de atributos para los objetos dentro del VAO para poder diferenciar las posiciones, normales, coordenadas UV.

Finalmente se desvincula el VAO y se finaliza la creación del *mesh*. Este proceso se realiza para cada *mesh* dentro de un modelo.

## Cámara

Para las distintas cámaras disponibles en el juego, se crearon tres abstracciones: *Camera*, *OrthographicCamera* y *PerspectiveCamera*.

La abstracción *Camera* representa una cámara como si fuera un objeto dentro de la escena, que puede desplazarse por la misma, cuando es tan solo un contenedor de información de la transformación que debe ser aplicada a cada vértice para que se vea más lejos o cerca dependiendo de la posición de la cámara.

Esta transformación se logra asignando en cada frame el valor de la matriz de transformación de vista (que se obtiene basándose en la orientación de la cámara) a un uniform en el shader que se aplica a todos los objetos.

Las abstracciones *OrthographicCamera* y *PerspectiveCamera* permiten crear cámaras con una proyección ortogonal y perspectiva respectivamente. La noción de perspectiva se obtiene asignando la matriz de transformación de perspectiva de la cámara al uniform correspondiente en el shader por defecto.

Estas abstracciones permitieron crear y trabajar de manera cómoda con las distintas cámaras que hay en el juego. De esta manera, se agilizó el desarrollo y se eliminaron posibles errores.

Las distintas cámaras en el juego se pueden cambiar usando la tecla V. Estas son:

- Libre (permite mover la cámara a través de la escena)
- Isométrica (como en el juego original)
- Centrada en el jugador desde atrás (fija)
- Primera persona (permite rotar)
- Tercera persona (permite rotar)

## HUD

Para el HUD se utilizaron dos abstracciones. La primera, llamada *Canvas*, que permite dibujar un plano liso que se muestra con proyección ortogonal al punto de vista. La segunda es *TextObject*, que como su nombre lo sugiere, permite tener texto dentro de un *Canvas*. Ambos objetos se renderizan utilizando vertex y fragment shaders distintos. Esto es por la naturaleza simple que tienen ambos objetos. No tienen en cuenta luces ni una cámara, están siempre fijos respecto al *viewport*.

Un punto interesante a mencionar es cómo generamos el texto. Para esto utilizamos una clase *singleton* denominada *FontManager*. Su función es cargar todas las texturas correspondientes a las fuentes que se encuentran bajo el directorio **Assets/Fonts/** (dando una versatilidad importante a la hora de estilizar el juego), y proveerlas a todos los *TextObjects* que necesiten la fuente.

Aquí es donde radica el uso que le dimos a la librería FreeType, puesto que nos permitió cargar fuentes en formato TTF (*true type font*) de manera relativamente sencilla.

El funcionamiento consta de las siguientes etapas:

- Encontrar todos los archivos bajo el directorio mencionado.
- Crear una *FreeType Face* (representación de una fuente de FreeType)
- Generar las dimensiones del glifo y desplazamiento dentro de este (*bearing*) que se utilizará para cada carácter. Esta información se almacena en un *map* que vincula cada carácter con la información necesaria para renderizarlo junto con la textura correspondiente.

Finalmente, para renderizar texto, solo hace falta acceder al *map* que guarda la información de la fuente que se quiera utilizar, teniendo en cuenta que cada carácter se renderiza a continuación del anterior.

## Desarrollo del obligatorio

### Cargado de información

No se carga información que no se despliegue en forma gráfica. Todos los ajustes por defecto son asignados mediante código y posteriormente actualizados de esa misma manera.

### Movimiento

Para abstraer el movimiento se realizó una clase llamada *Transform* que crea la matriz de modelo para un objeto. Está abstraer la posición, rotación y escala. En base a la rotación se obtiene un vector *forward* referente a la dirección frontal del objeto.

A partir de la posición, rotación y escala definidos se crea la matriz modelo multiplicando una matriz identidad por la posición, escala y luego rotación. Dado el orden, tenemos una rotación sobre el centro del objeto, luego un cambio de la escala del objeto también sobre el centro, y finalmente la traslación (Notar orden inverso a la multiplicación).

### Colisiones

Para la detección de colisiones se utilizó una simple implementación de caja sobre un objeto. Esto es conocido como *AABB collisions*<sup>1</sup>.

A los objetos que necesitan colisiones se le define una posición mínima y máxima que se usará para crear una *caja* alrededor del objeto. Luego, antes de dibujar cada frame se chequea las colisiones con todos los objetos que tienen una *caja de colisiones* para determinar todas las colisiones en la

---

<sup>1</sup> [https://developer.mozilla.org/en-US/docs/Games/Techniques/3D\\_collision\\_detection](https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection)

escena.

Si un objeto necesita conocer las colisiones, este puede consultar su *caja de colisiones* correspondientes.

A su vez se agregó por razones de debugging la capacidad de dibujar dichas cajas en wireframe.

## **Settings**

Se implementó una clase *Settings* que mantiene toda la configuración del juego. Los valores que tiene son actualizados mediante eventos de teclas. Esta clase sigue el patrón Singleton y es consultada por el bucle principal del juego para aplicar los ajustes en ese momento.

## **Características particulares de la solución**

Como se especificó en secciones anteriores, para nuestra solución utilizamos shaders y VBOs proveniente de versiones de OpenGL superiores a 2.0.

Se utilizaron dos shaders, cada uno con fragment y vertex shaders.

Uno de ellos se utilizó para todo los objetos tridimensionales. Este toma uniforms para las propiedades de la cámara, de luces, y posiciones y materiales respectivos del objeto que se está dibujando.

El otro shader fue utilizado para dibujar los caracteres. Este toma posición y color para poder dibujar cada carácter.

## **Conclusiones**

El equipo aprendió los fundamentos de OpenGL tal como cargar objetos 3D y renderizarlos, cargar texturas y aplicar transformaciones.

A su vez, se profundizó en el conocimiento de C++.

La decisión de haber utilizado una versión mayor a 2.0 de OpenGL y por lo tanto, haber tenido que escribir shaders en GLSL permitió un aprendizaje más profundo de los temas dados en el curso, particularmente de la iluminación.

Se buscó crear un motor de videojuegos inspirado en Unity que fuera capaz de ser utilizado para crear distintos tipos de juegos.

Se logró crear una abstracción tal sobre OpenGL y las herramientas utilizadas que logró separar el código del motor del código del juego en particular.

## **Trabajo futuro**

Para que el juego sea más divertido de jugar y más lindo visualmente, se pensaron las siguientes mejoras que por temas de tiempo no fueron implementadas:

- Más obstáculos: El tren, distintos autos y más
- Partículas: Al agarrar monedas, al morir y más
- Efectos de sonido: De fondo, al agarrar monedas y más



## Tabla de requerimientos

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:	No	Es endless
Algún objeto se mueve aleatoriamente:	Si	Se mueven en direcciones distintas y con velocidades distintas
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):	Si	
Efectos Gráficos opcionales	Si/No	

Implementan sistema de partículas:	No	
Implementan sombras:	No	
¿Implementan otros efectos gráficos?:	Si	
<b>¿Qué otros efectos gráficos implementaron? (descripción breve)</b>		
1- Skybox - Cubemap		
2- Movimiento del río (se hace mediante el movimiento de una textura)		
3- Se pueden imprimir los colliders/caja de colisiones		
4- Phong shading con modelo de reflejo phong		

## Referencias

- <https://learnopengl.com>
- <https://gamedev.stackexchange.com>
- <https://www.blender.org/>