DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

***Space Invaders*® 3D**

Fundamentos de la Computación Gráfica

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Bustamante, Luis Ricardo	43/18	luisbustamante097@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

La computación gráfica es un campo muy extenso y de muchas ramas. Uno de los objetivos principales de la misma es la generación de imágenes o video a través de una computadora, lo cual es una disciplina relativamente nueva. Sus inicios se dieron alrededor de 1960 donde, gracias a la evolución constante de las computadoras, se podía realizar cosas cada vez más complejas. Un ejemplo de ello fue *Sketchpad*[1], un software creado por Ivan Sutherland en 1963 que permitía modelar figuras geométricas mediante un lápiz óptico como dispositivo de entrada. Este es uno de los precursores de los softwares CAD, y uno de los avances más grandes dentro de la computación gráfica, pues muchos de los conceptos utilizados para el modelado de figuras siguen siendo utilizados en programas modernos.

Por otro lado, anterior a la creación de Sketchpad hubo otro hito de la computación gráfica, y además este fue uno de los primeros videojuegos de la historia. *Tennis For Two*[2] se creó en 1958 por el físico William Higinbotham, era un simulador de tenis que se jugaba en un osciloscopio. A diferencia de videojuegos creados anteriormente, este fue el primero hecho con la mera intención de entretener y no como producto de una investigación académica.

De este último ejemplo se puede ver que los videojuegos son una parte fundamental en la historia de la computación gráfica. A partir de esos años los videojuegos seguirían evolucionando a la misma velocidad que la computación en sí.

Con todo esto en mente, pareció adecuado desarrollar un videojuego de aquella temprana época para este trabajo. Se decidió implementar desde cero el famosísimo *Space Invaders*®de 1978, puesto que sus dinámicas son sencillas de poder implementar, sin perder ningún aspecto fundamental de un videojuego normal. Con el objetivo de poder aplicar algunos de los temas de la materia, este fue implementado en un escenario 3D. Está escrito en *Javascript*, usando una herramienta especializada en creación de entornos 3D llamada *Three.js*, para poder ser jugado desde cualquier navegador.

[1] *Sketchpad: A Man-Machine Graphical Communication System* (PhD thesis) - Ivan Edward Sutherland
[2] *Tennis for Two, uno de los primeros videojuegos de la historia* - Hipertextual

2. Desarrollo

El informe se divide en múltiples secciones donde se explicaran los detalles más importantes del juego realizado. Aunque primero se explicarán algunos detalles técnicos de la implementación.

2.1. Detalles técnicos del juego

2.1.1. Cómo probarlo

Como ya se mencionó, el juego está hecho para funcionar en cualquier navegador. Para ponerlo en marcha es necesario levantar un servidor local en la computadora a probar, para lo cual se dejó un script en Python dentro de la carpeta raíz del proyecto llamado `start_server.py`, el cual está diseñado para levantar un servidor en cualquier sistema operativo. Para levantarlo solo hay que ejecutar con Python dicho script, es decir que se debe ejecutar la siguiente línea en una terminal:

```
python3 start_server.py
```

En caso de no poder levantar el servidor local se deja una alternativa para acceder al juego de forma online a través del siguiente enlace: `https://fcg-tp-final.glitch.me/`.

2.1.2. Implementación

Con respecto a los detalles de la implementación, todo el juego está escrito en JavaScript, junto a la librería llamada *Three.js*[1] (versión r133). Además, se utilizó 2 librerías no oficiales de *Three.js*, pero que funcionan conjunto a ella. Una para agregar una funcionalidad extra al uso del teclado, y otra para hacer más fácil el uso de *sprites* de texto. Su uso será justificado adecuadamente más tarde.

2.1.3. Rendimiento

El juego fue desarrollado y probado en una computadora con las siguientes características: *Intel Core i7-7700HQ*, 12GB de RAM y *NVIDIA Geforce GTX 1050* de GPU. El rendimiento del juego en el equipo es de 60FPS, incluso sin usar la placa de video.

Dado que es una computadora con unas especificaciones generosas, se probo en otra computadora con especificaciones más básicas para ver el rendimiento del juego. Esta tenía un *Intel Core i3-5005* con 4GB de RAM con una GPU integrada *Intel HD Graphics 5500*. En la misma el rendimiento era notablemente más bajo, pero mucho mejor de lo que se esperaba, puesto que se ejecutaba a unos 50FPS de media.

Un último detalle que no se logró arreglar es que el rendimiento del juego en el navegador Mozilla Firefox es mucho menor al probado en otros navegadores, independientemente del sistema operativo y la computadora en la que se lo pruebe. No se logró dar con la causa de ello, pero se pudo observar que en cualquier navegador basado en Chromium el juego tiene muy buena performance, con lo cual se recomienda ejecutarlo en Google Chrome para una experiencia más óptima.

2.2. Aspectos fundamentales del entorno 3D

Como en todo motor de renderizado 3D, las dos cosas más importantes del proyecto son la cámara y la escena. La escena[2] es el entorno donde todos los objetos visibles viven, por lo que cada cosa que se quiera agregar al juego deberá agregarse a la escena en una posición específica (de 3 coordenadas). Entonces la cámara no es más que un objeto más de la escena, por el cual el resto de las entidades de la escena son renderizados desde la misma hacia nuestra pantalla.

Para este juego en particular se necesitó dos pares de escena-cámara. El primer par es para los objetos del juego per se, el cual tiene un tipo de camera especial llamado Cámara Perspectiva, pues está renderiza los objetos como nosotros los interpretamos normalmente en un espacio 3D (o en la vida real), justamente por eso es la más usada para videojuegos tridimensionales, y es la principal del trabajo. El segundo par es una escena con una Cámara Ortográfica[3]. A diferencia de la anterior esta remueve el sentido de la perspectiva, dejando que todos los objetos que observa no tengan profundidad. Por esta razón es utilizada generalmente en juegos en 2D. En el caso del juego, se la necesita para mostrar los *sprites* de la interfaz.

El siguiente elemento fundamental es la iluminación. Se cuenta con una luz ambiental un poco tenue, 3 luces detrás de cada escudo y una luz encima de los enemigos. Todo esto se configuró para tener un ambiente ideal para el juego.

Por último se tienen unos elementos básicos como las grillas a los laterales y una base semitransparente entre ambas para denotar el límite de movimiento en el juego. Adicionalmente, se cuenta con un *skybox* en la escena mostrando un fondo espacial, y se tiene un colchón de "estrellas" que se mueve debajo de la base del juego que se extiende hasta el límite de visión de la cámara. Además de la estética creada por estas partículas, este tiene la finalidad de dar una mejor experiencia con respecto al 3D del juego, puesto que brinda una sensación de movimiento en el espacio bastante acorde, que también hace que se pueda diferencia de una versión 2D del juego.

2.3. Implementación del juego

Primero que nada es necesario explicar la estructura de ejecución del código en esta clase de proyectos. La burocracia para hacer funcionar esto es así: Se tiene, por un lado, la parte de inicialización de todos los elementos necesarios del juego, desde la escena y la cámara (explicados en la sección anterior), hasta cada una de las naves del escenario. A esto le llamamos la función `init()` y es ejecutada solo una vez. Por otro lado, todas las dinámicas del juego son implementadas en la función `animate()`, en la cual se programa todo lo que va a pasar en solo un *frame* de la ejecución del juego. Esta función entonces es ejecutada hasta 60 veces por segundo.

La última función descrita también tiene unos pasos básicos a seguir, en este caso se divide en 3: la línea de código `requestAnimationFrame(animate)` que es la que permite que se ejecute la función `animate()` de la manera descrita en el párrafo anterior. Luego está `render()` que es la que tendrá toda la burocracia para que se rendericen las escenas necesarias. En el caso del juego se tienen que renderizar dos escenas, primero la que representa al mundo 3D y luego la que tiene la cámara ortográfica. Por último se tiene la función `update()` que es donde se escribirán cada una de las dinámicas de los objetos del escenario.

Existe un detalle muy importante en el código fuente del juego, lo único que se ejecuta directamente en el archivo `main.js` es la siguiente función:

```
async function main(){
    await init()
    animate()
}
```

Se tienen las dos funciones importantes que se mencionaron anteriormente, pero conjunto a dos *keywords* que no había mencionado antes. `Async` y `await`[4] son *keywords* especiales para el uso correcto de programación asincrónica en JavaScript. La razón de su uso es una sola, queremos que la función `init()` se ejecute completa antes de pasar a ejecutarse `animate()`. Los motivos de ello serán explicados en la sección siguiente.

2.3.1. Inicialización de entidades del juego

Es sabido que el *Space Invaders* es un juego bastante simple en cuanto a dinámicas como en cantidad de objetos en pantalla. Sin embargo, cada objeto tiene una inicialización particular que se tratara de describir resumidamente.

Lo primero que se quiere mencionar es el modo de carga que tienen tanto la nave principal como una nave enemiga. Ambas naves son modelos distintos cargados desde un archivo `.obj`, y en el caso de la nave principal también es necesario cargar un archivo de textura `.mtl`. Lógicamente, estos archivos tienen un peso mayor al texto convencional, por lo que se van a tardar más en cargar. Esta carga es hecha mediante llamadas de la técnica AJAX[3] las cuales suceden de manera asíncrona, es decir que mientras se cargan dichos archivos el código seguirá ejecutándose. Esto es un problema porque en la función `animate()` se tiene la ejecución de lo que pasa *frame* por *frame*, suponiendo que todo está correctamente cargado, lo cual hace que si la función `init()` no se terminó de cargar bien ocurran errores en la ejecución del código. Para resolver esto se utiliza el modelo de *Promises*[5] de JavaScript,

[3] *AJAX* - Asynchronous JavaScript and XML

conjunto a las funciones `async` y `await`. Los detalles se pueden ver explícitos en el código fuente, pero básicamente se crea una *promise* al momento de cargar el archivo `.obj` o `.mtl`, la cual se deberá terminar de manera sincrónica con el código.

Esto además de solucionar el problema del `init()` incompleto también es utilizado para arreglar otro problema en la carga de naves enemigas. Como sabemos, el juego deberá crear $12 * 5 = 60$ naves enemigas, con lo cual se eligió cargar el archivo `.obj` de las naves solo una vez y luego asignar el mismo *geometry* para las 60 naves. De esta forma se agiliza la creación de las naves y además resulta en un mejor rendimiento de juego. El problema de esta idea era que cuando la creación de las 60 naves empezaba antes de que se termine de cargar dicho archivo; el código fallaba. Por lo cual también se utilizó dicho modelo de *promises* para solucionar dicha cuestión.

Fuera de ello, la inicialización de las naves es bastante sencillo. Tanto para las naves enemigas como para la nave principal se utilizó un diseño *low poly* para reducir la carga del navegador. Adicionalmente, se redujo la cantidad de vértices de los modelos originales en el software *Blender*[4] para poder tener un mejor rendimiento en el algoritmo de detección de colisiones (el cual se explicara luego).

Las naves tienen el material estándar del paquete Three.js, fue elegido en vez de la que venía con el archivo original para optimizar el rendimiento. También se le agregó una componente de *metalness* y *roughness* para mejorar su perspectiva 3D, pues de lo contrario parecían simples *sprites*. Por otro lado, la textura de la nave principal es la del archivo original, puesto que al ser un único objeto en escena con ese material no impactaba al rendimiento de juego.

Por último, se inicializan los escudos que protegen la nave, los cuales son figuras simples que se van haciendo transparentes a medida que se les dispara. Y además se inicializan los corazones en la cámara ortográfica, que representan las vidas restantes de la nave.

2.3.2. Dinámicas y animación de las entidades

Para esta altura ya queda bien en claro que la función `animate()` tiene programado que se va a realizar en cada *frame*. El juego tiene varias dinámicas particulares, por lo que esta sección será dividida en las 4 principales.

2.3.2.1 Movimiento de las naves y de la cámara

El movimiento de las naves enemigas es ciertamente sencillo, pues lo que tienen que hacer es moverse de un borde al otro. Este movimiento se realizará teniendo en cuenta la última nave viva del lado adecuado, es decir que si hay una sola nave viva, esta seguirá yendo desde un borde al otro. Por otro lado, cada choque con un borde cambiará de sentido a las naves para poder moverse al borde contrario, aumentando su velocidad mínimamente y haciendo un pequeño paso dirigido a la nave principal para darle la dificultad conocida al juego.

Con respecto a la nave principal, también tiene una dinámica básica, ante la presión de una de las teclas de movimiento, la nave se moverá hacia donde corresponde. Se utilizó una versión alternativa del archivo `KeyboardState.js`[6] de Three.js. La razón de su uso es porque la clase `KeyboardState` original no tiene la función `.pressed()` que devuelve `true` mientras que la tecla esté presionada.

Algo más interesante se da con respecto al movimiento de la cámara de la escena principal. Esta sigue al movimiento de la nave principal, pero no de forma estática. Tiene un pequeño *delay* entre el movimiento de la nave y la llegada de la cámara a la misma coordenada x de la nave. Esto se agregó para tener un poco más de dinamismo con respecto al movimiento. Los detalles de la implementación están bien documentados en el código fuente, pero básicamente se logró con una pila de movimientos, en el cual se van apilando los siguientes movimientos a realizar de la cámara, pero se va vaciando más lento de lo que se debería mover. Cuando la cámara se acerca a cierto radio de alejamiento de la nave, la pila se limpia y la cámara salta directamente a la coordenada x de la nave.

2.3.2.2 Disparos de las naves

Todos los disparos de las naves son un pequeño cubo con la capacidad de ser colisionable contra el objetivo a golpear. Los disparos de la nave principal son desencadenados por la tecla Espacio, y existirán

[4] *Blender* - Free and Open 3D Creation Software

hasta que impacten con una nave enemiga o se alejen lo suficiente para que la cámara deje de verlos.

El método por el cual disparan las naves enemigas es ligeramente más complejo. En cada *frame* se tiene una probabilidad del 4 % de que una nave de la primera línea dispare. Si ocurre que una nave tiene que disparar, se elige al azar entre todas las naves vivas de la primera línea para que solo una de ellas dispare. Luego sus disparos tienen el mismo comportamiento que los de la nave principal.

2.3.2.3 Sistema de detección de colisiones

El framework de Three.js no posee un sistema de detección de colisiones nativo, por lo cual se tuvo que implementar una solución para poder detectar las colisiones de las balas de las naves contra sus objetivos. Tanto las naves enemigas como la principal tienen un sistema de detección de colisiones especial para poder implementar la dinámica de destrucción de naves, o la de reducción de vida de la principal. Este sistema está basado en el sistema de colisiones de Lee Stemkoski[7] y utiliza la técnica de *Raycasting*[5]. Los componentes principales del método son el objeto a ser colisionado, una lista de los objetos colisionables del mismo y una función `handler` que será ejecutada cuando haya una colisión.

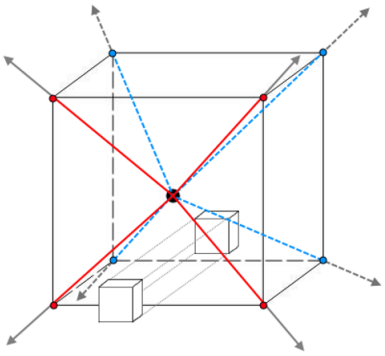

Figura 1: Raycasters en un cubo simple, junto a cubo muy pequeño que puede pasar sin ser detectado.

El método funciona evaluando cada uno de los vértices del objeto a ser golpeado, creando un rayo desde el centro del objeto en dirección a cada uno de estos vértices. Si encuentra algún objeto collisionable intersecando con el rayo y además que esté dentro de la distancia entre el vértice y el origen del objeto, entonces se tendrá una colisión como tal; con lo cual se ejecutará la función *handler*. Con esto se controla cuando las balas colisionan contra el objetivo y se aplican los efectos deseados dentro del *handler*. El algoritmo funciona bien, pero resulta ser bastante pesado pues evalúa vértice por vértice del objeto a ser golpeado.

Por otro lado, si el objeto es muy simple (con muy pocos vértices) el sistema de colisiones anterior tiene muchos puntos ciegos donde no se detectaran las colisiones. Este fenómeno se puede observar en la Figura 1. Justamente ocurre eso con los escudos, debido a que estos tienen la geometría de un prisma, por lo que tienen varios puntos ciegos para las balas por su pequeño tamaño

Así pues se tuvo que implementar un nuevo sistema de colisiones para los escudos, el cual resulta ser mucho más sencillo que el anterior descrito. Este solo evaluará si las balas están dentro del prisma que representa el escudo. Este método también aliviana un poco la carga del CPU, aunque solo es aplicable para figuras muy simples y cuadradas.

2.3.2.4 Fin del juego

El juego se termina solo si ocurre alguna de las siguientes condiciones:

- No hay más naves enemigas que destruir. En cuyo caso el juego se da como ganado.
- La vida de la nave principal sea cero, o que los enemigos se acerquen demasiado a la nave. En estos caso el juego se da como perdido.

En ambos casos se finalizará con un *sprite* de texto indicando si se ha ganado o perdido la partida. Se decidió implementar los *sprites* de texto mediante un componente no oficial de Three.js llamado *three-spritetext*[8]. La razón de su uso es solo por facilidad de uso y legibilidad del código, puesto que de lo contrario es mucho más engorroso renderizar texto en pantalla.

[5] *Raycaster* for Three.js

3. Conclusiones

El trabajo realizado terminó siendo realmente un desafío, debido a que solo se tenía los conocimientos básicos del framework, como los del lenguaje en sí. En un principio se creyó que el elegir un juego sencillo y clásico tendría una implementación mucho más sencilla, pero se terminaron necesitando soluciones bastante elaboradas a problemas que no parecían tan difíciles. Se aprendió muchísimo en el proceso, pues por cada obstáculo nuevo surgía una manera renovada de pensar para poder resolverlo.

Por otro lado, se utilizaron muy pocos recursos vistos a lo largo de la materia. Sin embargo, el hecho de tener tales conceptos permitieron un desarrollo del videojuego bastante bueno, con un resultado lo suficientemente optimizado para poder ser ejecutado en un espectro amplio de computadoras. Esto debido a que se trató de aligerar lo más que se pudo la carga del CPU y GPU, dado que se entendían muchas de las cosas que pasaban por debajo del framework.

Asimismo se dejaron muchas posibles optimizaciones y mejoras en el camino, que podrían haber resultado en tener un juego más performante y estilizado, pero que no formaban parte de los objetivos del proyecto, así como tampoco resultaban fáciles de implementar. Todas las dinámicas realizadas fueron pensadas equilibrando entre lo fácil de implementar, la relevancia de ellas dentro del juego y que no impacten demasiado en el rendimiento del mismo.

Por todo lo anterior, y además por haber logrado un videojuego con todas sus características principales funcionando correctamente, se espera haber alcanzado los objetivos buscados por la consigna del trabajo y de la matería en si.

Referencias

[1] Three.js team. Fundamentals of three.js. https://threejs.org/manual/#en/fundamentals.

[2] Three.js team. Creating a scene. https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene.

[3] Three.js team. Orthographic camera. https://threejs.org/docs/index.html#api/en/cameras/OrthographicCamera.

[4] MDN Web Docs. Making asynchronous programming easier with async and await. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Async_await.

[5] JsPoint. Javascript promises and async/await: As fast as possible. https://medium.com/jspoint/javascript-promises-and-async-await-as-fast-as-possible-d7c8c8ff0abc.

[6] Lee Stemkoski. Keyboard detection. https://stemkoski.github.io/Three.js/Keyboard.html.

[7] Lee Stemkoski. Collision detection. https://stemkoski.github.io/Three.js/Collision-Detection.html.

[8] Vasco Asturiano. A sprite based text component for three.js. https://github.com/vasturiano/three-spritetext.