Computación Gráfica
Trabajo práctico especial 1 - Ray Caster
Grupo 2

Guillermo Campelo
Juan Ignacio Goñi
Juan Tenaillon
Santiago Vazquez

Resumen

Se diseñó y desarrolló un motor de Ray Casting, determinando intersecciones entre rayos originados en la cámara y objetos. Esto permite representar escenas con figuras dentro.

Palabras Clave: *Ray Casting, escena, rayos, objetos, primitivas, triángulo, esfera, cuadrilátero..*

1. Introducción

En este informe, se explicará el diseño y el desarrollo del motor de Ray Casting, explicando cada una de sus componentes y que nos motivo a desarrollarlas de la manera en que lo hicimos.

En la segunda sección nos enfocaremos en el motor de Ray Casting y como se utiliza, luego en la tercera sección nos enfocaremos en la escena y los objetos que la componen, como sean las esferas, los triángulos y los cuadriláteros. En la cuarta sección nos enfocaremos en los resultados obtenidos y por último, en la quinta sección, presentamos nuestras conclusiones.

2. Ray Caster

2.1. Descripción

Ray Casting es una técnica que utiliza la intersección de rayos con una superficie para resolver un conjunto de problemas en el campo de la computación gráfica. Representa una solución fácil al problema de la representación en una escena.

En este caso se modela una escena almancenada en memoria, para la cual se fija una posición y dirección para la cámara. Desde esta se disparan rayos a intervalos regulares y para cada uno se muestra el color correspondiente al objeto más cercano con el que colisiona.

2.2. La Escena

Actualmente existen 6 escena predefinidas, pero la arquitectura contempla obtener nuevas escenas desde otras fuentes, por ejemplo de un archivo. En la sección 4 se describen y se muestran las imágenes obtenidas.

2.3. La Cámara

El modelo de la cámara contempla el ángulo de apertura del lente para el cálculo de la representación de la escena. Conociendo el ángulo de apertura horizontal y la relación entre el ancho y el alto de la imagen, se toma un plano imaginario con normal al vector director de la cámara. Se utiliza un vector para dar la rotación de la cámara y se realizan las correcciones necesarias al plano.

Luego en función de la resolución horizontal y vertical, se toma el delta ángulo en el eje x y eje y relativos a la cámara.

El Ray Caster iterará por los pixeles de la imagen y la cámara devolverá el rayo con origen en el centro de la cámara y que pase por el correspondiente pixel.

2.4. Modo de Uso

Para ejecutar el Ray Caster hay diferentes opciones. A continuación especificamos cuales son estas opciones y mostramos su modo de uso.

2.4.1. Nombre de Archivo de Entrada

Para poder utilizar el Ray Caster, es necesario especificar un nombre de archivo de entrada. Los posibles nombres para las escenas de entradas son de la forma *scene#.sc* y para utilizar el ray caster con un archivo de este tipo hay que especificarlo de la siguiente manera:

```
-i [nombre de archivo]
```

Suponiendo un nombre de escena *scene*1.*sc*, una posible llamada al Ray Caster sería:

```
cg_tpe1 -i scene1.sc
```

Este campo es **obligatorio** para poder utilizar la aplicación.

2.4.2. Nombre de Archivo de Salida

Al ejecutar el Ray Caster, se producirá un archivo de salida. El usuario de la aplicación puede elegir un nombre en particular para el mismo. Esto se hace utilizando la siguiente opción:

```
-o [nombre de archivo]
```

En caso de no especificar ningún nombre en particular para el mismo, el nombre del archivo de salida será el mismo que el archivo de entrada pero con extensión $*.bmp$ o $*.png$. Suponiendo que se requiere un nombre de archivo específico, el uso de la opción es la siguiente:

```
cg_tpe1 -i scene1.sc -o scene1.png
```

El uso de esta opción **no es obligatoria** para el uso de la aplicación.

2.4.3. Pixeles en la Imagen de Salida

Para indicar la cantidad de pixeles en la imagen de salida, se utiliza la opción indicada a continuación:

```
-size <alto>x<ancho>
```

En caso de no especificar ninguna cantidad en particular, se utilizará el tamaño por defecto de 640x480 pixeles. Un ejemplo de su uso sería:

```
cg_tpe1 -i scene1.sc -size 800x600
```

Este campo **no es obligatorio** para poder utilizar la aplicación.

2.4.4. Campo de Visión

Para indicar el ángulo de apertura horizontal en grados, se va a utilizar:

```
-fov x
```

donde x es un número en grados. En caso de no especificarse ningún ángulo, el valor por defecto es de 60 grados. Un ejemplo de su uso sería:

```
cg_tpe1 -i scene1.sc -fov 80
```

Este campo **no es obligatorio** para poder utilizar la aplicación.

2.4.5. Asignación de Color

Para indicar el modo de asignación de color de las primitivas en la escena, hay que utilizar esta opción:

```
-cm [random | ordered]
```

Donde *random* es el valor por defecto e indica una asignación de colores de forma aleatoria y *ordered*, donde la asignación de color dependerá del orden en que fueron descubiertas las primitivas. El orden de asignación es: *violeta*, *azul*, *verde*, *amarillo*, *naranja* y *rojo*. El orden de asignación a las figuras en la escena esta determinado por la secuencia de colisiones. El primer objeto dibujado en pantalla, obtiene el primer color de la lista. Luego siguen los demás objetos con los siguientes colores. Un ejemplo de su uso sería:

```
cg_tpe1 -i scene1.sc -cm ordered
```

El uso de esta opción **no es obligatoria** para el uso de la aplicación.

2.4.6. Variación del Color

Para indicar el modo de variación del color de los elementos de la escena, hay que utilizar esta opción:

```
-cv [linear | log]
```

Donde *linear* es el valor por defecto e indica que la variación del color es proporcional a la distancia entre el punto de colisión en la primitiva y la posición de la cámara y *log*, donde la variación del color es logarítmica. Un ejemplo de su uso sería:

```
cg_tpe1 -i scene1.sc -cv log
```

El uso de esta opción **no es obligatoria** para el uso de la aplicación.

2.4.7. Tiempo de Ejecución

Para determinar el tiempo que demoró la aplicación para renderizar la escena indicada, se utiliza:

```
-time
```

Un ejemplo de su uso sería:

```
cg_tpe1 -i scene1.sc -time
```

El uso de esta opción **no es obligatoria** para el uso de la aplicación.

3. Primitivas

Para modelar las escenas, se debieron modelar los distintos objetos presentes en la misma. En este caso, las primitivas seleccionadas fueron: *triángulo*, *cuadrilátero* y *esfera*.

A continuación explicaremos, para cada una de las primitivas, cómo se decidió el diseño de las mismas.

3.1. Rayo

3.1.1. Representación

La representación de un rayo consiste en un punto de origen y un punto de destino. Considerando que un rayo es de la forma:

$$Ray(t) = R_0 + t * R_d = (x_0, y_0, z_0) + t * (x_d, y_d, z_d) \quad (1)$$

Donde R_0 representa el punto donde se origina el rayo y R_d es la dirección del mismo. Para calcular la dirección del rayo realizamos la resta entre el destino y el inicio del mismo:

$$R_d = (x_{destino} - x_{origen}, y_{destino} - y_{origen}, z_{destino} - z_{origen}) \quad (2)$$

Al representar el rayo de esta forma, se simplifica la prueba de intersección entre el rayo y el plano que incluye a la figura, ya que si existe un t tal que se intersecten y además $t > 0$ entonces el rayo intersectará a la figura.

3.2. Triángulo

3.2.1. Representación

Hay muchas formas de representar un triángulo: 3 puntos, 3 vectores, entre otras. Nos pareció que la más intuitiva era representarlo como 3 puntos en el espacio, y eso hicimos. Pero tres puntos en el espacio pueden no formar un triÃąngulo, por lo que se corroboró que lo hagan. Ésto consiste en obtener 2 de los 3 vectores de lado (por ejemplo, el vector que une p1 y p2 y el que une p2 y p3) y el vector normal al plano que pasa por p1 y que ellos definen. Entonces si el vector normal es nulo, los tres puntos no forman un triángulo.

3.2.2. Intersección

Para determinar la intersección de un triángulo(p1,p2,p3) y un rayo (r0,r1) se implementó el siguiente algoritmo:

Primero, se obtiene el vector dirección del rayo y se hace el producto escalar con el vector normal del triángulo. Si éste producto es muy cercano a 0 (una tolerancia predefinida), entonces el rayo está sobre un plano paralelo(o el mismo) al del triángulo. Para saber si está en el mismo plano o en uno paralelo, se calcula el producto escalar entre el vector normal y el vector que une el punto p1 del triÃąngulo y el punto de origen del rayo r0. Entonces, si éste producto escalar es muy cercano a cero, entonces se toma que el rayo y el triángulo están sobre el mismo plano, caso contrario, están en planos paralelos. El cociente entre estos dos productos escalares determina si el rayo va hacia el plano o no dependiendo si es mayor a cero o no.

Una vez que se sabe que el plano determinado por el triángulo y el rayo intersectan, se busca ver si el punto donde intersectan está dentro del triángulo o no. El punto de intersección está determinado por la suma entre el punto de origen del rayo r0 y el vector dirección del rayo escalado al cociente de los productos escalares mencionado anteriormente. Finalmente, se calculan determinantes para saber si el punto está dentro de la figura del triángulo o no.

3.3. Cuadrilátero

3.3.1. Representación

Para representar un cuadrilátero, se pensó en la utilización de los cuatro puntos en el espacio. Se eligió esta representación debido a que, además de ser la más simple, también resulta útil al momento de calcular los planos en los que se encuentra para poder determinar la intersección de los rayos con el mismo.

En la siguiente sección, se explicará a grandes rasgos, cómo fue realizada la intersección del rayo con el cuadrilátero.

3.3.2. Intersección

Para determinar la intersección entre un cuadrilátero y un rayo, se dividió el cálculo en dos partes.

La primera parte consistie en determinar si el rayo en cuestión, intersecta al plano que contiene a la primitiva. Para realizar esto, se calcula el *plano contenedor* y una vez obtenida la ecuación del tipo

$$\mathrm{Ax} + \mathrm{By} + \mathrm{Cz} + \mathrm{D} = 0$$

En esta ecuación, se obtiene el t para el cual el rayo intersecta al plano y en caso de no existir tal t, el rayo no intersecta a la primitiva, pero en caso de si existir, es necesario determinar si el punto de intersección entre el plano y el rayo pertenece a la primitiva.

Para la segunda parte del cálculo, se supuso que cualquier cuadrilátero son dos triángulos. Donde si el cuadrilátero tiene por puntos $p1$, $p2$, $p3$ y $p4$, el primer triángulo estaría compuesto por $p1$, $p2$ y $p3$ y el segundo estaría compuesto por $p1$, $p4$ y $p3$. Asumiendo esto, despues se utiliza una función que determina si un punto está dentro de un triángulo que fue explicada en la sección 3.2.

3.4. Esfera

3.4.1. Representación

Para representar la esfera se pensó que sólo se necesitaba un punto para representar el centro y un valor que determine el radio de la misma.

4. Resultados Obtenidos

El motor de Ray Casting tiene la capacidad de renderizar cualquier escena que esté cargada en memoria. En esta implementación, se decidió mostrar la capacidad del motor mediante la renderización de seis escenas. Tres de ellas, propuestas por la cátedra y las otras tres, propuestas por el grupo desarrollador. A continuación, enseñaremos cada una de ellas con una breve descripción.

4.1. Primera Escena: `scene1.sc`

Esta escena contiene una pirámide rectangular, cuyo lado de la base y su altura tiene una longitud de 5. La base de la misma está centrada en el origen, en su vértice superior tiene posicionada en perfecto equilibrio una esfera de radio 1 y a una distacia de 3, en cada uno de los vértices, tiene una esfera de radio 0.5. Ver **Figura 1**.

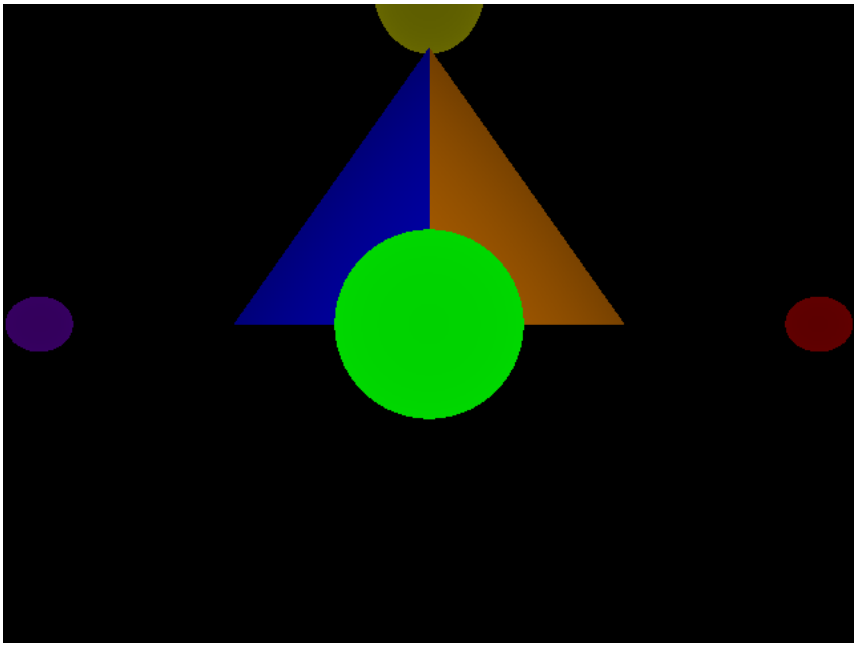

Figura 1: Escena número 1, triángulo y esferas.

4.2. Segunda Escena: `scene2.sc`

Esta escena contiene 64 esferas de radio 0.5 dispuestas en forma de cubo. Esto es, 4 esferas sobre el eje x, 4 sobre el eje y y 4 sobre el eje z formando un

cubo de 64 esferas. Cada una de ellas debe estar a una distacia de 1.5 de la otra y el centro de la figura debe estar en el centro del sistema de coordenadas. Ver **Figura 2**.

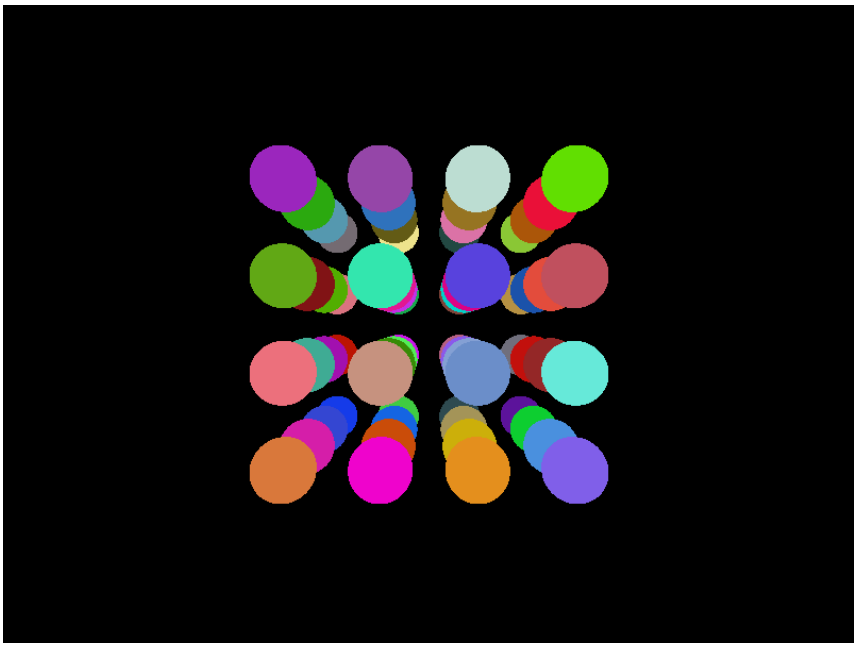

Figura 2: Escena número 2, esferas formando un cubo.

4.3. Tercera Escena: `scene3.sc`

Esta escena contiene, de forma intercalada, 3 cubos de lados 2 y dos esferas de radio 1 alineados de manera centrada sobre el eje x. Las bases de los cubos y las esferas están apoyadas sobre el eje $y = 0$ y la separación entre las figuras es de 0.5. Ver **Figura 3**.

4.4. Cuarta Escena: `scene4.sc`

Esta escena es similar a la tercera escena, pero los cubos estan formados por triángulos en vez de cuadriláteros. Ver **Figura 4**.

4.5. Quinta Escena: `scene5.sc`

Esta escena contiene 11 esferas alineadas a lo largo del eje x. Todas ellas a una distancia de 0.5 y con un radio de 0.5. Ver **Figura 5**.

4.6. Sexta Escena: `scene6.sc`

Esta escena contiene 20 triángulos que forman una estrella con centro en el centro del sistema de coordenadas. Ver **Figura 7**.

Figura 3: Escena número 3, cubos y esferas intercaladas.

4.7. Septima Escena: `scene7.sc`

Un único triángulo con una punta cerca de la cámara y las otras en forma lejana para ver la variación de color. Ver **Figura 8**.

5. Comentarios Finales

La utilización del Ray Caster para representar una escena resulta muy conveniente por su simplicidad, aunque no provee una solución real si las necesidades de respuesta solicitan imágenes rendereadas en tiempo real.

Luego de diseñar y desarrollar el motor de Ray Casting, podemos concluir que el motor es útil para el problema de visión de objetos en una escena desde un punto dado. Utilizando este motor, es posible determinar si un objeto es visible desde donde está la cámara o si el mismo es obstruido por otro objeto en el medio.

Durante la implementación, fuimos agregando optimizaciones tempranas para aumentar la performance. La utilización de *Threads* y particionado de la imágen disminuyó en gran medida el tiempo de renderización, aunque con ciertos comandos, tuvo que ser deshabilitada dicha optimización para cumplir con los requerimientos del problema.

Además, se nos ocurrió como una posibilidad, implementar el motor de Ray Casting para HTML5. Esto es debido a que es una nueva tecnología donde no hay nada similar implementado y se podría, de manera simple, obtener un motor 3D de reconocimiento de primitivas gracias al conocimiento adquirido en el desarrollo de este trabajo práctico.

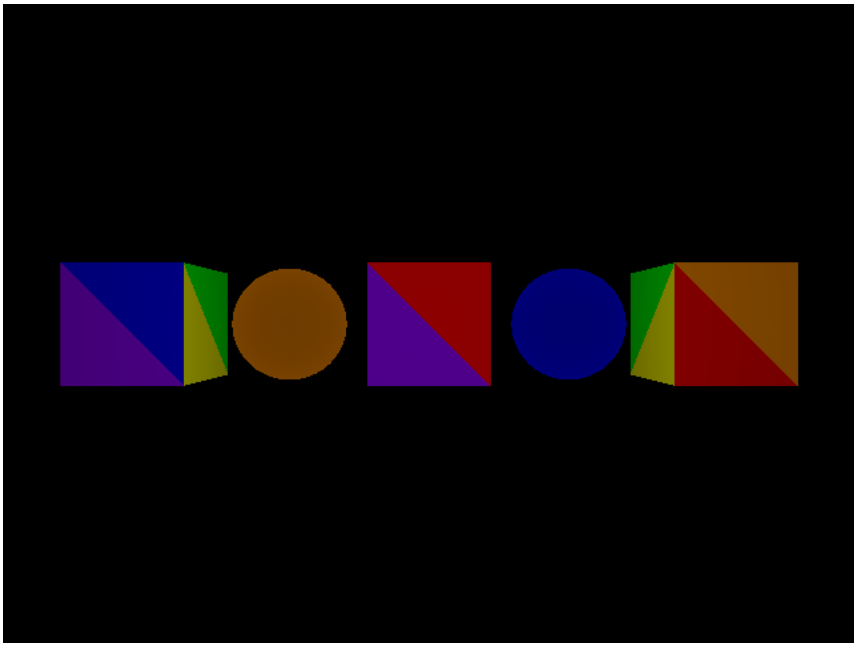

Figura 4: Escena número 4, cubos y esferas intercaladas.

Figura 5: Escena número 5, 11 esferas alineadas.

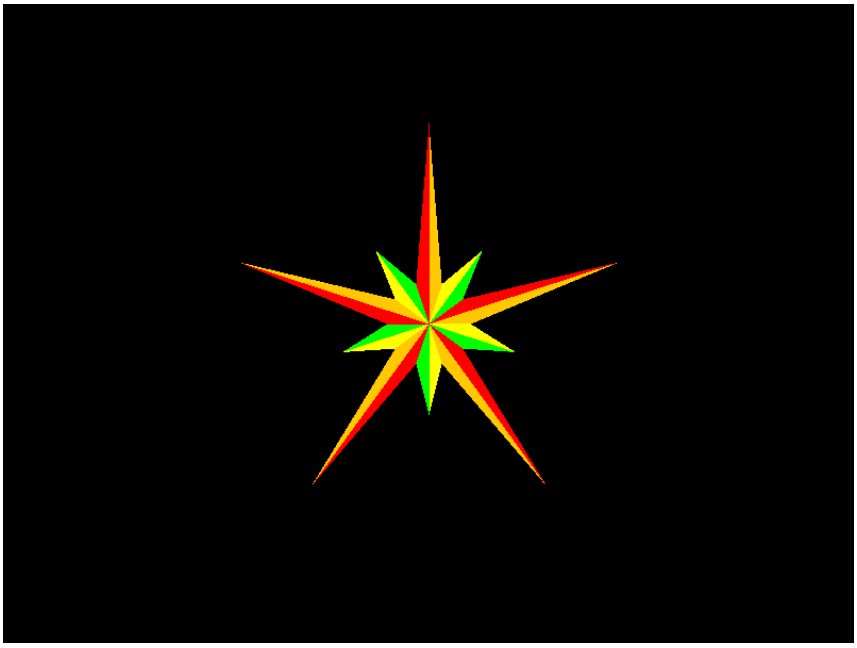

Figura 6: Escena número 6, 20 triángulos formando una estrella.

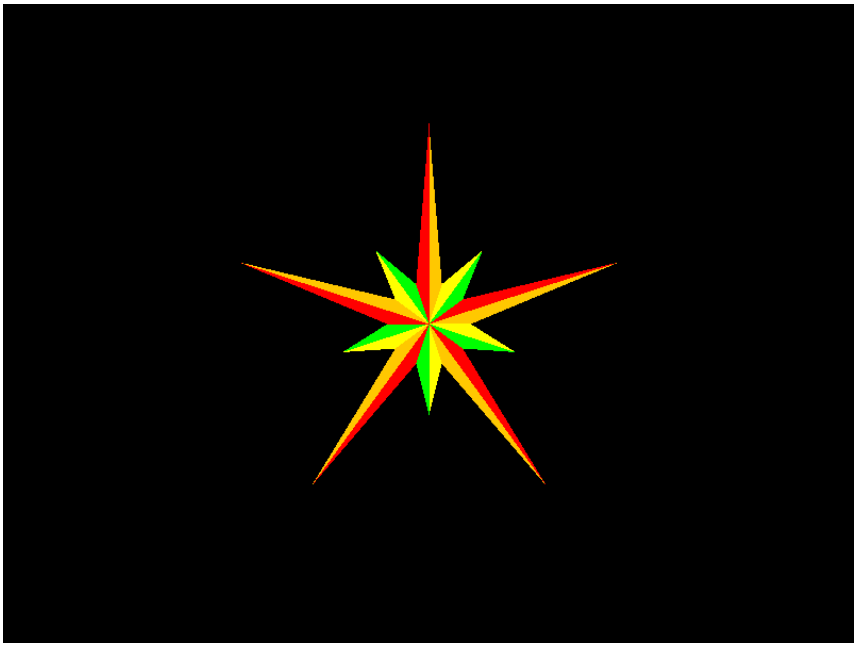

Figura 7: Escena número 6, 20 triángulos formando una estrella.

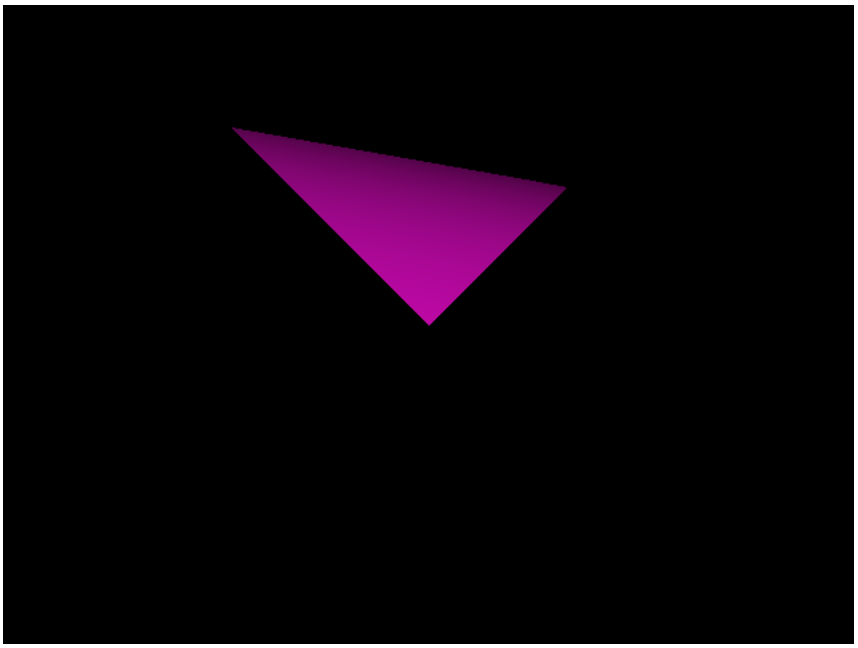

Figura 8: Escena número 7, un único triángulo que muestra la variación de color.