

UNIVERSIDAD CATOLICA SAN PABLO - AREQUIPA  
FACULTAD DE CIENCIAS DE LA COMPUTACIÓN



**Computacion Grafica**

---

**Informe Proyecto Final**

**Rubik Invaders**

---

Advisor: Manuel Eduardo Loaiza Fernández  
Student: Loayza Huarachi Angel Josue  
Salas Ticona Frank Roger

AREQUIPA, DECEMBER 2023





# Contents

<b>1</b>	<b>Introducción</b>	<b>4</b>
<b>2</b>	<b>Metodología y Logica</b>	<b>5</b>
2.1	Clase Figure . . . . .	5
2.2	Clase ShaderClass . . . . .	8
2.3	Clase Cube . . . . .	10
2.4	Clase deathRay . . . . .	13
<b>3</b>	<b>Resultados</b>	<b>16</b>
3.1	Guia de juego . . . . .	16
3.2	Imágenes de Gameplay . . . . .	17

# 1 Introducción

El presente informe aborda el desarrollo de un videojuego basado en el popular videojuego retro: “Space Invaders”, con el nombre de “Rubik Invaders”. Este proyecto se ha desarrollado utilizando el lenguaje de programación C++, aprovechando las potentes capacidades de la biblioteca gráfica OpenGL y las utilidades matemáticas de GLM. La implementación del juego se basa en una estructura de programación orientada a objetos que permite la interacción del jugador con un entorno 3D.

En Rubik Invaders, el jugador controla un personaje en un escenario donde los obstáculos son cubos de Rubik desarmados. La mecánica principal del juego implica la destrucción de estos cubos mediante rayos láser emitidos por el personaje controlado por el usuario. Una vez destruidos, los cubos deben rearmarse, lo que añade una capa adicional de desafío y estrategia al juego.

La elección de C++ como lenguaje de programación ofrece una base sólida para implementar la lógica del juego y manejar de manera eficiente los aspectos complejos del procesamiento gráfico. La utilización de OpenGL y GLM proporciona las herramientas necesarias para renderizar entornos tridimensionales, crear efectos visuales atractivos y aplicar algoritmos para resolver el rearmado de los cubos de Rubik, un aspecto fundamental del juego.

A lo largo de este informe, se presentarán los detalles del diseño y la implementación del videojuego, incluyendo la programación orientada a objetos empleada para la gestión de personajes, obstáculos y la interacción del jugador. Se discutirán los algoritmos utilizados para resolver el rearmado de los cubos de Rubik dentro del contexto del juego y cómo se han integrado en la mecánica del mismo.

El objetivo principal de este proyecto es ofrecer una experiencia de juego envolvente, desafiante y entretenida, al tiempo que proporciona un estudio técnico sobre el desarrollo de videojuegos en un entorno tridimensional utilizando herramientas poderosas como C++, OpenGL y GLM. De igual manera, se comparte el código libre en el siguiente link a GitHub para futuras mejoras: <https://github.com/angel452/R-Tree-and-VTK>



## 2 Metodología y Logica

A continuación, se describirá como se llevó a cabo esta implementación, detallando las funciones, clases, técnicas, etc.

En el código presentado, contamos con 4 clases primordiales para modelar este proyecto como: La clase **deathRay**, la clase **cube**, la clase **Figure**, y la clase **ShaderClass**.

### 2.1 Clase Figure

Empezando por lo más pequeño a lo más grande; la clase **Figure** nos servirá para guardar la información de un único cubo de los 26 en total para armar un cubo de Rubik. Eso quiere decir que es en esta estructura donde guardamos los 216 vértices para formar un cubo junto a métodos necesarios como las transformaciones típicas en computación gráfica: rotación, escalar, movimiento, etc.

A continuación les mostramos la estructura de la clase:

```
1  class Figure {
2      public:
3          unsigned int VAO, VBO;
4          int sizeVertexes;
5          float* vertexes;
6
7          Figure();
8          ~Figure();
9
10         void move(float, float, float);
11         void scale(float);
12         void rotate(float, char);
13         void rotate_inversa(float, char);
14         void matrixVectorMult(const float*, const float*, float*);
15         void transformVertices(const float*);
16         void transformVerticesCenter(const float*);
17         void calculateCenter();
18         void printVertexes();
19     };
```

### Variables publicas:

- VAO, VBO: Elementos esenciales de OpenGL para almacenar y organizar datos de los vértices y atributos de la geometría en la GPU
- sizeVertexes y vertexes: En uno guardamos la cantidad de vértices para tenerlo a disposición siempre; y en el otro, es donde inicializamos todos los vértices a manera de arreglo dinámico (la manera en como se hace esto se ve en el constructor).

### Constructores y Destructores:

- Figure(): Como se mencionó, en este constructor se aprovecha para inicializar las variables como: sizeVertexes y vertexes. Notemos que la estrategia que usamos es de usar memoria dinámica para guardar esta información.

```
1  Figure::Figure():
2      sizeVertexes(sizeof(cubeVertex) / sizeof(float))
3  {
4      vertexes = new float[sizeVertexes];
5      for (int i = 0; i < sizeVertexes; i++) {
6          vertexes[i] = cubeVertex[i];
7      }
8  }
```

- ~Figure: Usado para liberar la memoria que ya no estemos usando. Un dato curioso es que cuando intentamos graficar algún objeto usando memoria dinámica, es requerido en su totalidad un destructor; si no, no se mostrarán todas las figuras que uno intenta plasmar. Esto se hace simplemente llamando a:

```
1  delete [] vertexes;
```

### Métodos:

- scale, rotate, move: Estas 3 funciones tienen una estructura similar. Únicamente guardan en un array la matriz respectiva que será usada para la transformación que el jugador/usuario desee; dicha matriz, la pasan a la función transformVerticesCenter



- `ransformVerticesCenter`: Esta función tiene la tarea de cambiar los vértices a las nuevas posiciones dependiendo de la matriz que se le paso por parámetro; esto incluye, la lógica para mover los vértices al centro, aplicar la transformación, y devolver a la antigua posición; así, evitamos crear movimientos que no queremos. La lógica usada se encuentra en el siguiente código dividido en 3 pasos:

```
1 void Figure::transformVerticesCenter(const float* matrix) {
2     // 1. Calculo el centro coordenadas de la figura (Cx, Cy,
3     Cz)
4     float Cx, Cy, Cz;
5     for(int i = 0; i < sizeVertexes; i += 6){
6         Cx += vertexes[i];
7         Cy += vertexes[i + 1];
8         Cz += vertexes[i + 2];
9     }
10    Cx /= 36; Cy /= 36; Cz /= 36;
11
12    // 2. Traslado la figura al origen (restando el centro a
13    cada vertice)
14    for(int i = 0; i < sizeVertexes; i += 6){
15        vertexes[i] -= Cx;
16        vertexes[i + 1] -= Cy;
17        vertexes[i + 2] -= Cz;
18    }
19
20    // 3. Aplico la transformacion (multiplicando los vertices
21    por la matriz)
22    for(int i = 0; i < sizeVertexes; i += 6){
23        float originalVertex[4] = {vertexes[i], vertexes[i + 1],
24        vertexes[i + 2], 1.0f};
25
26        float transformedVertex[4];
27        matrixVectorMult(matrix, originalVertex,
28        transformedVertex);
29    }
```

```
25     vertexes[i] = transformedVertex[0];
26     vertexes[i + 1] = transformedVertex[1];
27     vertexes[i + 2] = transformedVertex[2];
28 }
29
30 // 4. Traslado la figura al centro original (sumando el
    centro a cada vertice)
31 for(int i = 0; i < sizeVertexes; i += 6){
32     vertexes[i] += Cx;
33     vertexes[i + 1] += Cy;
34     vertexes[i + 2] += Cz;
35 }
36 calculateCenter();
37 }
```

- calculateCenter(): Por último, la función calculateCenter. Su nombre, lo dice, está encargada de calcular el centro de cada figura o mejor conocido como cubie (una mini-cubito del cubo origina). Esto nos sirve para diferenciar entre camadas y hacer más fácil su rotación.

## 2.2 Clase ShaderClass

Gracias a esta clase, es que podemos configurar todos nuestros vertexShaders y fragmentShaders de manera más ordenada. Así como se explico, dicha estructura sirve para hacer toda la configuración inicial como el *Bind*, *Attach*, etc.

Procederemos a mostrar la estructura de la clase tratada:

```
1  class ShaderClass{
2      public:
3          unsigned int ID_ShaderProgram;
4          ShaderClass(const char*, const char*);
5          void Activar_ShaderProgram();
6          void Eliminar_ShaderProgram();
7          void setBool(const std::string &name, bool value) const;
8          void setInt(const std::string &name, int value) const;
9          void setFloat(const std::string &name, float value) const;
```





```
10 void setVec2(const std::string &name, const glm::vec2 &value)
    const;
11 void setVec2(const std::string &name, float x, float y) const;
12 void setVec3(const std::string &name, const glm::vec3 &value)
    const;
13 void setVec3(const std::string &name, float x, float y, float
    z) const;
14 void setVec4(const std::string &name, const glm::vec4 &value)
    const;
15 void setVec4(const std::string &name, float x, float y, float
    z, float w) const;
16 void setMat2(const std::string &name, const glm::mat2 &mat)
    const;
17 void setMat3(const std::string &name, const glm::mat3 &mat)
    const;
18 void setMat4(const std::string &name, const glm::mat4 &mat)
    const;
19 };
```

### Variables usadas:

- VertexShader y fragmentShader del cubito: Esta es la estructura que guarda este tipo de variables. Notemos, que los vertices que guardaremos en la estructura Figure, por medio de su constructor, tiene la forma de *vec3* para las posiciones y otro *vec3* para los colores. Así optimizamos y hacemos más rápido el renderizado de cada figura.

```
1 const char *vertexShaderSource = "#version 330 core\n"
2 "layout (location = 0) in vec3 aPosition;\n"
3 "layout (location = 1) in vec3 aColor;\n"
4 "out vec3 fragColor;\n"
5 "uniform mat4 model;\n"
6 "uniform mat4 view;\n"
7 "uniform mat4 projection;\n"
8 "void main()\n"
9 "{\n"
```

```
10 "    gl_Position = projection * view * model * vec4(aPosition,
11     1.0f);\n"
12 "    fragColor = aColor;\n"
13 "}\n";
14
15 const char *fragmentShaderSource = "#version 330 core\n"
16 "in vec3 fragColor;\n"
17 "out vec4 FragColor;\n"
18 "void main()\n"
19 "{\n"
20 "    FragColor = vec4(fragColor, 1.0f);\n"
21 "}\n";
```

- VertexShader y fragmentShader del rayo laser: Casi similar al anterior, estas variables solo cuenta con un *vec4* para el almacenamiento únicamente de la posición. Se encuentra de esta manera

```
1  const char *vertexShaderSource1 = "#version 330 core\n"
2      "layout (location = 0) in vec3 aPos;\n"
3      "void main()\n"
4      "{\n"
5      "    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
6      "}\n";
7  const char *fragmentShaderSource1 = "#version 330 core\n"
8      "out vec4 FragColor;\n"
9      "void main()\n"
10     "{\n"
11     "    FragColor = vec4(0.0f, 0.0f, 0.0f, 1.0f);\n"
12     "}\n";
```

## 2.3 Clase Cube

La clase Cube es de las más importantes dentro de nuestro proyecto, pues justo engloba e incluye las 2 clases anteriores para por fin mostrarlo en pantalla:



```
1  class cube {
2      public:
3          // --> For the cube
4          Figure cubes[27];
5          int cantidadCubos = 27;
6
7          // --> For movement of the cube
8          float offx, offy, offz;
9          int xMov = 0, yMov = 0, zMov = 0;
10         int movCant = 50;
11         float speed = (PI / 2) / 20;
12         float acumAngle = 0.0f;
13
14         // --> Initial scramble per cube
15         vector<string> posibleMovments = {"L", "L'", "R", "R'", "U",
16         "U'", "D", "D'", "F", "F'", "B", "B'"};
17         vector<string> scramble;
18         bool flag_initialScramble = false;
19         int actual_movment_scramble = 0;
20         bool isScrambled = false;
21
22         // --> For the solver
23         bool getSolucionActive = false;
24         bool isSolving = false;
25         bool isSolving2 = false;
26         bool isDead = false;
27         vector<string> movimientos_to_solve;
28         int number_movments = 0;
29         int actual_movment_solution = 0;
30         bool isMoving = false; // * Movimiento de camada
31
32         cube();
33         ~cube();
```



```
34      // --> Funciones
35      void renderInit();
36      void render(ShaderClass);
37
38      void rotateCamada_L(int &);
39      void rotateCamada_LI(int &);
40      ...
41
42      // * Scramble Functions
43      void moveScramble();
44
45      // * Solver Functions
46      void getSolution();
47      void moveSolution();
48
49      // * Funciones de Colision
50      bool isColliding(float, float, float);
51  };
```

### Variables publicas:

- `cubes`, `cantidadCubos` y `originalRubickVertexes`: Notemos que primero guardamos dentro de un array 27 objetos de tipo `Figura`, haciendo referencia a los 27 cubies que necesitamos para armar un cubo de rubik.
- Variables “For movement of the cube”: Como su nombre lo dice, las usamos para el movimiento del cubo tanto como figura completa y rotación de camada. Esta ultima, la controlamos mediante las variables *speed*, para saber cuanto rotaremos en cada iteración; y la variable *acumAngle*, para saber en qué momento dejar de rotar.
- Variables “Scramble and Solve”: Como se puede observar, las siguientes variables (desde *posibleMovments* hasta antes del constructor), sirven para el desarmado inicial, el cual se carga automáticamente en el constructor mediante movimientos conocidos aleatorios; y luego variables donde almacenar la solución dada por el algoritmo `Solve_rubik`, el cual se basa en el conocido método: “Fridrich”

### Constructores:



- `cube()`: Como se menciona anteriormente, utilizamos este espacio para mover inicialmente los 27 cubies en sus respectivas posiciones para hacer la forma de un cubo de rubik; por ello es que usamos variables llamadas `offset_x`, `offset_y`, `offset_z`. Y adicionalmente, se carga aleatoriamente un desarmado usando el lenguaje conocido para armar un cubo de rubik: `posibleMovments = "L", "L'", "R", "R'", "U", "U'", "D", "D'", "F", "F'", "B", "B'"`

**Métodos:** Los métodos que mencionaremos a continuación son los encargados de hacer las respectivas rotaciones del cubo, la lógica de las colisiones (cuando un rayo láser choca con uno de los cubos), y el movimiento aleatorio de los cubos por el espacio:

- `rotateCamada_X`: Resaltemos el hecho que todas las funciones que tengan la forma **rotateCamada** guardan la misma lógica; lo único que cambia es la indicación del eje en donde se desea rotar, o por ejemplo si es en sentido horario o antihorario. Por otro lado, gracias a que guardamos en esta estructura el `offset` de todos los ejes al momento de graficarlos en un comienzo, es que podemos tener la libertad de diferenciar las capas sin preocuparnos que estos se estén moviendo constantemente; por lo tanto, solo la capa seleccionada se someterá a una rotación (obviamente controlado por las variables “velocidad” (`speed`) y “acumAngle”. Y para terminar, recalculamos el centro de las nuevas posiciones a manera de reiniciar el cubo y empezar una nueva rotación sin problema.
- `isColliding`: Un método que nos permite saber si estos colisionando con un rayo láser con un cubo de rubik. La lógica usada es de calcular iterativamente la distancia entre dicho rayo láser con el cubo usando distancia euclidiana, y si cumple un mínimo de 3 unidades, lo consideramos como colisión.
- `randMove`: Esta función es el encargado de hacer que los cubos de rubik estén flotando aleatoriamente por nuestro espacio. La lógica usada para este algoritmo es conseguir posiciones aleatorias cercanas a la posición actual del cubo en  $(x,y,z)$ , y moverlos en sus respectivos vértices. Lo primordial, es que controlamos su movimiento mediante una variable que impide que se alejen demasiado (como si fuera una liga de tamaño  $X$  unida a cada cubo que no permite alejarse a más de  $X$  metros)

## 2.4 Clase `deathRay`

Por último, la clase `deathRay` o rayo láser. Dicha función tiene similitud que la clase `Figure`, pues al final es un objeto con vértices que se mueve en el espacio tras una inter-

acción del usuario (clic izquierdo); es más, veremos algunas funciones repetidas, pues son necesarias para poder visualizarlo en nuestra pantalla.

```
1  class deathRay {
2      public:
3          unsigned int VAO, VBO;
4          float vertexes[6];
5          float step;
6          glm::vec3 camPos, camDir;
7
8          deathRay(glm::vec3, glm::vec3);
9          void updateStep();
10         void renderInit();
11         void render(ShaderClass shp);
12         glm::vec3 getCenter();
13     };
```

### Variables publicas:

- VAO, VBO: De igual manera, este es un objeto independiente a la clase Figure; por ende, requiere de un elemento VAO y VBO para computarizar sus propios elementos
- vertexes[6]: Un array de tipo flotante donde guardaremos los vértices que comprenderá el rayo láser (que tendrá forma de cuadrado o rectángulo)
- camPos, camDir: Usando la lógica del tema de proyecciones, usamos estos vectores para controlar el trayecto del rayo. Calculamos en el momento la dirección en donde mira la cámara, y copiamos esos valores para sobrescribirlos en nuestra clase deathRay y moverlo en dicha dirección; esto lo encontraremos en el método updateStep().

### Métodos:

- updateStep: Esta función en cada llamada actualiza la posición del rayo láser a medida que avanza en una dirección específica. La lógica que usa para calcular la posición exacta es de calcular la dirección del rayo utilizando los primeros y últimos puntos del rayo. Se normaliza este vector de dirección para asegurar que tenga una



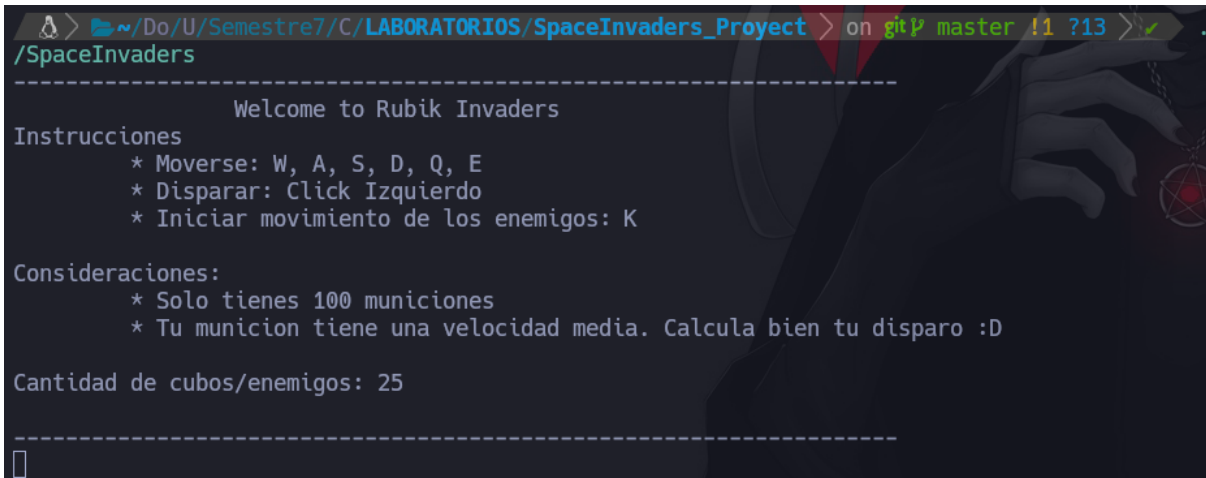
longitud de 1, y usa los puntos “startPoint” y “endPoint” para guardar el avance del rayo en la misma dirección.

- `getCenter`: Devuelve el punto central del rayo láser representado por los vértices. Tanto para X, Y, o Z, calcula la coordenada del punto medio entre los primeros y últimos puntos del rayo láser en su posición actual.

### 3 Resultados

A continuación, mostraremos distintos casos probando el proyecto con diferentes cantidades de cubos moviéndose por el espacio. Sin embargo, para una experiencia de juego, se incluyó una guía de juego junto a unas consideraciones que se debe tomar en cuenta.

#### 3.1 Guía de juego



```
> ~/Do/U/Semestre7/C/LABORATORIOS/SpaceInvaders_Project > on git master !1 ?13 > ✓  
/SpaceInvaders  
-----  
Welcome to Rubik Invaders  
Instrucciones  
* Moverse: W, A, S, D, Q, E  
* Disparar: Click Izquierdo  
* Iniciar movimiento de los enemigos: K  
  
Consideraciones:  
* Solo tienes 100 municiones  
* Tu munición tiene una velocidad media. Calcula bien tu disparo :D  
  
Cantidad de cubos/enemigos: 25  
-----  
█
```

Figure 3.1: Menú de inicio por consola/terminal



## 3.2 Imágenes de Gameplay

Número de cubos generados

- Granularidad 3

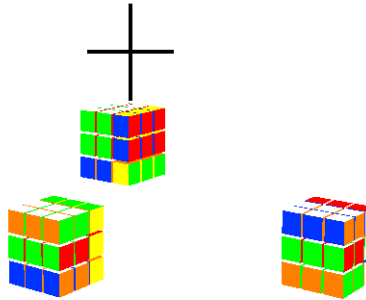


Figure 3.2: 3 cubos - Imagen 1

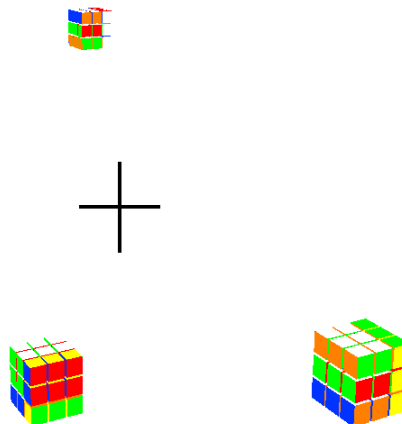


Figure 3.3: 3 cubos - Imagen 2



- Granularidad 10

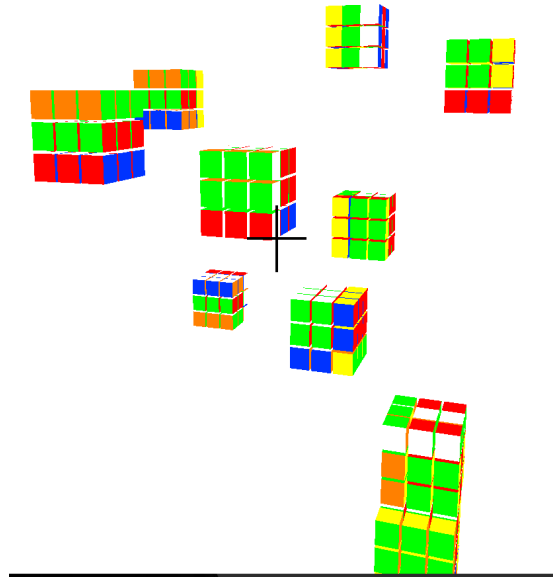


Figure 3.4: 10 cubos - Imagen 1

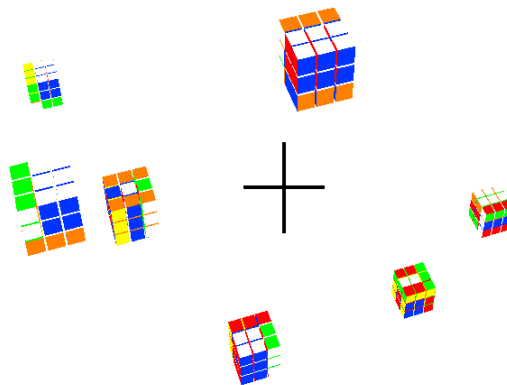


Figure 3.5: 10 cubos - Imagen 2

- Granularidad 25

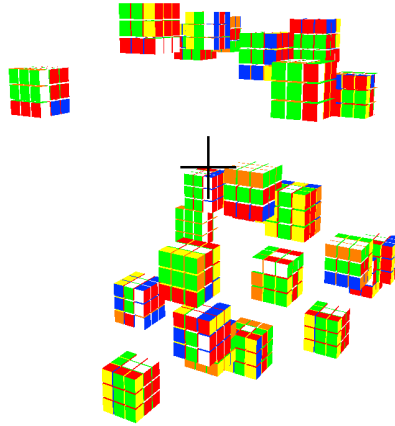


Figure 3.6: 25 cubos - Imagen 1

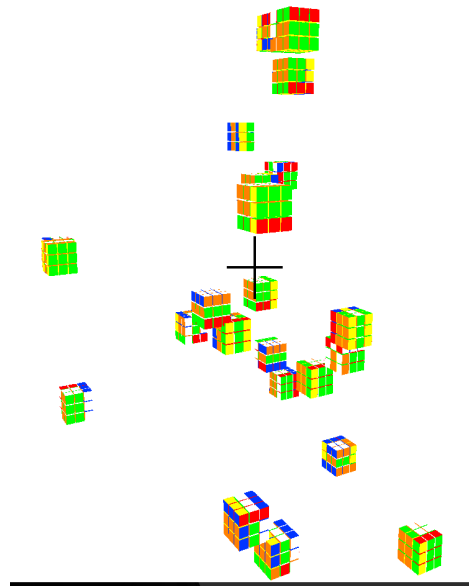


Figure 3.7: 25 cubos - Imagen 2

- Granularidad 50

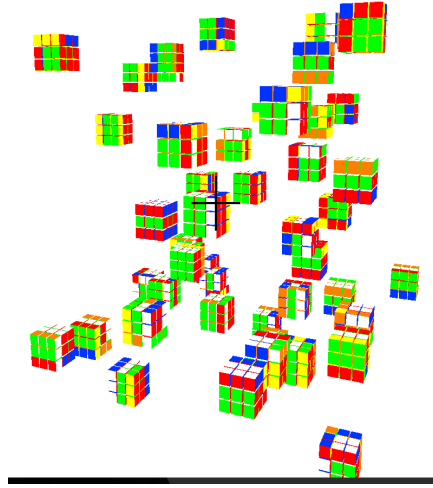


Figure 3.8: 50 cubos - Imagen 1

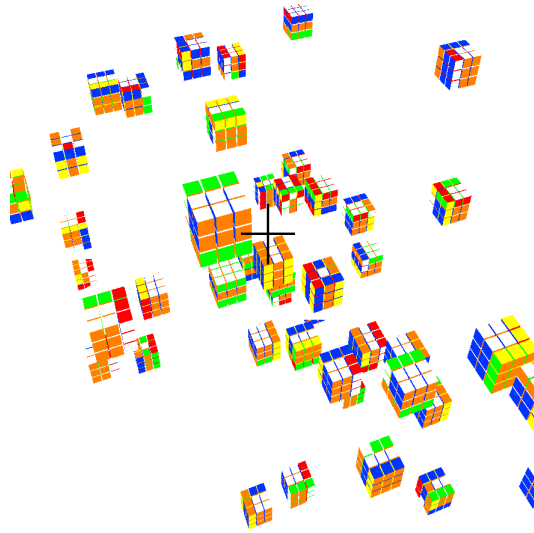


Figure 3.9: 50 cubos - Imagen 2

- Granularidad 100

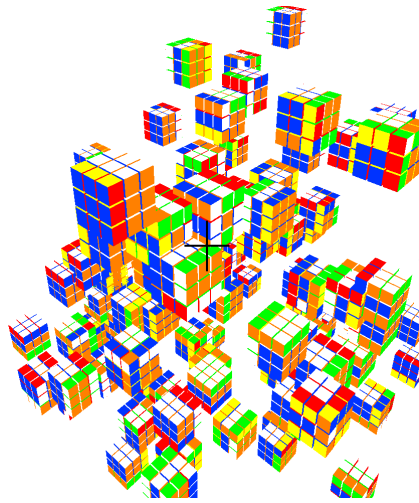


Figure 3.10: 3 cubos - Imagen 1

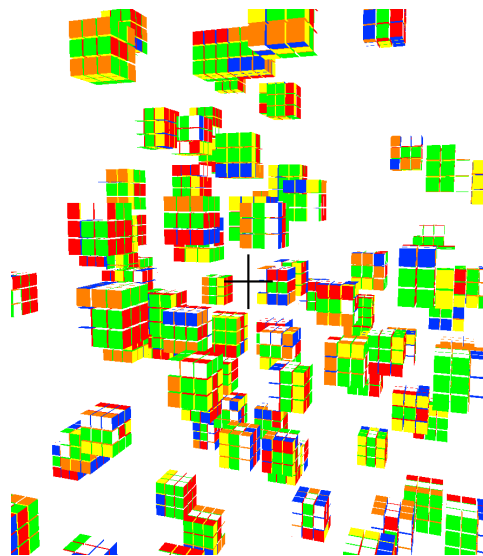


Figure 3.11: 3 cubos - Imagen 2

**Disparo** Recordemos que el disparo se efectúa dando clic izquierdo al mouse. También, que si no nos movemos, no veremos el misil, pues este sale de la dirección de donde apuntamos.

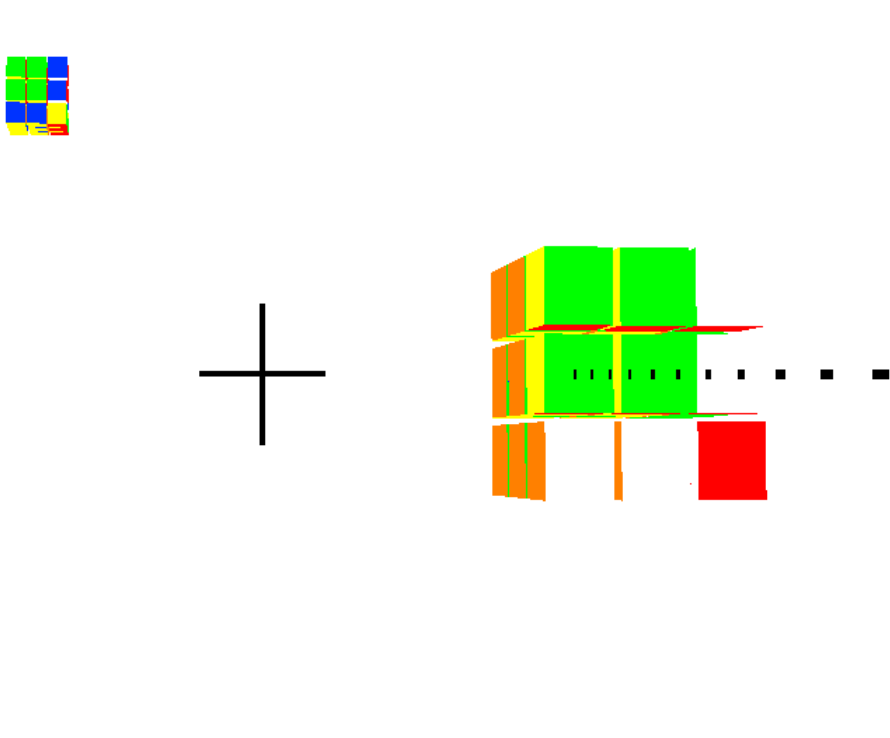


Figure 3.12: Disparo Misil

**Sin munición** Recordemos que tenemos un límite de munición de 100 unidades; pasado ello no nos permitirá disparar y nos mostrará nuestro puntaje.

```
Numero de movimientos: 4
Scramble: B'B'RD'
Movimientos: DR'BB
-----
COLISION DETECTED!... +1
COLISION DETECTED!... +1
COLISION DETECTED!... +1
COLISION DETECTED!... +1
COLISION DETECTED!... +1
COLISION DETECTED!... +1
COLISION DETECTED!... +1
Cube is dead
Cube is dead
Cube is dead
Death Ray Spawned! - Ammo: 93
Death Ray Spawned! - Ammo: 94
Death Ray Spawned! - Ammo: 95
Cube is dead
Death Ray Spawned! - Ammo: 96
Death Ray Spawned! - Ammo: 97
Death Ray Spawned! - Ammo: 98
Death Ray Spawned! - Ammo: 99
GG ... No more ammo! You killed 91 enemies
GG ... No more ammo! You killed 91 enemies
GG ... No more ammo! You killed 91 enemies
□
```

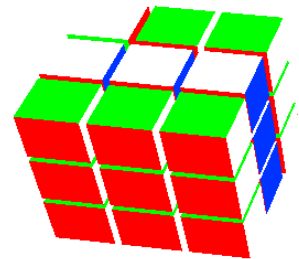


Figure 3.13: Disparo Misil