

2020

Matemática básica para el desarrollo de motores de juegos

MATEMÁTICA BÁSICA PARA JUEGOS 2D Y 3D

IGNACIO CEA FORNIES

Índice

ÍNDICE	1
PROLOGO	3
AGRADECIMIENTOS	6
INTRODUCCIÓN	7
BLOQUE 1: ÁLGEBRA ELEMENTAL EN EL MUNDO DE JUEGOS	8
INTRODUCCIÓN	9
PUNTOS Y VECTORES	10
MATRICES	19
PLANOS Y RECTAS	27
PLANO	27
RECTA	28
BLOQUE 2: PROYECCIONES Y SISTEMAS DE REFERENCIA EN JUEGOS	33
INTRODUCCIÓN	34
LA CAJA DE PROYECCIONES	35
TIPOS DE PROYECCIONES MÁS HABITUALES USADAS EN JUEGOS	37
PROYECCIÓN EN PERSPECTIVA CÓNICA ORTOGONAL (CÓNICA)	43
PROYECCIÓN SOBRE PANTALLA CUADRADA	44
PROYECCIÓN SOBRE PANTALLA PANORÁMICA ESTÁNDAR	44
PROYECCIÓN SOBRE PANTALLA PANORÁMICA GENERAL	45
PROYECCIÓN DE PUNTOS SINGULARES	45
PROYECCIÓN DE LÍNEAS CON INICIO O FINAL TRAS EL PUNTO DE VISTA	47
PROYECCIÓN DE POLÍGONOS CON PUNTOS POR DETRÁS DEL PUNTO DE VISTA	56
PROYECCIÓN EN PERSPECTIVA CILÍNDRICA ORTOGONAL (ORTOGONAL)	59
PROYECCIÓN EN PERSPECTIVA AXONOMÉTRICA ISOMÉTRICA (ISOMÉTRICA)	60
PROYECCIÓN EN PERSPECTIVA AXONOMÉTRICA CABALLERA (CABALLERA)	62
CAMBIO DE SISTEMA DE REFERENCIA	64
CAMBIO DE BASE	64
CAMBIO DE SISTEMA DE REFERENCIA	65
MATRICES DE GIRO. ÁNGULOS DE EULER	67
UN PAR DE EJEMPLOS	72
BLOQUE 3: LAS HOJAS DE FOTOGRAMAS. INTERPRETACIÓN EN 2.5D	74
INTRODUCCIÓN	75
HOJAS DE FOTOGRAMAS. “Sprite Sheets”	76
PERSPECTIVA CABALLERA 30 GRADOS APROXIMADA	78

PERSPECTIVA CABALLERA 30 GRADOS RÁPIDA	86
POSICIÓN RELATIVA EN PERSPECTIVA CABALLERA	89
ISOMÉTRICA APROXIMADA	90
PERSPECTIVA ISOMÉTRICA RÁPIDA	96
POSICIÓN RELATIVA EN ISOMÉTRICA	98
ALGORITMO BÁSICO	98
ALGORITMOS EVOLUCIONADOS	100
POSICIÓN RELATIVA EN OTRAS PERSPECTIVAS	101
<hr/>	
BLOQUE 4: DETECCIÓN DE COLISIONES EN JUEGOS	102
<hr/>	
INTRODUCCIÓN	103
RECTÁNGULOS	104
RECTÁNGULOS ORIENTADOS, UBICADOS EN UN PLANO Y ALINEADOS CON LOS EJES	104
RECTÁNGULOS ORIENTADOS Y GIRADOS UN MISMO ÁNGULO EN UN PLANO	106
RECTÁNGULOS EN DIFERENTES PLANOS	107
DEFINICIÓN DE UN RECTÁNGULO	108
PARALELEPÍPEDOS	123
<hr/>	
BLOQUE 5: ECUACIONES DE MOVIMIENTO ADAPTADAS A JUEGOS	128
<hr/>	
INTRODUCCIÓN	129
LAS ECUACIONES DE MOVIMIENTO	130
CONTROL DEL MOVIMIENTO DE LAS ENTIDADES DE UN JUEGO	135
ALGORITMO DE BRESENHAM	138
BRESENHAM 2D	138
BESENHAM 3D	141
<hr/>	
ANEXOS	147
<hr/>	
ILUSTRACIONES	148
ECUACIONES	149

Prologo

Aún recuerdo con mucho cariño y cierta nostalgia, un día, hace ya muchos años, a comienzos del otoño, cuando yo tenía entre 12 y 14 años, cuando al llegar del colegio, me encontré conectado al televisor del salón un aparato que no había visto en mi vida.

En la pantalla del televisor se podía ver un globo de color verde moviéndose dentro de un rectángulo de color azul y rebotando contra los bordes de aquel.

¿Qué es eso papá?, pregunté.

Un ordenador, respondió él.

¿Y para qué sirve?

Para muchas cosas.

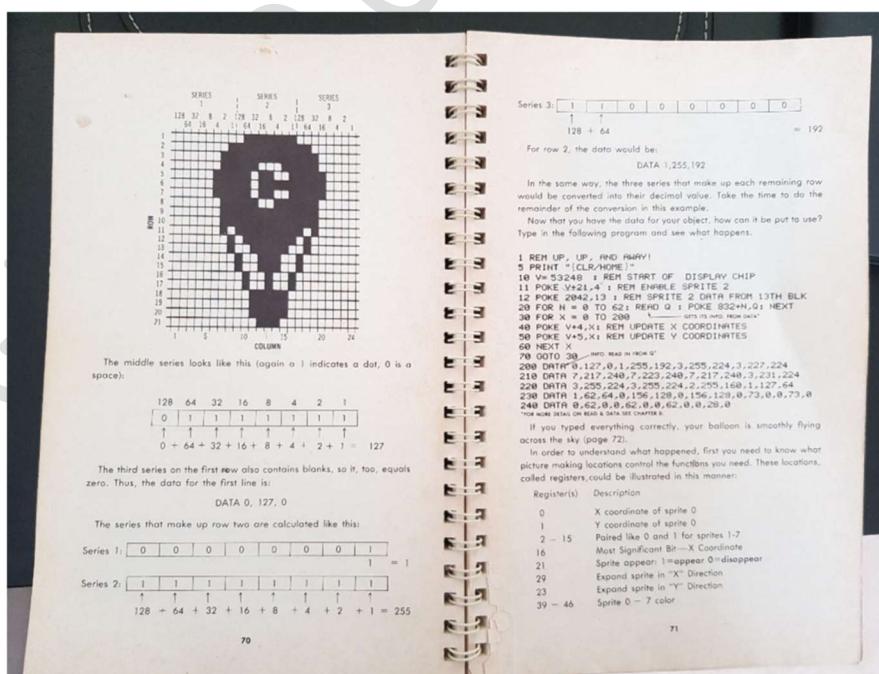
¿Eso lo has hecho tú?

Sí

¿Y cómo lo has hecho?

He escrito un programa. Unas instrucciones para que haga eso.

¡Hala!, ¿Puedo yo? ¿Me enseñas?



Mi padre nunca me enseñó. Hizo algo mucho mejor y perdurable en el tiempo. Me dio el manual de usuario y me dijo que me lo leyera. ¡Me dijo que aprendiera yo sólo!

La máquina en cuestión era un *Commodore64*®. Un ícono de la historia de los ordenadores. Con el tiempo descubrí que mi padre tuvo que pedir un préstamo para poder pagar el ordenador en cuestión y ¡que valía casi 120.000 pesetas! Toda una inversión para una familia modesta de clase media.

Quien me iba a decir que máquina y gesto iban a ser el inicio de una pasión que aún dura, que me relaja hasta el extremo, que me cabrea, que me enseña, y que me mantiene despierto y vivo. La pasión de programar y, especialmente la de programar juegos.



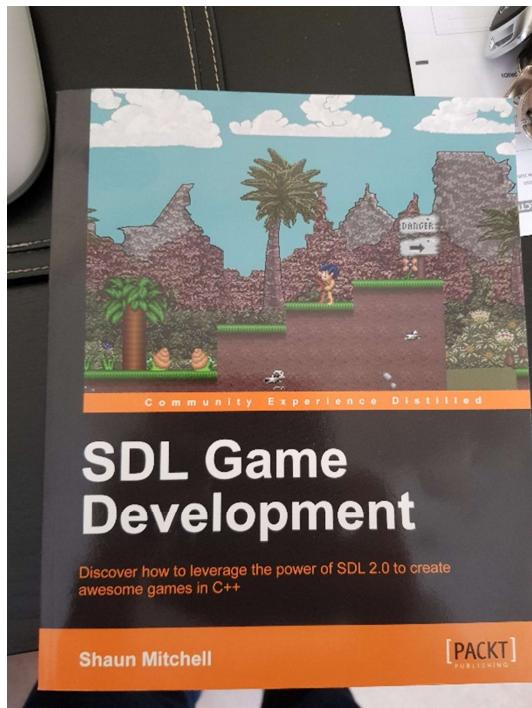
2: Los manuales que leí

A lo largo de los años aprendí Basic, código máquina, assembler, y más adelante: Pascal, C, orientación a objetos, C++, Java, Php, Python... y todo ello de una forma autodidacta. Y todo, todo, gracias a aquel gesto de mi padre.

Lo que siempre más me ha divertido programar son juegos. Su complejidad y la necesidad de tener que ejecutar todo a velocidades increíbles y sin interrupciones para conseguir fluidez, son retos que estimulan mi mente.

Soy consciente de que hoy hay mucho y muy buenos motores para casi cualquier tipo de juego que permiten centrarse sólo en la lógica de éste, dejando la complejidad técnica en manos de aquellos. Pero eso no es, al menos para mí, suficientemente estimulante. Yo prefiero centrarme en la programación del propio motor. Y En sentido contrario, soy muy malo pintando personajes y diseñando sonidos. Eso se lo dejo a otros mucho más hábiles en esa faceta. Yo me dedico a integrar en mis motores esas maravillas que están fuera de mis capacidades.

En 2015 comencé la creación en mis ratos libres, de una librería (motor si queréis) que me facilitara el diseño de juegos 2D y 3D. Quise introducirme en el dibujo de gráficos en Windows y compré para ello el libro: *SDL Game Development* de Packt Publishing. Lo estudié y empecé la construcción de mi propia librería intentando ponerla poco a poco en práctica con simples juegos copia de otros que había disfrutado o desarrollado en mi juventud.



3: Un libro muy interesante sobre librerías gráficas

Creo que, hoy, la librería es amplia. Cubre 2D y 3D y me facilita mucho el desarrollo de juegos. Hoy me cuesta más hacerme con los gráficos y los sonidos que la construcción del juego en sí.

Este año, con la llegada de la pandemia, que tan devastadores efectos está teniendo en el mundo, y el confinamiento que lo siguió, decidí empezar a escribir varios libros. Uno sobre la librería en sí, en el que aún estoy trabajando, y otro sobre algunos de los principios matemáticos que utilizo en el desarrollo de esta.

Hoy publico el segundo de ellos.

Perdonarme si tiene erratas o es denso. Lo he hecho con todo mi cariño.

Espero que lo disfrutéis.

Ignacio Cea Forniés

En Madrid, a 21 de noviembre de 2020.

Agradecimientos

A mi padre, Eduardo Cea, excelente padre y mejor maestro.

A mi madre, Quety, excelente madre y mejor consejera.

A mi mujer Paz, y a mis hijas Sandra y María, lo mejor de mi vida y que soportan sin quejarse (mucho) que pase más tiempo con el ordenador que con ellas.

IGNACIO CEA FORNIÉS

Introducción

El mundo de los juegos por ordenador requiere de la ejecución constante de muchas funciones y cálculos matemáticos. Especialmente en los juegos llamados 3D, en los que los giros, traslaciones y cambios de punto de vista son constantes. La matemática también la encontramos al mover elementos por el mundo simulado en el que transcurre la acción, o al calcular como esos elementos se ven a través de la pantalla del ordenador.

Toda esa matemática debe ser ejecutada a la mayor velocidad posible para garantizar una adecuada fluidez en el juego. Las tarjetas gráficas actuales, tremadamente potentes, ayudan mucho a la función específica de dibujar en el mundo tridimensional, siendo especialmente rápidas en el trazado de triángulos llenos de una determinada textura. Triángulos a los que, en última instancia, puede reducirse cualquier forma geométrica.

No pretendemos que este ensayo sea un compendio de toda la matemática necesaria para la construcción de complejos y potentes motores para juegos tipo *Unity*^{®1} o *Unreal*^{®2}, pero si sentar los conceptos elementales que permiten, al menos, el desarrollo básico de estos. La matemática presentada será, por tanto, suficiente para el desarrollo de motores sencillos como el que proporciona la librería *QGAMES*³ en el que se basan todos los juegos construidos por *Community Networks*.

Así, Este documento está dividido en cinco grandes bloques:

1. Un primer bloque en el que se introducen conceptos básicos del álgebra, necesarios para la comprensión del resto del documento.
2. Un segundo bloque en el que se profundiza en la representación de los elementos del juego en la pantalla y en dónde se habla de los diferentes tipos de proyecciones utilizadas de forma más habitual en juegos 2D y 3D.
3. Un tercer bloque en el que se trata en detalle las diferentes maneras de interpretar una hoja de fotogramas (sprite sheet en inglés), utilizadas para recoger habitualmente las diferentes formas que adoptan nuestros personajes en un juego (especialmente) 2D.
4. Un cuarto bloque en el que se analizan diferentes técnicas para la detección de colisiones entre objetos, algo evidentemente muy importante en cualquier juego.
5. Y un quinto y último bloque en el que se profundiza en la forma de mover los elementos de un juego de tal manera que el control sobre ellos, sobre su posición y sobre si colisionan o no, sea completa.

Este ensayo requiere de conocimientos mínimos de álgebra y programación.

¹ <https://unity.com/es>

² <https://www.unrealengine.com/en-US/>

³ By Community Networks. Para 2D y 3D. Centrados en la lógica del juego.

<https://sourceforge.net/projects/qgames-library/>

Bloque 1: Álgebra elemental en el mundo de juegos

IGNACIO CEA FORNIÉS

Introducción

La base de toda la matemática de juegos son los elementos básicos del álgebra en el espacio \mathbb{R}^3 ; esto son: puntos, vectores, líneas y planos.

Analizaremos en este bloque la forma de identificarlos y las operaciones más habituales con ellos dentro de la lógica de un juego.

Todas las fórmulas y análisis que llevaremos a cabo son válidos también en un juego de 2 dimensiones con sólo hacer 0 el valor de una de las coordenadas, habitualmente la Z.

El capítulo no pretende ser un tratado de álgebra, sino sólo sentar las bases para la mejor comprensión del resto del documento.

Puntos y vectores

Puntos y vectores se utilizan por ejemplo para saber la localización de un personaje, hacia dónde están mirando o el sentido de su movimiento.

Sea P un punto en el espacio geométrico \mathbb{R}^3 :

$$P = \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} \in \mathbb{R}^3$$

Y sea entonces \overrightarrow{OP} un vector en el espacio geométrico \mathbb{R}^3 que une el origen de referencia O con el punto P :

$$\overrightarrow{OP} = \begin{pmatrix} P_x \\ P_y \\ P_z \end{pmatrix} \in \mathbb{R}^3$$

Las coordenadas del vector \overrightarrow{OP} y del punto P son equivalentes: $\overrightarrow{OP} \equiv P$ que, aunque matemáticamente no es correcto puesto que no se pueden hacer con los puntos las mismas operaciones que con los vectores, es útil para muchas de las fórmulas y conclusiones a las que llegaremos más adelante.

Sean, por otro lado, dos puntos P_1 y P_2 , y \vec{V} el vector que une P_1 con P_2 con P_1 y P_2 diferentes de 0, y calculado como:

$$\vec{V} = \begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} = \begin{pmatrix} P_{2x} - P_{1x} \\ P_{2y} - P_{1y} \\ P_{2z} - P_{1z} \end{pmatrix} \neq P_2$$

1: Coordenadas del vector que une dos puntos

En este caso, las coordenadas del punto P_2 , extremo del vector \vec{V} , no coinciden con las del propio \vec{V} . \vec{V} nos sirve, por tanto, para marcar dirección y sentido, pero no ubicación.

Con los vectores pueden llevarse a cabo las siguientes operaciones básicas, útiles en la construcción de juegos:

- **Suma algebraica entre dos vectores:**

$$\text{Sean } \vec{V}_1 = \begin{pmatrix} V_{1x} \\ V_{1y} \\ V_{1z} \end{pmatrix} \text{ y } \vec{V}_2 = \begin{pmatrix} V_{2x} \\ V_{2y} \\ V_{2z} \end{pmatrix} \Rightarrow \vec{V}_1 \pm \vec{V}_2 = \begin{pmatrix} V_{1x} + V_{2x} \\ V_{1y} + V_{2y} \\ V_{1z} + V_{2z} \end{pmatrix}$$

Se ha visto anteriormente que las coordenadas de un punto P coinciden con las del vector \overrightarrow{OP} que une el origen de referencia con dicho punto P .

En un caso general, y haciendo uso de la operación suma, dados dos puntos P_1 y P_2 y el vector que los une \vec{V} , se puede concluir que:

$$\overrightarrow{OP_2} = \overrightarrow{OP_1} + \vec{V}$$

Y haciendo uso de la equivalencia no matemática de que $\overrightarrow{OP} \equiv P$, podríamos escribir la anterior igualdad como:

$$\mathbf{P}_2 = \mathbf{P}_1 + \vec{V}$$

2: Ecuación básica de movimiento de un punto en el espacio

Que nuevamente sin ser matemáticamente correcto, es útil en la programación de juegos y muestra la equivalencia entre puntos y vectores.

Podría servir, por ejemplo, para calcular la nueva ubicación (posición) de un personaje en la pantalla tras empujarle (fuerza vectorial).

- **Módulo de un vector**, que representa la longitud de éste:

$$\text{Sea } \vec{V} = \begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} \text{ un vector} \Rightarrow \|\vec{V}\| = \sqrt{V_x^2 + V_y^2 + V_z^2} \in \mathbb{R}$$

- **Distancia entre dos puntos**:

Dados dos puntos P_1 y P_2 , podemos calcular el vector \vec{V} que los une como la diferencia entre las coordenadas de sus extremos, entonces la longitud de ese vector equivaldrá a la distancia entre los puntos P_1 y P_2 .

- **Dirección y sentido de un vector**:

Sean dos vectores \vec{V}_1 y \vec{V}_2 . Ambos tendrán la misma dirección si las rectas que los contienen son la misma, lo cual se garantiza, según el teorema de Thales, si se garantiza también la igualdad de proporciones entre sus coordenadas ; es decir:

$$\vec{V}_1 \parallel \vec{V}_2 \Leftrightarrow \frac{V_{1x}}{V_{2x}} = \frac{V_{1y}}{V_{2y}} \text{ y } \frac{V_{1x}}{V_{2x}} = \frac{V_{1z}}{V_{2z}} \Rightarrow \text{también } \frac{V_{1y}}{V_{2y}} = \frac{V_{1z}}{V_{2z}}$$

Que equivale a decir que:

$$\vec{V}_1 \parallel \vec{V}_2 \Leftrightarrow \vec{V}_2 = k\vec{V}_1, k \neq 0 \Rightarrow k = \frac{V_{1x}}{V_{2x}} = \frac{V_{1y}}{V_{2y}} = \frac{V_{1z}}{V_{2z}}$$

Además, esos vectores tendrán el mismo sentido (apuntan hacia el mismo punto de la recta que los contiene), si los signos de sus coordenadas con el mismo; es decir:

$$\text{Sea } f(x): \mathbb{R}^3 \rightarrow \{1, 0, -1\} \text{ tal que } \begin{cases} x > 0 \Rightarrow f(x) = 1 \\ x < 0 \Rightarrow f(x) = -1 \\ x = 0 \Rightarrow f(x) = 0 \end{cases}$$

$$f(V_{1x}) = f(V_{2x}) \text{ y } f(V_{1y}) = f(V_{2y}) \text{ y } f(V_{1z}) = f(V_{2z})$$

Esto equivale a decir que tendrán el mismo sentido si $k > 0$ y el contrario si $k < 0$.

En cualquier caso, veremos más adelante que hay maneras más sencillas y rápidas de comprobar si dos vectores tienen o no la misma dirección y el mismo sentido.

- **Producto de un vector por un escalar:**

$$\text{Sea } \vec{V} = \begin{pmatrix} V_x \\ V_y \\ V_z \end{pmatrix} \text{ un vector y } k \in \mathbb{R} \Rightarrow k\vec{V} = \begin{pmatrix} kV_x \\ kV_y \\ kV_z \end{pmatrix}$$

Multiplicar un vector \vec{V} por un escalar k produce un vector \vec{V}' , k veces más largo que el anterior y de la misma dirección y sentido; es decir:

$$\vec{V}' = k\vec{V} \Rightarrow \|\vec{V}'\| = \|k\vec{V}\| = k\|\vec{V}\|$$

- **Vector unitario:**

Si multiplicamos un vector \vec{V} por un escalar igual al inverso de su módulo ($\frac{1}{\|\vec{V}\|}$), se genera un nuevo vector \vec{V}' cuyo módulo será la unidad; esto es:

$$\text{Sea } \vec{V} \text{ y } \vec{V}' = \frac{\vec{V}}{\|\vec{V}\|} \Rightarrow \|\vec{V}'\| = \left\| \frac{\vec{V}}{\|\vec{V}\|} \right\| = \frac{1}{\|\vec{V}\|} \|\vec{V}\| = 1$$

Se dice entonces que \vec{V}' es el vector unitario de \vec{V} .

- **Producto escalar entre dos vectores:**

$$\text{Sea } \vec{V}_1 = \begin{pmatrix} V_{1x} \\ V_{1y} \\ V_{1z} \end{pmatrix} \text{ y } \vec{V}_2 = \begin{pmatrix} V_{2x} \\ V_{2y} \\ V_{2z} \end{pmatrix} \Rightarrow \vec{V}_1 \cdot \vec{V}_2 = V_{1x}V_{2x} + V_{1y}V_{2y} + V_{1z}V_{2z} \in \mathbb{R}^3$$

3: Producto escalar entre dos vectores. Definición básica

O también:

$$\vec{V}_1 \cdot \vec{V}_2 = \|\vec{V}_1\| \|\vec{V}_2\| \cos \theta$$

4: Producto escalar entre dos vectores. Definición alternativa

Donde θ es el ángulo más pequeño que forman entre ellos.

Si \vec{V}_1 y \vec{V}_2 son perpendiculares, su producto escalar será 0.

El producto escalar cumple, además, las siguientes propiedades. Sean tres vectores \vec{A} , \vec{B} y \vec{C} y k un escalar, entonces:

$$\vec{A} \cdot (\vec{B} + \vec{C}) = (\vec{A} \cdot \vec{B}) + (\vec{A} \cdot \vec{C})$$

$$k(\vec{A} \cdot \vec{B}) = k\vec{A} \cdot \vec{B}$$

Y, por tanto:

$$\vec{A} \cdot \vec{A} = \|\vec{A}\|^2$$

5: Cálculo rápido del módulo al cuadrado de un vector usando el producto escalar

- **Producto vectorial entre dos vectores:**

Representado un vector perpendicular al plano formado por los otros dos⁴ y orientado según la regla del tornillo⁵:

$$\text{Sea } \vec{V}_1 = \begin{pmatrix} V_{1x} \\ V_{1y} \\ V_{1z} \end{pmatrix} \text{ y } \vec{V}_2 = \begin{pmatrix} V_{2x} \\ V_{2y} \\ V_{2z} \end{pmatrix} \Rightarrow \vec{V}_1 \times \vec{V}_2 = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \\ V_{1x} & V_{1y} & V_{1z} \\ V_{2x} & V_{2y} & V_{2z} \end{bmatrix}$$

$$\vec{V}_1 \times \vec{V}_2 = (V_{1y}V_{2z} - V_{1z}V_{2y})\vec{i} - (V_{1x}V_{2z} - V_{1z}V_{2x})\vec{j} + (V_{1x}V_{2y} - V_{1y}V_{2x})\vec{k}$$

$$\vec{V}_1 \times \vec{V}_2 = \begin{pmatrix} V_{1y}V_{2z} - V_{1z}V_{2y} \\ V_{1z}V_{2x} - V_{1x}V_{2z} \\ V_{1x}V_{2y} - V_{1y}V_{2x} \end{pmatrix} \in \mathbb{R}^3$$

6: Producto vectorial entre dos vectores. Definición básica

Y se demuestra que:

$$(\vec{V}_1 \times \vec{V}_2) \cdot \vec{V}_1 = 0$$

$$(\vec{V}_1 \times \vec{V}_2) \cdot \vec{V}_2 = 0$$

Por lo que el producto vectorial es perpendicular a los vectores que le dieron lugar, conclusión útil para muchas operaciones relacionadas con el funcionamiento de los planos.

También se demuestra que:

$$\|\vec{V}_1 \times \vec{V}_2\| = \|\vec{V}_1\| \|\vec{V}_2\| \sin \theta$$

Y que por ello $\vec{V}_1 \times \vec{V}_2$ se puede poner como:

$$\vec{V}_1 \times \vec{V}_2 = \|\vec{V}_1\| \|\vec{V}_2\| \sin \theta \vec{n}$$

7: Producto vectorial entre dos vectores. Definición alternativa

Siendo \vec{n} un vector unitario perpendicular a \vec{V}_1 y a \vec{V}_2 .

El producto escalar cumple además las siguientes propiedades. Sean tres vectores \vec{A} , \vec{B} y \vec{C} y k un escalar, entonces:

$$\vec{A} \times \vec{A} = \vec{0}$$

$$\vec{A} \times \vec{0} = \vec{0}$$

$$\vec{A} \times \vec{B} = -\vec{B} \times \vec{A}$$

$$k(\vec{A} \times \vec{B}) = k\vec{A} \times \vec{B}$$

⁴ Veremos más adelante que un plano viene representado por un vector perpendicular a él y que, por tanto, dos vectores determinan por sí solos un plano que pasa por el origen.

⁵ La regla del tornillo dice que el sentido del nuevo vector es el mismo que el del movimiento de un tornillo (ascendente o descendente) que girara desde el vector a la izquierda del operador X hacia el de la derecha.

$$\vec{A} \times (\vec{B} + \vec{C}) = (\vec{A} \times \vec{B}) + (\vec{A} \times \vec{C})$$

$$A.(B \times C) = (A \times B).C$$

- **Punto medio de un vector:**

Sea \overrightarrow{AB} el vector que une dos puntos A y B . Entonces su punto medio P será el extremo del vector \overrightarrow{OP} (según ya hemos analizado anteriormente), calculado como:

$$\overrightarrow{OP} = \overrightarrow{OA} + \frac{\|\overrightarrow{AB}\|}{2} \overrightarrow{ab} = \overrightarrow{OA} + \frac{\|\overrightarrow{AB}\|}{2} \frac{\overrightarrow{AB}}{\|\overrightarrow{AB}\|} = \overrightarrow{OA} + \frac{1}{2} \overrightarrow{AB} = \frac{1}{2} \begin{pmatrix} B_x + A_x \\ B_y + A_y \\ B_z + A_z \end{pmatrix} \equiv P$$

Donde \overrightarrow{ab} es el vector unitario de \overrightarrow{AB} .

- **Proyección de un vector en la dirección de otro:**

Sean \vec{u} y \vec{v} dos vectores que forman entre si un ángulo θ . Sean \vec{u}_1 y \vec{u}_2 dos vectores perpendiculares entre sí, tales que $\vec{u} = \vec{u}_1 + \vec{u}_2$ y verificando que $\vec{u}_2 \cdot \vec{v} = 0$ ($\vec{u}_2 \perp \vec{v}$) y que \vec{u}_1 es paralelo a \vec{v} ($\vec{u}_1 \times \vec{v} = \vec{0}$). Sea k un escalar que cumple que: $\vec{u}_1 = k\vec{v}$.

Entonces:

$$\vec{u} = \vec{u}_1 + \vec{u}_2 = k\vec{v} + \vec{u}_2$$

$$\vec{u} \cdot \vec{v} = (k\vec{v} + \vec{u}_2) \cdot \vec{v} = k(\vec{v} \cdot \vec{v}) + (\vec{u}_2 \cdot \vec{v}) = k\|\vec{v}\|^2 + 0$$

Y despejando k :

$$k = \frac{\vec{u} \cdot \vec{v}}{\|\vec{v}\|^2}$$

8: Módulo de la proyección de un vector sobre otro, usando el producto escalar

Si \vec{v} fuera, además, un vector unitario, el producto escalar entre ambos representaría el módulo de la proyección de \vec{u} sobre \vec{v} .

En *QGAMES*, se define la clase *QGAMES::Position* y su equivalente *QGAMES::Vector* (*typedef*) con los siguientes métodos implantados y que recogen todas las definiciones realizadas anteriormente:

```
/** \ingroup Game */
/** \ingroup Foundation */
/* @{ */

/**
 * @file
 * File: position.hpp \n
 * Framework: Commy Game Library (CGL) \n
 * Author: Ignacio Cea Forniés (Community Networks) \n
 * Creation Date: 01/12/2014 \n
 * Description: Defines a position in the space. \n
 * Versions: 1.0 Initial
 */

#ifndef __QGAMES_POSITION__
#define __QGAMES_POSITION__

#include <Common/definitions.hpp>
#include <Common/genalgorithms.hpp>
#include <Common/matrix.hpp>
#include <math.h>
#include <iostream>
```

```

#include <vector>

namespace QGAMES
{
    class Position;
    typedef Position Vector;
    typedef std::vector <Position> Positions;

    /** A position defines a coordinate in the space (3D possible). */
    class Position
    {
        public:
            Position ()
                : _posX (bdata0), _posY (bdata0), _posZ (bdata0)
                { }
            Position (bdata px, bdata py, bdata pz = bdata0)
                : _posX (px), _posY (py), _posZ (pz)
                { }
            Position (const std::string& p);
            Position& operator = (const Position& p)
                { _posX = p._posX; _posY = p._posY; _posZ = p._posZ;
                    return (*this); }

            bdata posX () const
                { return (_posX); }
            bdata posY () const
                { return (_posY); }
            bdata posZ () const
                { return (_posZ); }

            Position plus (const Position& p) const
                { return (Position (_posX + p._posX, _posY + p._posY, _posZ + p._posZ)); }
            Position less (const Position& p) const
                { return (Position (_posX - p._posX, _posY - p._posY, _posZ - p._posZ)); }
            Vector divide (bdata d) const
                { return ((d == bdata0)
                    ? Vector::noPoint : Vector (_posX / d, _posY / d, _posZ / d)); }
            Vector multiply (bdata d) const
                { return (Vector (_posX * d, _posY * d, _posZ * d)); }
            Vector transformWith (const Matrix3& m) const;
            bdata distanceTo (const Position& p) const
                { return (p.less (*this).module()); }
            bdata module2 () const
                { return (dotProduct (*this)); }
            bdata module () const
                { return (sqrt (module2())); }
            bdata dotProduct (const Vector& v) const
                { return ((_posX * v._posX) + (_posY * v._posY) + (_posZ * v._posZ)); }
            bdata angleWith (const Vector& v) const;
            Vector crossProduct (const Vector& v) const
                { return (Vector (((_posY * v._posZ) - (_posZ * v._posY)),
                                ((_posZ * v._posX) - (_posX * v._posZ)),
                                ((_posX * v._posY) - (_posY * v._posX)))); }

            Vector& normalize ()
                { if (*this != Vector ()) *this = *this / module (); return (*this); }

            Position rotate (const Position& e1, const Position& e2, bdata a) const;
            Position projectOver (const Position& c, const Vector& o) const;
            Position round () const
                { return (Position ((int) _posX,
                                   (bdata) ((int) _posY),
                                   (bdata) ((int) _posZ))); }

            bool closeTo (const Position& p, bdata t = bdata1) const // The tolerance...
                { return ((p - *this).module () <= t); }
            bool operator == (const Position& p) const
                { return (closeTo (p, __QGAMES_ERROR)); }
            bool operator != (const Position& p) const
                { return (!closeTo (p, __QGAMES_ERROR)); }

            Position operator + (const Position& p) const
                { return (this -> plus (p)); }
            Position& operator += (const Position& p)
                { return (*this = *this + p); }
            Position operator - (const Position& p) const
                { return (this -> less (p)); }
            Position operator - () const
                { return (*this * -1); }
    };
}

```

```

Position& operator == (const Position& p)
    { return (*this = *this - p); }
Vector operator * (bdata i) const
    { return (this -> multiply (i)); }
friend Vector operator * (bdata d, const Vector& v)
    { return (v.multiply (d)); }
friend Vector operator * (const Matrix3& m, const Vector& v)
    { return (v.transformWith (m)); }
Vector& operator *= (bdata d)
    { return (*this = *this * d); }
Vector operator / (bdata d) const
    { return (this -> divide (d)); }
Vector& operator /= (bdata d)
    { return (*this = *this / d); }

friend std::ostream& operator << (std::ostream& o, const Position& p)
{
    if (p == QGAMES::Position::_noPoint)
        o << std::string ("\"");
    else
        o << (p.posX () * __BD 100) / __BD 100.0 << "," <<
            (p.posY () * __BD 100) / __BD 100.0 << "," <<
            (p.posZ () * __BD 100) / __BD 100.0;
    return (o);
}
friend std::istream& operator >> (std::istream& i, Position& p)
    { std::string t; i >> t; p = QGAMES::Position (t); return (i); }

// Managing list of points
static Position centerOf (const Positions& pos);

private:
void adjustCoordinates ();

public:
static Position _cero;
static Position _noPoint;
static Vector _xNormal;
static Vector _yNormal;
static Vector _zNormal;

protected:
bdata _posX, _posY, _posZ;
};

#endif

// End of the file
/*@}*/

```

4: Código Position. Módulo "include"

```

#include <Common/position.hpp>
#include <Common/matrix.hpp>
#include <Common/genalgorithms.hpp>

QGAMES::Position QGAMES::Position::_cero =
    QGAMES::Position (bdata0, bdata0, bdata0);
QGAMES::Position QGAMES::Position::_noPoint =
    QGAMES::Position (__MINBDATA__, __MINBDATA__, __MINBDATA__);
QGAMES::Vector QGAMES::Vector::_xNormal =
    QGAMES::Vector ((QGAMES::bdata) 1, (QGAMES::bdata) 0, (QGAMES::bdata) 0);
QGAMES::Vector QGAMES::Vector::_yNormal =
    QGAMES::Vector ((QGAMES::bdata) 0, (QGAMES::bdata) 1, (QGAMES::bdata) 0);
QGAMES::Vector QGAMES::Vector::_zNormal =
    QGAMES::Vector ((QGAMES::bdata) 0, (QGAMES::bdata) 0, (QGAMES::bdata) 1);

// ---
QGAMES::Position::Position (const std::string& p)
    : _posX (bdata0), _posY (bdata0), _posZ (bdata0)
{
    if (p == std::string ("\""))
    {
        _posX = _posY = _posZ = __MINBDATA__;
    }
}

```

```

        else
        {
            std::vector <QGAMES::bdata> elmts = QGAMES::getElementsFromAsBDATA (p, ',', 3);
            _posX = elmts [0]; _posY = elmts [1]; _posZ = elmts [2];
        }
    }

// ---
QGAMES::Vector QGAMES::Position::transformWith (const QGAMES::Matrix3& m) const
{
    QGAMES::bdata rc[3] = { __BD_0, __BD_0, __BD_0 };
    QGAMES::bdata vc[3] = { posX (), posY (), posZ () };

    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            rc [i] += m.element (i, j) * vc [j];

    return (QGAMES::Vector (rc [0], rc [1], rc [2]));
}

// ---
QGAMES::bdata QGAMES::Position::angleWith (const QGAMES::Vector& v) const
{
    QGAMES::bdata v1M = module ();
    QGAMES::bdata v2M = v.module ();
    if (v1M == __BD_0 || v2M == __BD_0)
        return (__MAXBDATA__);
    return (acos ((float) (dotProduct (v) / (v1M * v2M))));
}

// ---
QGAMES::Position QGAMES::Position::rotate (const QGAMES::Position& e1,
    const QGAMES::Position& e2, QGAMES::bdata a) const
{
    // If no angle... no rotation...
    if (a == __BD_0)
        return (*this);

    // The axis AB: A = e1, B = e2; AB = e2 - e1;
    // n = normal vector parallel to AB passing through the origin: (e2 - e1)/|(e2 - e1)|
    // P, point to rotate = this;
    // v = [n . (P - A)] * n, because it is used twice in the formula and creating a
    variable is faster...
    // You can check the formula at: http://www.josechu.com/mates/giros_espacio_es.htm
    Vector n = (e2 - e1).normalize ();
    Vector v = (n.dotProduct (*this - e1)) * n;
    Position Pp = e1 + v + (cos (a) * ((*this - e1) - v)) + (sin (a) * (n.crossProduct
(*this - e1)));
    Pp.adjustCoordinates (); // Just because sin and cos functions returns very complex
data...
    return (Pp);
}

// ---
QGAMES::Position QGAMES::Position::projectOver (const QGAMES::Position& c, const
QGAMES::Vector& o) const
{
    // This formula is to project a point (this) on a plane.
    // The plane is represented by a point (c) and a normal vector to the plane
    (perpendicular)...
    Vector oR = o; if (oR.module () != (QGAMES::bdata) 1) oR.normalize (); // The normal
vector has to have module 1...
    return (*this - (oR * ((*this - c).dotProduct (oR))));
    // c-this is a vector linking c and this point,
    // Then the distance between the point (this) and the plane is calculated using the
    dot product.
    // (knowing that the dot product is the length of the shadow of a vector over other)
    // Then a vector parallel to the normal one but with the length of the previous
    shadow is calculated.
    // Finally the point original point (this) is moved down that distance because it is
    the same that
    // the distance to the plane...
}

// ---
QGAMES::Position QGAMES::Position::centerOf (const QGAMES::Positions& pos)
{

```

```
if (pos.size () == 0)
    return (QGAMES::Position::_noPoint);

QGAMES::bdata cx, cy , cz; cx = cy = cz = __BD 0;
for (QGAMES::Positions::const_iterator i = pos.begin (); i != pos.end (); i++)
    { cx += (*i).posX (); cy += (*i).posY (); cz += (*i).posZ (); }

return (QGAMES::Position (cx / __BD pos.size (), cy / __BD pos.size (), cz / __BD
pos.size ()));
}

// ---
void QGAMES::Position::adjustCoordinates ()
{
    _posX = adjustDecimals (_posX);
    _posY = adjustDecimals (_posY);
    _posZ = adjustDecimals (_posZ);
}
```

5: Código Position. Módulo "src"

Matrices

Las matrices son uno de los elementos más interesantes en álgebra. Y muy necesarias para el cálculo de giros y traslaciones.

Una matriz es un arreglo bidimensional de números tal que:

$$A = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \quad 1 \leq i \leq m \text{ y } 1 \leq j \leq n$$

Se dice que A es una matriz de dimensión $m \times n$, donde m es el número de filas, y n el de columnas y que se suele denotar como $A_{m \times n}$.

Si nos fijamos, cuando $m = 1$ o $n = 1$ tendremos una representación semejante a la de un vector. Por tanto, podemos decir que un vector es un caso específico de una matriz, que nuevamente, sin ser matemáticamente correcto, es útil.

Las operaciones básicas con una matriz son las siguientes:

Sean A y B dos matrices de la misma dimensión $m \times n$, y k un escalar, entonces:

$$\begin{aligned} C_{m \times n} &= A_{m \times n} + B_{m \times n} = \begin{bmatrix} a_{11} + b_{11} & \cdots & a_{1n} + b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & \cdots & a_{mn} + b_{mn} \end{bmatrix} \\ C_{m \times n} &= kA_{m \times n} = \begin{bmatrix} ka_{11} & \cdots & ka_{1n} \\ \vdots & \ddots & \vdots \\ ka_{m1} & \cdots & ka_{mn} \end{bmatrix} \end{aligned}$$

Sea ahora A una matriz de dimensión $m \times l$ y otra B de dimensión $l \times n$, entonces se define el producto entre esas dos matrices como:

$$C_{m \times n} = A_{m \times l} B_{l \times n} = \begin{bmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Y en donde:

$$c_{ij} = \sum_{k=1}^l (a_{ik} b_{kj})$$

Es importante destacar los siguientes tipos de matriz:

- **Matriz nula:** Se denomina matriz nula a aquella matriz en la que:

$$O_{m \times n} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \text{ en la que } O_{ij} = 0$$

Por tanto, se cumple que:

$$O_{m \times n} = A_{m \times l} O_{l \times n}$$

- **Matriz cuadrada:** Se denomina matriz cuadrada aquella en la que el número de filas es igual al número de columnas; es decir, es de dimensión $n \times n$.

- **Matriz identidad:** Se llama matriz identidad a aquella matriz cuadrada en la que se verifica que:

$$I_{n \times n} = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{bmatrix} \text{ en la que } \begin{cases} I_{ij} = 1 \text{ si } i = j \\ I_{ij} = 0 \text{ si } i \neq j \end{cases}$$

- **Matriz traspuesta:** Se llama matriz traspuesta a aquella en la que se intercambian los elementos fila y columna (y por tanto la dimensión). Sea $A_{m \times n}$ una matriz de dimensión $m \times n$. Entonces:

$$A^T{}_{n \times m} = \begin{bmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & \ddots & \vdots \\ a_{1m} & \cdots & a_{nm} \end{bmatrix} \text{ es decir } a_{ij}^T = a_{ji}, 1 \leq i \leq m \text{ y } 1 \leq j \leq n$$

Que cumple además las siguientes propiedades:

$$(A^T)^T = A$$

$$(A + B)^T = A^T + B^T$$

$$(AB)^T = B^T A^T$$

- **Adjunto de un elemento de una matriz cuadrada A :** Es el valor que se obtiene de:

$$\text{adj}a_{ij} = (-1)^{i+j} |A_{ij}|; 1 \leq i, j \leq n$$

Siendo A_{ij} la matriz que resulta de eliminar el elemento ij de la matriz A .

El símbolo $|A|$ denota cálculo del determinante (ver más abajo).

- **Matriz adjunta de una matriz cuadrada A :** se llama a la matriz compuesta por todos los elementos adjuntos de la matriz A .

$$\text{adj}A = \begin{bmatrix} \text{adj}a_{11} & \cdots & \text{adj}a_{1n} \\ \vdots & \ddots & \vdots \\ \text{adj}a_{n1} & \cdots & \text{adj}a_{nn} \end{bmatrix}$$

- **Determinante de una matriz cuadrada:** Llamamos determinante de A , $|A|$, cuando A es cuadrada de dimensión n , al número obtenido al sumar todos los diferentes productos de n elementos que se pueden formar con los elementos de dicha matriz, de modo que en cada producto figuren un elemento de cada distinta fila y uno de cada distinta columna.

A cada producto se le asigna el signo (+) si la permutación de los subíndices de filas es del mismo orden que la *permutación* de los subíndices de columnas, y signo (-) si son de distinto orden.

En el caso de las matrices de dos dimensiones:

$$|A| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix}$$

$$|A| = a_{11}a_{22} - a_{12}a_{21}$$

Para el caso particular de matrices tres dimensiones, que son las utilizadas en el mundo de los juegos esto se traduce en:

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

$$|A| = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13} - a_{13}a_{22}a_{31} - a_{12}a_{21}a_{33} - a_{23}a_{32}a_{11}$$

- **Matriz inversa:** Llamamos matriz inversa A^{-1} de una matriz cuadrada A , a aquella que cumple que:

$$A \times A^{-1} = I$$

Y se puede demostrar que:

$$A^{-1} = \frac{1}{|A|} (\text{adj}A)^T$$

- **Transformado de un vector:** Se llama transformado de un vector al vector resultado de multiplicarlo por una matriz llamada matriz de transformación.

$$v_{1 \times n} = A_{n \times n} u_{n \times 1}$$

$$u_{n \times 1} = A_{n \times n}^{-1} v_{1 \times n}$$

Esta fórmula es quizás, de todas las que tienen que ver con las matrices explicadas en este capítulo, la más importante en el mundo de los juegos.

Adicionalmente podríamos entrar en temas muy interesantes sobre si dada una transformación hay vectores que al transformarse permanecen igual o simplemente multiplicados por un escalar. A los primeros se les denominan auto – vectores y a los segundos auto – valores, pero ni unos ni otros son importantes en el mundo de los juegos.

Aplicando la definición a un mundo tridimensional, tendríamos:

$$\begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix} = A \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

Todos estos conceptos se aterrizzan en *QGAMES* en la clase *QGAMES::Matrix* definida (en forma de plantilla) como:

```
/** \ingroup Game */
/** \ingroup Foundation */
/* @{/}

/**
 * @file
 * File: matrix.hpp \n
 * Framework: Commy Game Library (CGL) \n
 * Author: Ignacio Cea Forniés (Community Networks) \n
 * Creation Date: 01/12/2014 \n
 * Description: Defines a 3x3 matrix and its operations. \n
 * Versions: 1.0 Initial
 */

#ifndef __QGAMES_MATRIX__
#define __QGAMES_MATRIX__
```

```

#include <Common/definitions.hpp>
#include <array>

namespace QGAMES
{
    /** To manipulate a squared matrix.
        This is a template. But what is really used in the QGAMES are usually 3x3 and 4x4
        The rows are named from 0 to S - 1. The same for the columns.
        The definition is valid except a matrix with dimension 1. */
    template <typename T, unsigned S> class SMatrix
    {
        public:
            constexpr SMatrix () ;
            constexpr SMatrix (T d)
                : _data ()
                    { SMatrix <T, S> (); for (int i = 0; i < S; i++) _data [i][i] = d; }

            constexpr SMatrix (const std::array <std::array <T, S>, S>& m);
            constexpr SMatrix (const SMatrix <T, S>& m);
            SMatrix <T, S>& operator = (const SMatrix <T, S>& m) = default;

            constexpr T element (unsigned y, unsigned x) const
                { assert (x >= 0 && x < S && y >= 0 && y < S); return (_data [y][x]); }

            void setElement (unsigned y, unsigned x, const T& d) const
                { assert (x >= 0 && x < S && y >= 0 && y < S); _data [y][x] = d; }

            constexpr T* operator [] (int n) const
                { assert (n >= 0 && n < S); return (_data [n]); }

            T* operator [] (int n)
                { assert (n >= 0 && n < S); return (_data [n]); }

            constexpr unsigned dimension () const
                { return (S); }

            // Comparation...
            constexpr bool operator == (const SMatrix <T, S>& m) const;
            constexpr bool operator != (const SMatrix <T, S>& m) const
                { return (!(*this == m)); }

            // Add...
            constexpr SMatrix <T, S> plus (const SMatrix <T, S>& m) const;
            constexpr SMatrix <T, S> operator + (const SMatrix <T, S>& m) const
                { return (plus (m)); }

            constexpr SMatrix <T, S>& operator += (const SMatrix <T, S>& m)
                { *this = *this + m; return (*this); }

            // Subtract...
            constexpr SMatrix <T, S> less (const SMatrix <T, S>& m) const;
            constexpr SMatrix <T, S> operator - (const SMatrix <T, S>& m) const
                { return (less (m)); }

            constexpr SMatrix <T, S>& operator -= (const SMatrix <T, S>& m)
                { *this = *this - m; return (*this); }

            // Multiply & Divide...
            constexpr SMatrix <T, S> multiply (const SMatrix <T, S>& m) const;
            constexpr SMatrix <T, S> operator * (const SMatrix <T, S>& m) const
                { return (multiply (m)); }

            constexpr SMatrix <T, S>& operator *= (const SMatrix <T, S>& m)
                { *this = *this * m; return (*this); }

            constexpr SMatrix <T, S> multiply (T n) const;
            constexpr SMatrix <T, S> operator * (T n) const
                { return (multiply (n)); }

            constexpr SMatrix <T, S> operator / (T n) const
                { assert (n != (T) 0); return (multiply (1 / n)); }

            constexpr SMatrix <T, S>& operator /= (T n)
                { *this = *this * n; return (*this); }

            constexpr SMatrix <T, S>& operator * (T n, const SMatrix <T, S>& m)
                { *this = *this / n; return (*this); }

            constexpr friend SMatrix <T, S> operator * (T n, const SMatrix <T, S>& m)
                { return (m * n); }

            // Output & input operators...
            friend std::ostream& operator << (std::ostream& o, const SMatrix <T, S>& m)
            {
                o << S;
                for (unsigned i = 0; i < S; i++)
                    for (unsigned j = 0; j < S; j++)
                        o << std::endl << m [i][j];
            }
    };
}

```

```

        return (o);
    }

friend std::istream& operator >> (std::istream& i, const SMatrix <T, S>& m)
{
    unsigned s = 0;

    i >> s;
    assert (S == s);

    for (unsigned i = 0; i < S; i++)
        for (unsigned j = 0; j < S; j++)
            i >> m [i][j];

    return (i);
}

// Other operations
constexpr SMatrix <T, S - 1> without (unsigned i, unsigned j) const;
constexpr SMatrix <T, S> adjoint () const;
constexpr SMatrix <T, S> transposed () const;
constexpr T determinant () const; // Recursive template...
constexpr SMatrix <T, S> inverse () const;

T _data [S][S];
};

/** When the matrix is 1 unit only
Not all funcitons are needed, just the ones used by the method determinant. */
template <typename T> class SMatrix <T, 1>
{
public:
constexpr SMatrix ()
: _data ()
{ _data [0][0] = (T) 0; }
constexpr SMatrix (T m)
: _data ()
{ _data [0][0] = m; }
constexpr SMatrix (const SMatrix <T, 1>& m)
: _data ()
{ _data [0][0] = m._data [0][0]; }
constexpr SMatrix <T, 1>& operator = (const SMatrix <T, 1>& m)
{ _data [0][0] = m._data [0][0]; }

constexpr T* operator [] (int n) const
{ assert (n == 0); return (_data [0]); }
T* operator [] (int n)
{ assert (n == 0); return (_data [0]); }

constexpr T determinant () const
{ return (_data [0][0]); }

T _data [1][1];
};

// The matrix needed for the games...
using Matrix4 = SMatrix <bdata, 4>;
using Matrix3 = SMatrix <bdata, 3>;
using Matrix2 = SMatrix <bdata, 2>;
using Matrix1 = SMatrix <bdata, 1>;
}

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S>::SMatrix ()
: _data ()
{
    for (int i = 0; i < S; i++)
        for (int j = 0; j < S; j++)
            _data [i][j] = (T) 0;
}

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S>::SMatrix (const std::array <std::array <T, S>, S>& m)
: _data ()
{

```

```

{
    SMatrix <T, S> ();
    for (int i = 0; i < S; i++)
        for (int j = 0; j < S; j++)
            _data [i][j] = m [i][j];
}

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T,S>::SMatrix (const SMatrix <T, S>& m)
: _data ()
{
    SMatrix <T, S> ();
    for (int i = 0; i < S; i++)
        for (int j = 0; j < S; j++)
            _data [i][j] = m._data [i][j];
}

// ---
template <typename T, unsigned S>
constexpr bool QGAMES::SMatrix <T, S>::operator == (const QGAMES::SMatrix <T, S>& m) const
{
    bool result = true;

    for (int i = 0; i < S && result; i++)
        for (int j = 0; j < S && result; j++)
            result &= (_data [i][j] == m._data [i][j]);

    return (result);
}

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S> QGAMES::SMatrix <T, S>::plus (const QGAMES::SMatrix <T, S>& m) const
{
    QGAMES::SMatrix<T, S> dt;

    for (int i = 0; i < S; i++)
        for (int j = 0; j < S; j++)
            dt._data [i][j] = _data [i][j] + m._data [i][j];

    return (dt);
}

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S> QGAMES::SMatrix <T, S>::less (const QGAMES::SMatrix <T, S>& m) const
{
    QGAMES::SMatrix<T, S> dt;

    for (int i = 0; i < S; i++)
        for (int j = 0; j < S; j++)
            dt._data [i][j] = _data [i][j] - m._data [i][j];

    return (dt);
}

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S> QGAMES::SMatrix <T, S>::multiply (const QGAMES::SMatrix <T, S>& m) const
{
    QGAMES::SMatrix <T, S> result;

    for (int i = 0; i < S; i++)
        for (int j = 0; j < S; j++)
            for (int k = 1; k < S; k++)
                result._data [i][j] += _data [i][k] * m._data [k][j];

    return (result);
}

```

```

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S> QGAMES::SMatrix <T, S>::multiply (T n) const
{
    QGAMES::SMatrix <T, S> result;

    for (int i = 0; i < S; i++)
        for (int j = 0; j < S; j++)
            result._data [i][j] = _data [i][j] * n;

    return (result);
}

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S - 1> QGAMES::SMatrix <T, S>::without (unsigned i,
unsigned j) const
{
    assert (i >= 0 && i < S && j >= 0 && j < S);

    QGAMES::SMatrix <T, S - 1> result;

    int iP = 0;
    int jP = 0;
    for (int iC = 0; iC < S; iC++)
    {
        if (iC != i)
        {
            jP = 0;
            for (int jC = 0; jC < S; jC++)
            {
                if (jC != j)
                {
                    result._data [iP][jP] = _data [iC][jC];
                    jP++;
                }
            }

            iP++;
        }
    }

    return (result);
}

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S> QGAMES::SMatrix <T, S>::adjoint () const
{
    QGAMES::SMatrix <T, S> result;

    for (int i = 0; i < S; i++)
        for (int j = 0; j < S; j++)
            result._data [i][j] = (((i + j) % 2) == 0) ? 1 : -1) * without (i,
j).determinant ();

    return (result);
}

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S> QGAMES::SMatrix <T, S>::transposed () const
{
    QGAMES::SMatrix <T, S> result;

    for (int i = 0; i < S; i++)
        for (int j = 0; j < S; j++)
            result._data [i][j] = _data [j][i];

    return (result);
}

// ---
template <typename T, unsigned S>
constexpr T QGAMES::SMatrix <T, S>::determinant () const
{
    T result = (T) 0;

```

```

        for (int j = 0; j < s; j++)
            result += (((0 + j) % 2) == 0) ? 1 : -1) * _data [0][j] * without (0,
j).determinant ();

        return (result);
    }

// ---
template <typename T, unsigned S>
constexpr QGAMES::SMatrix <T, S> QGAMES::SMatrix <T, S>::inverse () const
{
    T det = determinant ();
    assert (det != (T) 0);

    return ((1 / det) * adjoint ().transposed ());
}

#endif

// End of the file
/*@*/

```

6: Código Matrix

Es interesante analizar en este código cómo están definidas las diferentes operaciones. Algunas de ellas son recurrentes en términos de plantilla. Llevarlo a otro lenguaje que no sea C++, no es del todo intuitivo.

Planos y rectas

Plano

Un plano p puede ser definido como el conjunto de puntos P del espacio que cumplen:

$$P \in p \Leftrightarrow \overrightarrow{AP} \perp \vec{n}$$

Siendo \vec{n} un vector que representa al plano y ortogonal a éste (vector director) y A un punto cualquiera contenido en el plano p .

Por tanto, la ecuación del plano se podría poner como:

$$P \in p \Leftrightarrow \overrightarrow{AP} \cdot \vec{n} = 0$$

9: Ecuación general del plano

Razonemos algunos elementos importantes dentro del plano:

- **Sistema de referencia ortonormal contenido en un plano:**

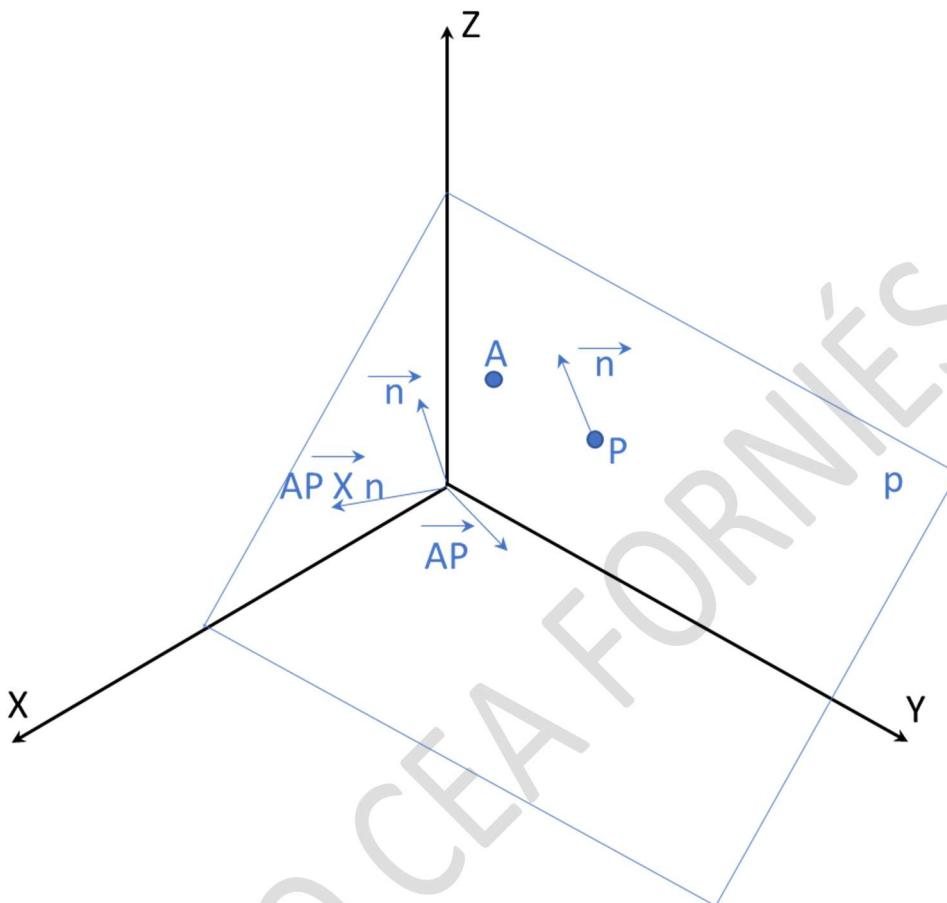
Dado que, por definición, en un plano, \overrightarrow{AP} es siempre perpendicular a \vec{n} , entonces $\overrightarrow{AP} \times \vec{n}$ también por definición será un vector contenido en el plano y además también perpendicular a ambos, según hemos visto anteriormente.

El módulo de ese vector perpendicular es:

$$\|\overrightarrow{AP} \times \vec{n}\| = \|\overrightarrow{AP}\| \|\vec{n}\| \sin \theta = \|\overrightarrow{AP}\| \|\vec{n}\|$$

Y si además \vec{n} ya fuera un vector unitario, entonces el módulo del vector perpendicular sería el mismo que el del vector \overrightarrow{AP} . Luego \overrightarrow{AP} , $\overrightarrow{AP} \times \vec{n}$ y \vec{n} constituirían un sistema de referencia ortogonal en cualquier punto P del plano p .

El sistema Luego $\frac{\overrightarrow{AP}}{\|\overrightarrow{AP}\|}, \frac{\overrightarrow{AP} \times \vec{n}}{\|\overrightarrow{AP}\|}$ y \vec{n} es además ortonormal, pues además de ser perpendiculares entre sí, tendrían todos ellos módulo 1.



7: El plano y sus vectores representativos

Recta

Las rectas r pueden ser definidas como el conjunto de puntos P que cumplen:

$$P \in r \Leftrightarrow \overrightarrow{AP} \parallel \vec{n}$$

Siendo \vec{n} un vector en la dirección de la recta (llamado vector director) y A un punto contenido en ella. Por tanto, la ecuación general de una recta se puede obtener a partir de:

$$P \in r \Leftrightarrow \overrightarrow{AP} \times \vec{n} = \vec{0}$$

10: Ecuación general de la recta

- **Punto más cercano de una recta a un plano:**

Dada una recta r , el punto más cercano de ella a otro Q es aquel punto P de la recta que verifica que:

$$\overrightarrow{QP} \cdot \vec{n} = 0$$

El módulo del vector \overrightarrow{QP} , sería la distancia buscada:

$$d(Q, r) = \|\overrightarrow{PQ}\|$$

Veremos más adelante una forma sencilla de resolverlo, pero antes avancemos en otros conceptos.

- **Intersección entre una recta y un plano:**

La intersección entre una recta (definida por el vector director \vec{n}_1 y el punto A) y un plano (definido por el vector director \vec{n}_2 y el punto B), será el punto P de ambas que cumpla:

$$\overrightarrow{AP} \parallel \vec{n}_1 \text{ y } \overrightarrow{BP} \perp \vec{n}_2$$

Es decir:

$$\overrightarrow{AP} \times \vec{n}_1 = \vec{0} \text{ y } \overrightarrow{BP} \cdot \vec{n}_2 = 0$$

Sistema de ecuaciones de 3 ecuaciones con 3 incógnitas, compatible y determinado⁶, siempre y cuando \vec{n}_1 y \vec{n}_2 no sean perpendiculares entre si (la recta es paralela al plano); es decir siempre y cuando $\vec{n}_1 \cdot \vec{n}_2 <> 0$.

En determinadas circunstancias, este cálculo puede simplificarse.

- **Intersección de un plano y una recta perpendicular a él:**

Imaginemos que la recta de intersección fuera perpendicular al propio plano. Es decir:

$$\vec{n}_1 = \vec{n}_2 = \vec{n}.$$

Sea el vector \overrightarrow{AB} el que une un punto conocido de la recta (A) con un punto conocido del plano (B); entonces, y según hemos visto anteriormente, $\overrightarrow{AB} \cdot \vec{n}$ (si \vec{n} es unitario) representa el módulo de la proyección de \overrightarrow{AB} sobre la recta de intersección y

$$\overrightarrow{OP} (\equiv P) = (\overrightarrow{AB} \cdot \vec{n})\vec{n} + \overrightarrow{OA}$$

11: Punto de intersección de plano con una recta perpendicular a él

el punto de intersección buscado.

- **Rotación del punto B de un plano alrededor un eje $A - A'$:**

Coja papel y lápiz en este punto para “dibujar” y entender mejor las explicaciones que se van a dar, pues vamos a razonar la ecuación de giro de un punto alrededor de un eje, algo tremadamente utilizado en el mundo de los juegos.

En ese mismo caso: $\overrightarrow{OB} - ((\overrightarrow{AB} \cdot \vec{n})\vec{n} + \overrightarrow{OA}) = \overrightarrow{AB} - (\overrightarrow{AB} \cdot \vec{n})\vec{n}$ es un vector contenido en un plano paralelo al que contiene a \vec{P} y a \vec{B} , perpendicular por tanto a \vec{n} que va en la dirección de \vec{P} a \vec{B} , con inicio en el origen de coordenadas, y cuyo módulo

⁶ Más adelante se analizará una fórmula más sencilla para determinar la intersección entre recta y plano.

sería, según toda la reflexión llevada a cabo previamente, $\|\overrightarrow{AB}\| \sin \theta$, igual a la distancia de \overrightarrow{PB} ; es decir:

$$\|\overrightarrow{PB}\| = \|\overrightarrow{AB}\| \sin \theta$$

Donde θ es el ángulo que forman entre si los vectores \overrightarrow{AB} y \vec{n} .

Y en ese caso, sea $\vec{n} \times \overrightarrow{AB}$ un vector perpendicular al anterior y con el mismo módulo, tal y como se vio también anteriormente, dado que \vec{n} es unitario. Vector que, además estará contenido en el plano.

Por tanto, el vector \vec{n} , el vector $\overrightarrow{AB} - (\overrightarrow{AB} \cdot \vec{n})\vec{n}$, y el vector $\vec{n} \times \overrightarrow{AB}$ constituyen en sí mismo un sistema de referencia, situado en un plano paralelo al que contiene al vector \overrightarrow{PB} pero pasando por el origen, y en el que el vector \overrightarrow{PB} (que es el que queremos girar) tendrá de módulo 1 (cambio de escala). ¿Bien hasta aquí?

Si girásemos ese vector, de módulo 1 en ese sistema de referencia, un ángulo φ alrededor del origen, las coordenadas del nuevo vector en ese mismo sistema de coordenadas serían: $(\cos \varphi, \sin \varphi)$ cuyo módulo sigue siendo 1, obviamente. Y poniéndolo en la escala original sus coordenadas serían:

$$\overrightarrow{PB} \text{ girado} = (\overrightarrow{AB} - (\overrightarrow{AB} \cdot \vec{n})\vec{n}) \cos \varphi + (\vec{n} \times \overrightarrow{AB}) \sin \varphi$$

El punto girado obtenido estará referenciado al origen, por lo que para llevarlo al plano original basta alcanzar el punto P de corte de la recta con el eje, por tanto:

$$\begin{aligned}\vec{n} &= \frac{\overrightarrow{AA'}}{\|\overrightarrow{AA'}\|} \text{unitario} \\ \vec{v} &= (\overrightarrow{AB} \cdot \vec{n})\vec{n}\end{aligned}$$

$$\overrightarrow{OB'} (\equiv B') = \vec{v} + \overrightarrow{OA} (\equiv A) + (\overrightarrow{AB} - \vec{v}) \cos \varphi + (\vec{n} \times \overrightarrow{AB}) \sin \varphi$$

12: Rotación de un punto B alrededor de un eje determinado por el vector n y el punto A

La rotación de formas más complejas alrededor de un eje consistiría, por tanto, en la rotación de sus puntos más representativos, lo que equivaldría a repetir la fórmula anterior n veces: 1 por punto.

- **Proyectar un punto C sobre un plano p representado por un vector n y un punto B:**

Consiste en, simplemente calcular el punto de intersección entre una recta paralela a \vec{n} (perpendicular por tanto al plano p) y que pasara por C con el plano p ; o sea:

$$\overrightarrow{OC'} (\equiv C') = \overrightarrow{OC} (\equiv C) - (\overrightarrow{CB} \cdot \vec{n})\vec{n}$$

13: Proyección de un punto en un plano

Proyectar una figura geométrica sobre un plano consistiría simplemente en proyectar cada uno de los principales puntos que lo representan. Es evidente que la proyección de esa figura no tiene por qué conservar el aspecto de la original.

- **Distancia de un punto B a una recta definida por un punto C y un vector director n:**

Siempre y cuando \vec{n} sea normal (módulo 1), nos podemos apoyar en el anterior punto. Si proyectamos el punto C en uno B' , sobre el plano definido por B y \vec{n} , la distancia entre B y B' coincide con la distancia del punto B a la recta.

Es decir:

$$d = \|\overrightarrow{CC'}\| = \|(\overrightarrow{CB} \cdot \vec{n})\vec{n}\|$$

14: Distancia de un punto a una recta

- **Intersección de una recta con un plano:**

Rehagamos un poco la fórmula vista anteriormente.

Sea un plano p , definido por un vector normal \vec{n} y un punto P . El plano estaría formado por todos los puntos X que cumplen que:

$$p \equiv \overrightarrow{(P - X)} \cdot \vec{n} = 0$$

O, puesto de manera cartesiana:

$$(P_x - X_x)n_x + (P_y - X_y)n_y + (P_z - X_z)n_z = 0$$

$$X_x n_x + X_y n_y + X_z n_z = P_x n_x + P_y n_y + P_z n_z$$

Sea r la recta que pasa por dos puntos A y B . La recta estaría formada por todos los puntos X que cumplen que:

$$r \equiv X = \overrightarrow{OX} = \overrightarrow{OA} + \overrightarrow{(B - A)}t, t \in \mathbb{R}$$

O bien extendiéndolo:

$$X = \begin{cases} X_x = A_x + (B_x - A_x)t \\ X_y = A_y + (B_y - A_y)t \\ X_z = A_z + (B_z - A_z)t \end{cases}$$

Por tanto, el punto de intersección estará en dónde las coordenadas se igualan; esto es:

$$[A_x + (B_x - A_x)t]n_x + [A_y + (B_y - A_y)t]n_y + [A_z + (B_z - A_z)t]n_z = P_x n_x + P_y n_y + P_z n_z$$

Despejando t:

$$t = \frac{(P_x - A_x)n_x + (P_y - A_y)n_y + (P_z - A_z)n_z}{(B_x - A_x)n_x + (B_y - A_y)n_y + (B_z - A_z)n_z} = \frac{\vec{n} \cdot \overrightarrow{AP}}{\vec{n} \cdot \overrightarrow{AB}}$$

Y, en consecuencia, las coordenadas del punto de corte serían:

$$X = \begin{cases} X_x = A_x + (B_x - A_x) \frac{\vec{n} \cdot \overrightarrow{AP}}{\vec{n} \cdot \overrightarrow{AB}} \\ X_y = A_y + (B_y - A_y) \frac{\vec{n} \cdot \overrightarrow{AP}}{\vec{n} \cdot \overrightarrow{AB}} \\ X_z = A_z + (B_z - A_z) \frac{\vec{n} \cdot \overrightarrow{AP}}{\vec{n} \cdot \overrightarrow{AB}} \end{cases}$$

15: Coordenadas del punto de corte entre una recta y un plano

IGNACIO CEA FORNIES

Bloque 2: Proyecciones y Sistemas de referencia en juegos

IGNACIO CEA FORNIÉS

Introducción

La función quizás más elemental de un juego es la de representar un escenario lo más realista e inmersivo posible, en el que el jugador o jugadores interaccionan con los diferentes elementos que allí se representan.

En este bloque aprenderemos como llevar a cabo la transformación de la realidad en figuras geométricas de la pantalla para todo tipo de juegos, 2D y 3D.

La caja de proyecciones

Supongamos que ese mundo en el que transcurre cualquier tipo juego fuera siempre tridimensional y que lo observáramos a través de una caja oscura atada a nuestra cabeza, con un cristal al fondo. Una caja opaca (salvo en ese fondo) que nos eliminara cualquier detalle periférico y que nos obligara a concentrarnos sólo en lo que viéramos a través del cristal.

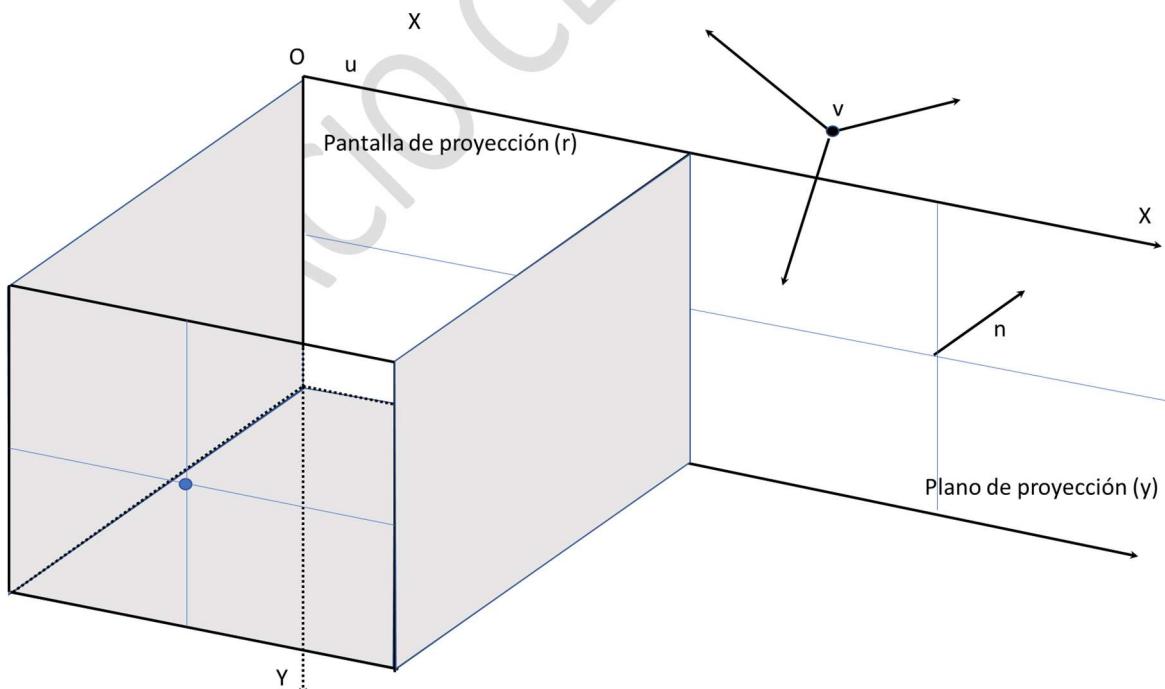
Al mover la cabeza, la caja se movería con ella y en consecuencia cambiaría lo que vemos a través del cristal ubicado en el fondo de ésta. Cada punto del mundo real observable tendría siempre un reflejo en ese cristal. A nuestros efectos, lo veríamos “proyectado” en ella.

Tomemos un sistema de coordenadas fijo, con respecto al que pudiera ubicarse cualquier objeto de la realidad (ortonormal⁷). Llamemos a ese sistema, sistema de coordenadas universal (u) o simplemente **sistema universal**.

Tomemos un sistema de coordenadas tridimensional móvil y ubiquémoslo arbitrariamente en la esquina superior del cristal de la caja (también ortonormal). Elijamos el eje \overrightarrow{OX} positivo hacia la derecha, el eje \overrightarrow{OY} positivo hacia abajo, y el eje \overrightarrow{OZ} positivo hacia el fondo. Llamémosle sistema de coordenadas de proyección (v) o, abreviado, **sistema de proyección**.

Dentro de ese sistema de proyección, definamos el plano $\vec{n} = (0,0,1)$ que pasa por el punto $O = (0,0,0)$, como el **plano de proyección** (y) y que coincidiría entonces, en ubicación, con el cristal de nuestra caja.

El ancho y alto del cristal por el que vemos la realidad sería entonces el tamaño de la zona visible de ese plano de proyección. Llámese **pantalla de proyección** (r) a esa zona.



8: La caja de proyección

⁷ Ortonormal: Los tres vectores que lo definen forman ángulos rectos entre si y tienen de módulo la unidad.

Intentemos ahora trasladar todo lo explicado a un mundo virtual, el de un juego.

Imaginemos que nuestra caja fueran unas gafas de realidad virtual. El cristal del fondo de la caja, serían las pantallas de proyección.

En este ejemplo, el plano de proyección coincide con la(s) pantalla de proyección. Obvio. Pero si no utilizáramos unas gafas de realidad virtual, sino un ordenador, una tablet o un móvil para representar la realidad, eso no ocurriría. La pantalla de proyección no se mueve. Ni la cabeza tampoco. Y es el ratón o el teclado lo que nos sirve para mover el sistema de proyección v con respecto al sistema universal u (y con él, el plano de proyección y).

Se ha elegido a propósito que el sistema de proyección (y) tenga el mismo origen y dirección en los ejes \overrightarrow{OX} y \overrightarrow{OY} que la pantalla de un ordenador. Y así, la acción anterior de trasladar los puntos del plano de proyección a la pantalla de del ordenador (r) sería inmediata, simplemente descartando la coordenada z , que, por otro lado, si todo hubiera sido bien calculado debería ser 0 respecto al sistema de referencia de proyección.

Los pasos por seguir, entonces, para representar la realidad de un juego, toda vez que se mueva el ratón o se maneja el teclado serían 4 pasos tan “simples” como:

- Calcular las coordenadas de la realidad (que están siempre referenciadas al sistema universal) con respecto al sistema de proyección v . Esto es una operación de cambio de coordenadas o de sistema de referencia.
- Proyectar entonces esas nuevas coordenadas en el plano de proyección y .
- Trasladar las coordenadas proyectadas a la pantalla de proyección r . O lo que equivale, simplificadamente, a coger el punto proyectado y prescindir de la coordenada z (que, en cualquier caso, tras realizar todas las anteriores operaciones debería ser 0).
- Dibujar los puntos calculados de la pantalla de proyección.

Existe mucha álgebra definida para hacer cambios de coordenadas de un sistema de referencia a otro. Pero, antes de entrar en ello, analicemos un poco más los diferentes tipos de proyección plana⁸ (porque la pantalla lo es) que suelen considerarse en un juego.

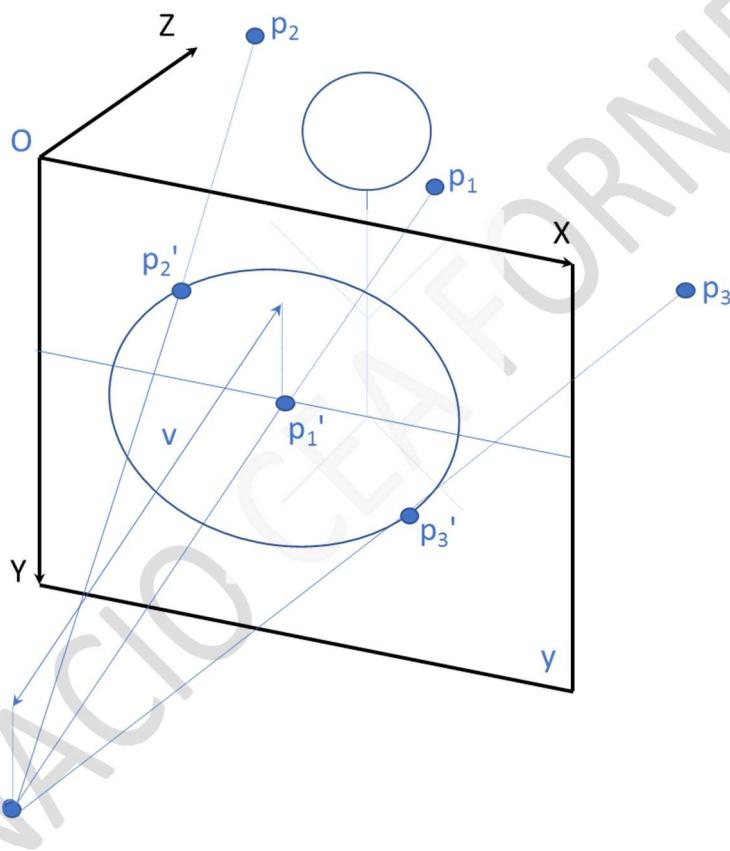
⁸ Existen otros tipos de proyección no planas, como las que se utilizan, por ejemplo, para representar el globo terráqueo.

Tipos de proyecciones más habituales usadas en juegos

Básicamente hay dos grandes tipos:

- **Proyección cónica:** El punto proyectado (p') es la intersección sobre el plano de proyección de una recta que une el punto de vista (o) situado a cierta distancia (v) del plano de proyección, con el punto (p).

Todos los posibles rayos así trazados formarían en última instancia un cono en el que su base sería la pantalla de proyección y su vértice el punto o . Vamos a suponer un cono ortogonal, es decir: en el que su altura es perpendicular al plano de proyección. Según la ilustración siguiente la altura de dicho cono coincidiría con v .



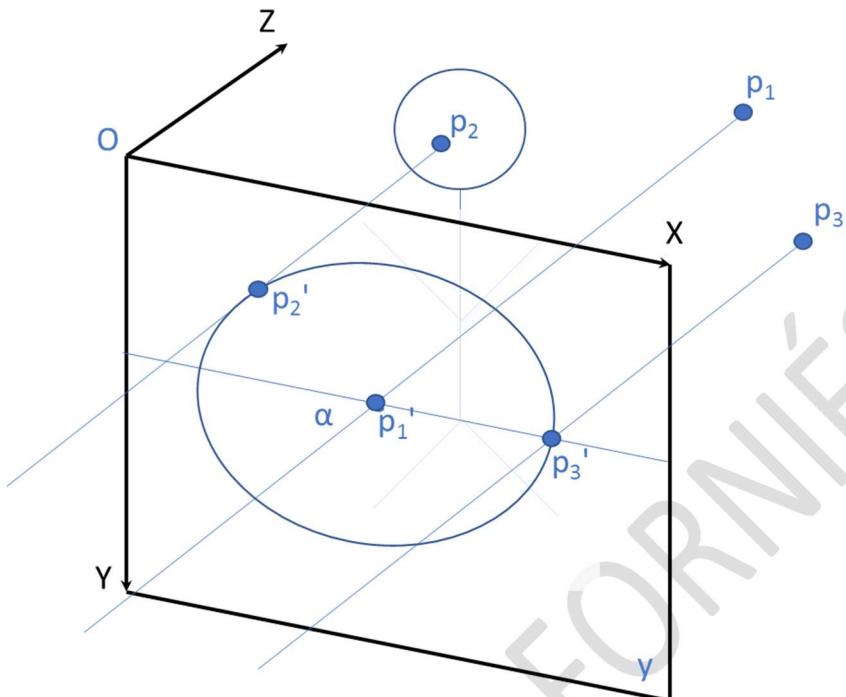
9: Proyección cónica

La proyección cónica produce una mayor sensación de realidad, pues sus supuestos se acercan más a cómo trabaja la vista humana. Cualquier elemento situado delante del plano de proyección y , sin embargo, tendrá un reflejo en este más grande de lo que es en realidad.

- **Proyección cilíndrica:** El punto proyectado (p'), es la intersección sobre el plano de proyección de una recta que forma un determinado ángulo constante (α) con el plano de proyección y que pasa por el punto p .

Todos los posibles rayos formarían en última instancia un cilindro infinito en el que su base sería la pantalla de proyección formando un ángulo α con ella. Si ese ángulo fuera

de 90 grados, la proyección se denominaría **cilíndrica ortogonal** y **cilíndrica oblicua** en otro caso.



10: Proyección cilíndrica

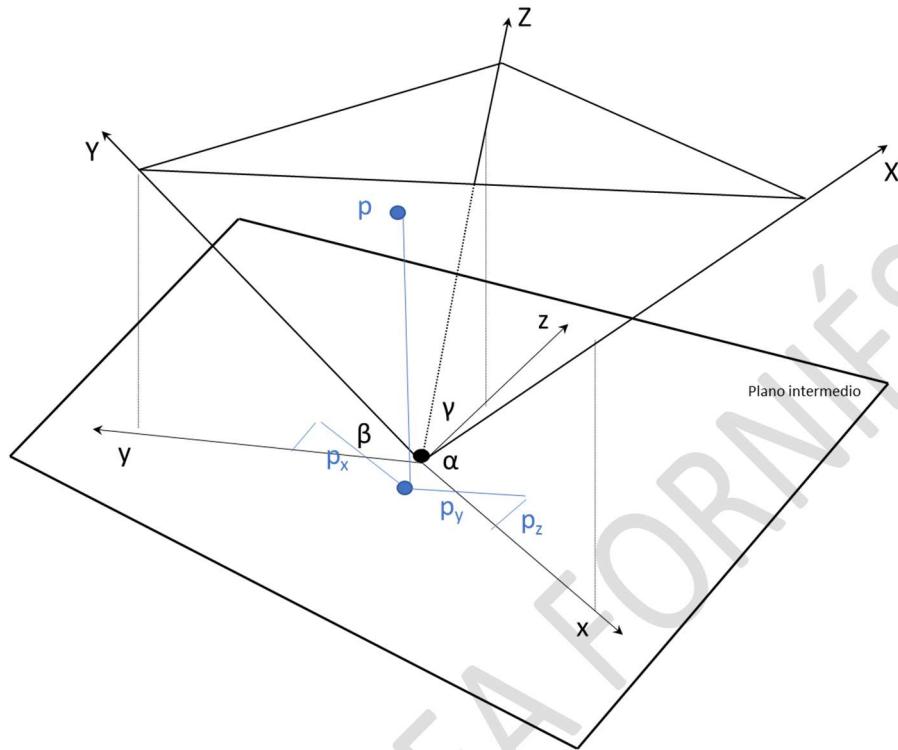
El mundo de los juegos 3D se basa siempre en la utilización de la proyección cónica ortogonal, mientras que el mundo de los juegos 2D se suele basar mayoritariamente en la cilíndrica ortogonal.

En este último caso, es necesario tener presente que el efecto de profundidad no se consigue de manera directa. Una imagen como la de la figura anterior situada más lejos que la mostrada, se proyectaría de la misma manera que la primera sobre el plano de proyección y , lo que no permitiría distinguir cuál de las dos está más cerca o más lejos del observador.

Para conseguir cierto efecto de profundidad podríamos reducir la escala de los elementos conforme estos se alejan del plano de proyección para que cuando sean proyectados aparezcan más pequeños. Este pequeño truco se complica si, además, esos elementos estuvieran en movimiento. Tendríamos además que hacer que los más alejados se movieran en los planos paralelos al plano de proyección más lentos que los que estuvieran más cerca para crear una cierta sensación de profundidad.

La proyección cilíndrica, más estos pequeños trucos, es suficiente para muchos juegos, pero no para todos. ¿Existe alguna manera de conseguir efectos de profundidad sin necesidad de recurrir a la perspectiva cónica 3D fáciles de calcular e implementar?

Imaginemos que proyectáramos los ejes de nuestro sistema de coordenadas v sobre un plano, al que llamaremos intermedio, que formara un ángulo α con el eje \overrightarrow{OX} , β con el eje \overrightarrow{OY} y γ con el eje \overrightarrow{OZ} , tal y como se muestra en la figura.



11: Proyección axonométrica

Cualquier otro punto se podría trasladar a ese plano a partir de sus distancias trasladadas sobre los ejes proyectados. Sea p' :

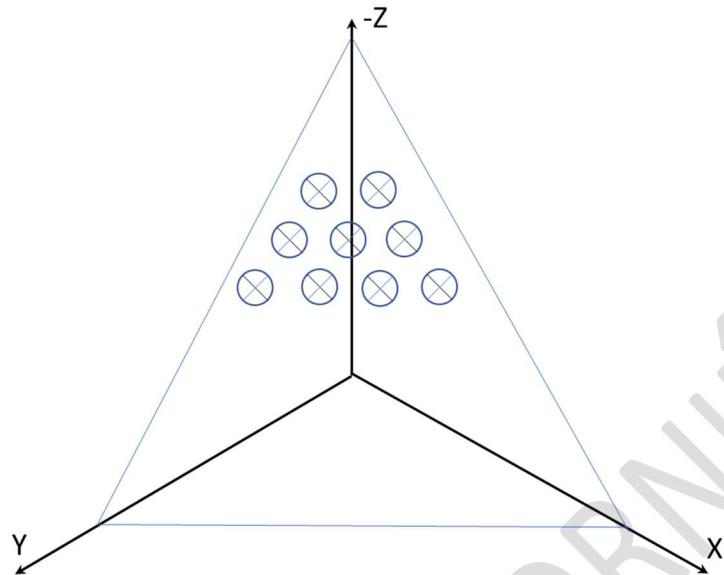
$$p' = (x \cos \alpha, y \cos \beta, z \cos \gamma)$$

A los valores de $\cos \alpha$, $\cos \beta$ y $\cos \gamma$ se les denomina factores de reducción (sobre la realidad), dado los ángulos estarían siempre entre 0 y 90 y su coseno es siempre menor que 1.

No importa la manera de llegar a ello para nuestra reflexión, pero imaginemos que esos porcentajes de reducción fueran iguales a 0.816 y que los ángulos formados entre si por los ejes proyectados fueran 120 grados.

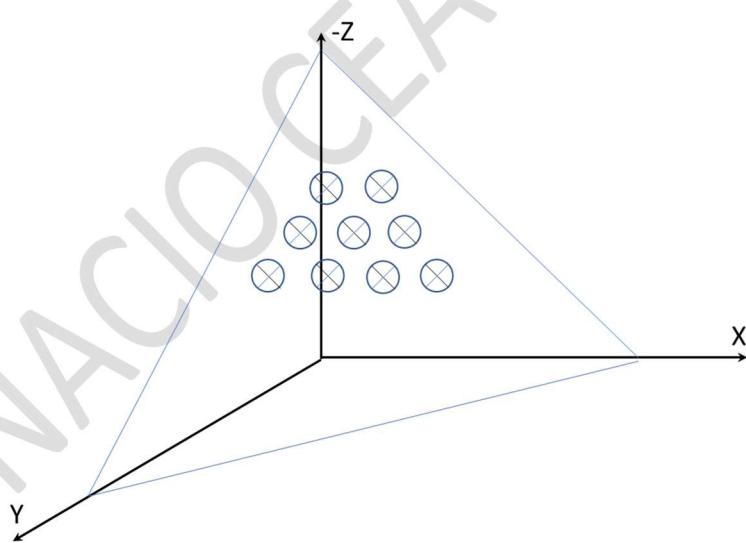
Cojamos ahora lo proyectado de esa manera y proyectémoslo cilíndricamente sobre nuestro plano de proyección y . Supongamos que esa proyección cilíndrica fuera ortogonal. La

proyección obtenida se denomina **proyección axonométrica isonómica**⁹ (isométrica para abreviar).



12: Proyección isométrica

Si, por el contrario, eligiéramos una proyección cilíndrica oblicua, obtendríamos una **proyección axonométrica caballera** (caballera para abreviar).



13: Proyección caballera

En la proyección isométrica la realidad conserva sus proporciones relativas y permite hacerse una idea cierta de la ubicación de unos objetos con respecto a otros. En la proyección caballera la realidad no conserva sus proporciones relativas, pero sigue siendo fácil hacerse una idea de la ubicación de unos objetos con respecto a otros.

Ninguna de las dos es tan cercana a la realidad como la cónica, pues no se tiene plenamente la sensación de profundidad que aporta aquella, pero es suficiente para muchas circunstancias

⁹ Hay muchos otros tipos: Dimétrico, trimétrico, etc. Nótese que, además, se ha cambiado la orientación de los ejes, para simplificar razonamientos futuros. Ahora -z es hacia arriba.

sobre todo cuando la realidad se observa de manera “estática” desde fuera de la escena en donde aquella transcurre. A ambas se las conoce a menudo (y así se hará también en este ensayo) como proyecciones 2.5D.

A lo largo del documento profundizaremos en los tipos de proyecciones comentados: **Cónica, ortogonal, isométrico y caballera**.

En los juegos encontramos ejemplos de todos los tipos mencionados:

Fornite®, en perspectiva 3D cónica:



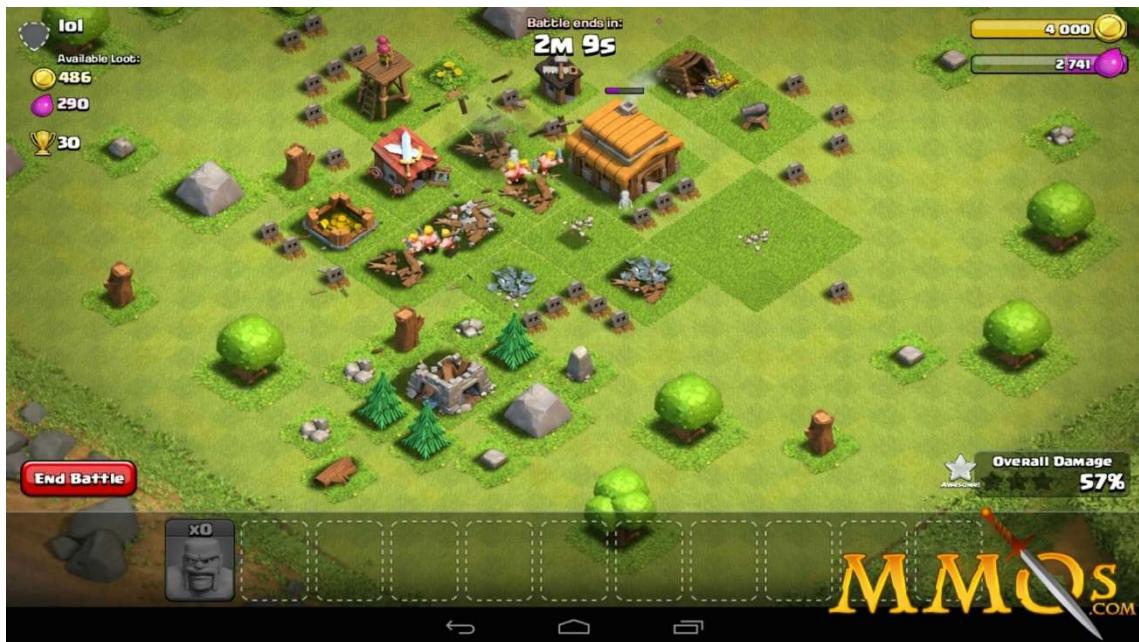
14: Fortnite®. Perspectiva cónica

Mario Bros® , en perspectiva 2D ortogonal:



15: Mario Bros®. Perspectiva ortogonal

Clash of clans®, en perspectiva 2.5D isométrica:



16: Clash of Clans®. Perspectiva isométrica

Entombed® para Commodore 64, en perspectiva 2.5D caballera. Esta perspectiva ha caído en cierto desuso en los juegos actuales.

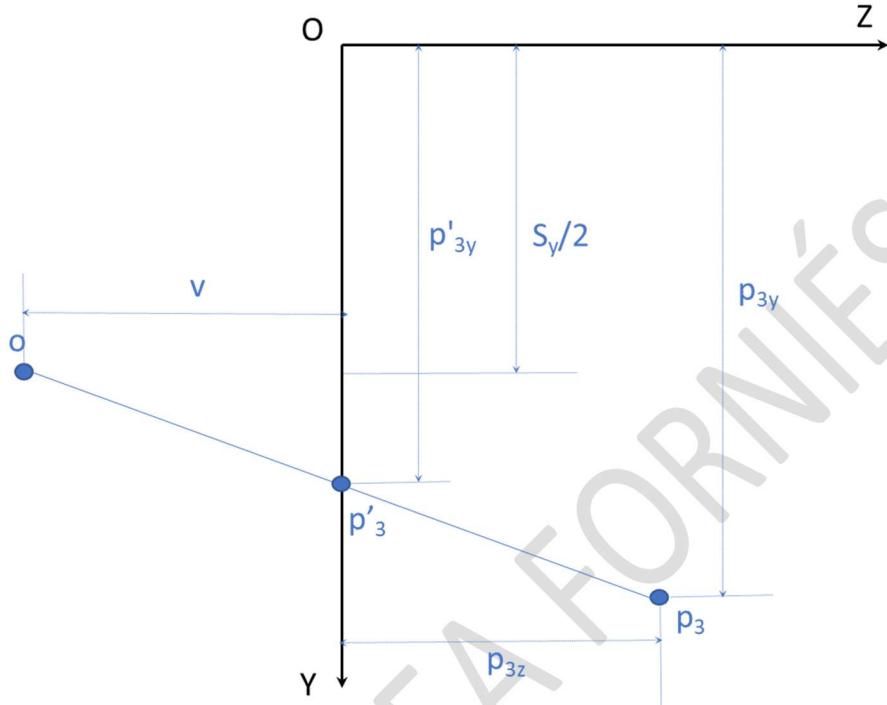


17: Entombed®. Perspectiva caballera

Vamos a profundizar en cada una de ellas:

Proyección en perspectiva cónica ortogonal (cónica)

¿Cómo calculamos el punto equivalente en la pantalla de proyección en una proyección cónica? Volvamos a la ilustración 5 y observémosla de lado. Así:



18: Proyección cónica. Cálculo en la pantalla de proyección

El eje \overrightarrow{OX} sería perpendicular al plano de la hoja.

Como ejemplo de cálculo, hemos cogido punto p_3 .

Sea v la distancia que existe del punto de vista al plano de proyección y , y S_y la altura de la pantalla de proyección.

Sea, por tanto, $\frac{S_y}{2}$ el centro en el eje \overrightarrow{OY} de la pantalla de proyección que coincide con la altura a la que se sitúa el punto de vista o .

Podemos entonces establecer la siguiente relación por triángulos semejantes:

$$\frac{p'_{3y} - \frac{S_y}{2}}{v} = \frac{p_{3y} - \frac{S_y}{2}}{v + p_{3z}}$$

Y despejando p'_{3y} :

$$p'_{3y} = v \frac{p_{3y} - \frac{S_y}{2}}{v + p_{3z}} + \frac{S_y}{2} = \frac{vp_{3y} + \frac{S_y}{2}p_{3z}}{v + p_{3z}}$$

De igual manera, observándolo desde arriba (en este caso el que sería perpendicular al plano de la hoja sería el eje \overrightarrow{OY}), obtendríamos el valor de la coordenada en x:

$$p'_{3x} = v \frac{p_{3x} - \frac{S_x}{2}}{v + p_{3z}} + \frac{S_x}{2} = \frac{vp_{3x} + \frac{S_x}{2}p_{3z}}{v + p_{3z}}$$

Generalizando entonces a cualquier punto del espacio, sus coordenadas de proyección sobre y serían:

$$p' = \frac{1}{v + p_z} \left(vp_x + \frac{S_x}{2}p_z, vp_y + \frac{S_y}{2}p_z \right)$$

En donde se puede observar que conforme se aleja el punto (aumenta en consecuencia el valor de z), su proyección se acerca al centro de la pantalla; es decir, al punto de visión:

$$\lim_{p_z \rightarrow \infty} p' = \left(\frac{S_x}{2}, \frac{S_y}{2} \right)$$

Veamos algunos casos particulares, considerando tamaños de pantalla habituales:

Proyección sobre pantalla cuadrada

La pantalla de proyección tiene una relación 1:1 entre sus dimensiones de ancho y alto.

Entonces $S_x = S_y = S$ y hagamos, por convención, $v = \frac{S}{2}$, para establecer una distancia propicia desde el que observar la pantalla. Así:

$$p' = \left(\frac{vp_x + \frac{S_x}{2}p_z}{v + p_z}, \frac{vp_y + \frac{S_y}{2}p_z}{v + p_z} \right) = \frac{S}{2} \left(\frac{p_x + p_z}{\frac{S}{2} + p_z}, \frac{p_y + p_z}{\frac{S}{2} + p_z} \right)$$

$$p' = \frac{S}{S + 2p_z} (p_x + p_z, p_y + p_z)$$

16: Coordenadas del punto proyectado en perspectiva cónica sobre una pantalla cuadrada

Dónde S es el ancho (o el alto de la pantalla).

Proyección sobre pantalla panorámica estándar

La pantalla de proyección tiene una relación es 16:9 entre sus dimensiones de ancho y alto.

Entonces $S_x = \frac{16}{9}S_y$ y hagamos, nuevamente por convención, $v = \frac{S_x}{2}$ (tomando como referencia la distancia más grande) para establecer una distancia propicia desde la que observar la pantalla. Así:

$$S_y = \frac{9}{16}S_x$$

$$p' = \left(\frac{vp_x + \frac{S_x}{2}p_z}{v + p_z}, \frac{vp_y + \frac{S_y}{2}p_z}{v + p_z} \right) = \frac{S_x}{2} \left(\frac{p_x + p_z}{\frac{S_x}{2} + p_z}, \frac{p_y + \frac{9}{16}p_z}{\frac{S_x}{2} + p_z} \right)$$

$$\mathbf{p}' = \frac{s_x}{s_x + 2p_z} \left(\mathbf{p}_x + \mathbf{p}_z, \mathbf{p}_y + \frac{9}{16} \mathbf{p}_z \right)$$

17: Coordenadas del punto proyectado en perspectiva cónica sobre una pantalla panorámica

Proyección sobre pantalla panorámica general

Podemos generalizar a cualquier tipo de dimensión, en función del ancho de la pantalla.

Sea:

- r_a es la relación de aspecto de la pantalla y
- w el ancho de la pantalla.

Entonces:

$$\mathbf{p}' = \frac{w}{w + 2p_z} \left(\mathbf{p}_x + \mathbf{p}_z, \mathbf{p}_y + \frac{1}{r_a} \mathbf{p}_z \right)$$

18: Coordenadas del punto proyectado en perspectiva cónica sobre una pantalla de cualquier relación de aspecto

Proyección de puntos singulares

Retomemos la ecuación general de la proyección cónica calculada inicialmente:

$$\mathbf{p}' = \left(\frac{vp_x + \frac{S_x}{2} p_z}{v + p_z}, \frac{vp_y + \frac{S_y}{2} p_z}{v + p_z} \right)$$

Vamos a estudiar algunas situaciones especiales.

Supongamos, por ejemplo, que:

$$p_z = 0$$

Entonces:

$$\mathbf{p}' = (p_x, p_y)$$

Lo que quiere decir que, cuando el punto está ubicado en el mismo plano de proyección y , sus coordenadas proyectadas coinciden con él mismo, lo cual es lógico, pero sirve para validar en cierta medida los cálculos realizados.

Por otro lado, si:

$$p_z = -v$$

Lo que significa que el punto está situado en el mismo plano que el punto de vista, la proyección presenta una singularidad; es decir, no hay intersección posible con el plano de proyección, al ser el denominador de los cálculos igual a cero.

Además, si:

$$p_z \rightarrow \infty$$

Entonces:

$$p' = \left(\frac{S_x}{2}, \frac{S_y}{2} \right)$$

Lo que implica que cuanto más lejos está el punto del plano de proyección, su imagen más se acercará al centro de la pantalla que es justo el lugar en el que se está proyectando el punto de vista, como ya hemos visto anteriormente.

Pero ¿Qué pasa si p_z fuera negativo pero menor que $-v$?

Un primer efecto lo podemos observar en la última fórmula. Si $p_z \rightarrow -\infty$, el punto proyectado será el mismo que el anterior cuando tendía a $+\infty$, lo cual no parece tener mucho sentido.

Parece, por ello, que la fórmula general descrita no tiene mucho sentido cuando los valores de p_z están por detrás del punto de vista. Parece claro que si el punto está por detrás del punto de vista no tendríamos que proyectarlo en la pantalla.

¿Y Qué ocurre con valores entre 0 y $-v$ de p_z ? La proyección seguirá teniendo sentido dentro del plano de proyección, pero llegará un momento en que ésta no aparezca dentro de la pantalla de proyección. Esto ocurrirá a partir de los valores que hacen que:

$$\frac{vp_x + \frac{S_x}{2} p_z}{v + p_z} < 0$$

$$\text{si } p_x < \frac{S_x}{2}$$

Dado que la proyección de los puntos se haría tiendiendo hacia 0, o bien:

$$\frac{vp_x + \frac{S_x}{2} p_z}{v + p_z} > S_x$$

$$\text{si } p_x > \frac{S_x}{2}$$

Dado que la proyección de los puntos se haría tiendiendo hacia el límite de la pantalla (recordemos que la coordenada x de proyección del punto de vista es justo $\frac{S_x}{2}$).

O sea:

$$vp_x + \frac{S_x}{2} p_z < 0$$

$$p_z < -\frac{2v}{S_x} p_x$$

$$\text{si } p_x < \frac{S_x}{2}$$

O:

$$\frac{vp_x + \frac{S_x}{2} p_z}{v + p_z} > S_x$$

$$p_z < -\frac{2v}{S_x} (S_x - p_x)$$

$$\text{si } p_x > \frac{S_x}{2}$$

Resumiendo, el punto es visible cuando la coordenada z es negativa pero mayor que $-v$ siempre y cuando:

$$p_z \in \left(-v, -\frac{2v}{S_x} p_x \right), \text{ si } p_x < \frac{S_x}{2}$$

$$\text{o } p_z \in \left(-v, -\frac{2v}{S_x} (S_x - p_x) \right), \text{ si } p_x > \frac{S_x}{2}$$

Podríamos proceder de igual manera con el eje Y, obteniendo que el punto proyectado es visible siempre y cuando:

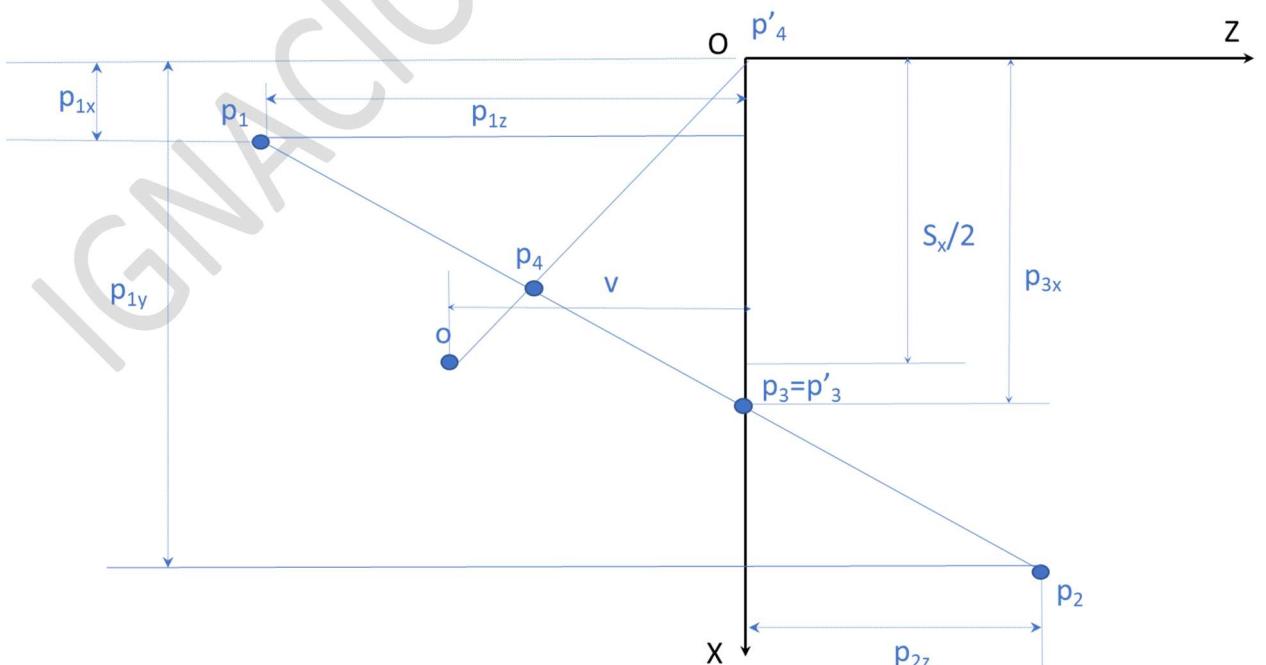
$$p_z \in \left(-v, -\frac{2v}{S_y} p_y \right), \text{ si } p_y < \frac{S_y}{2}$$

$$\text{o } p_z \in \left(-v, -\frac{2v}{S_y} (S_y - p_y) \right), \text{ si } p_y > \frac{S_y}{2}$$

Estas fórmulas nos vendrán bien más adelante.

Proyección de líneas con inicio o final tras el punto de vista

Pero ¿Qué sucedería si lo que estuviéramos proyectando fuera no un punto sino una recta, en la que uno de los puntos estuviera potencialmente dentro de la pantalla, esto es, que $p_z \geq -v$ y el otro no? ¿Cómo sería la proyección de ésta?:



19: Proyección de una recta en cónica con principio por detrás del punto de vista

La ecuación de la recta que une los dos puntos, en función de un parámetro t , sería:

$$l \equiv \forall p = p_1 + \overrightarrow{p_1 p_2} t$$

O bien, por coordenadas:

$$l \equiv \forall p \begin{cases} p_x = p_{1x} + (p_{2x} - p_{1x})t \\ p_y = p_{1y} + (p_{2y} - p_{1y})t \\ p_z = p_{1z} + (p_{2z} - p_{1z})t \end{cases} \forall t \in [0,1]$$

Dónde p_1 es un punto siempre por detrás del punto de visión es decir que $p_{1z} < -v$, y t varía entre 0 y 1 para obtener cualquiera de los puntos de la línea.

Sus puntos proyectados sobre el plano de proyección serían, según las ecuaciones vistas anteriormente:

$$p' = \left(\frac{vp_x + \frac{S_x}{2} p_z}{v + p_z}, \frac{vp_y + \frac{S_y}{2} p_z}{v + p_z} \right)$$

Introduciendo el valor de las coordenadas de la recta l , tendremos en función del parámetro t la imagen proyectada de todos los puntos de la recta:

$$p' = \left(\frac{v[p_{1x} + (p_{2x} - p_{1x})t] + \frac{S_x}{2}[p_{1z} + (p_{2z} - p_{1z})t]}{v + [p_{1z} + (p_{2z} - p_{1z})t]}, \frac{v[p_{1y} + (p_{2y} - p_{1y})t] + \frac{S_y}{2}[p_{1z} + (p_{2z} - p_{1z})t]}{v + [p_{1z} + (p_{2z} - p_{1z})t]} \right)$$

El punto último que debería verse sobre la pantalla de proyección r sería el primero en el que o bien la coordenada x' de proyección es 0 o bien lo es y' (siempre y cuando la recta se dirija hacia la parte izquierda o superior de la pantalla. Es decir:

O:

$$\frac{v[p_{1x} + (p_{2x} - p_{1x})t_1] + \frac{S_x}{2}[p_{1z} + (p_{2z} - p_{1z})t_1]}{v + [p_{1z} + (p_{2z} - p_{1z})t_1]} = 0$$

O:

$$\frac{v[p_{1y} + (p_{2y} - p_{1y})t_3] + \frac{S_y}{2}[p_{1z} + (p_{2z} - p_{1z})t_3]}{v + [p_{1z} + (p_{2z} - p_{1z})t_3]} = 0$$

Despejando t_1 , en la primera:

$$v[p_{1x} + (p_{2x} - p_{1x})t_1] + \frac{S_x}{2}[p_{1z} + (p_{2z} - p_{1z})t_1] = 0$$

$$t_1 \left[v(p_{2x} - p_{1x}) + \frac{S_x}{2}(p_{2z} - p_{1z}) \right] = -vp_{1x} - \frac{S_x}{2}p_{1z}$$

$$t_1 = \frac{-vp_{1x} - \frac{S_x}{2}p_{1z}}{(p_{2x} - p_{1x}) + \frac{S_x}{2}(p_{2z} - p_{1z})}$$

Y despejando t en la segunda, de manera equivalente:

$$t_3 = \frac{-vp_{1y} - \frac{S_y}{2}p_{1z}}{(p_{2y} - p_{1y}) + \frac{S_y}{2}(p_{2z} - p_{1z})}$$

E introduciendo el valor obtenido de t en un caso o el otro en el valor p_z de la recta:

$$p_{zt1} = p_{1z} + (p_{2y} - p_{1y})t_1$$

O:

$$p_{zt3} = p_{1z} + (p_{2y} - p_{1y})t_3$$

Aquel que produzca un valor de p_z menor, pero por encima del valor mínimo de p_z será el valor de t que represente el último punto visible de la recta y, por tanto, con una coordenada proyectada x o $y = 0$. Sustituyendo ese valor de t en la ecuación de la recta obtendremos el punto límite de la recta. No deberíamos utilizar por tanto más el p_1 original, sino este nuevo punto.

En definitiva, es necesario describir nuestra primera versión de un algoritmo para pintar proyecciones de rectas con algún punto detrás del punto de vista de la siguiente manera:

- Verificar que al menos uno de los puntos de la recta tiene una coordenada $z \leq -v$ (en nuestro caso más típico hemos fijado ese valor como $=-\frac{w}{2}$, siendo w el ancho). Si es así, la recta es potencialmente visible (dependerá de si la proyección de dichos puntos cae o no dentro de la pantalla de proyección r)
- Si los dos puntos tienen coordenadas z por detrás del punto de vista, la recta no será visible. No habrá puntos proyectados sobre la pantalla r . Por tanto, no es necesario pintarla.
- Si los dos puntos tienen coordenadas por delante del punto de vista, la recta es, en principio, potencialmente visible y se calculan normalmente sus puntos proyectados y se dibuja una línea entre ambos.
- De otra manera:
 - Calcular por puntos p_{zt} y p_{zt} , según las ecuaciones de arriba y tras haber calculado t_1 y t_3 .
 - Si p_{zt1} es menor que p_{zt1} la proyección de la recta abandona antes el eje X que el Y , por lo que el punto de proyección límite sería:

$$p' = \left(0, \frac{vp_y + \frac{S_y}{2}p_z}{v + p_z} \right)$$

En donde:

$$p_x = p_{1x} + (p_{2x} - p_{1x})t_1$$

$$p_y = p_{1y} + (p_{2y} - p_{1y})t_1$$

$$p_z = p_{1z} + (p_{2y} - p_{1y})t_1$$

- En otro caso, abandona antes el eje Y que el X , por lo que el punto de proyección límite tendría de coordenadas:

$$p' = \left(\frac{vp_x + \frac{S_x}{2} p_z}{v + p_z}, 0 \right)$$

Y en donde:

$$p_x = p_{1x} + (p_{2x} - p_{1x})t_3$$

$$p_y = p_{1y} + (p_{2y} - p_{1y})t_3$$

$$p_z = p_{1z} + (p_{2z} - p_{1z})t_3$$

- Por tanto, deberíamos sustituir el p_1 inicial por uno u otro para, a partir de ellos, generar la proyección deseada en el límite de la pantalla.

Pero, hemos comentado que estas ecuaciones son sólo válidas cuando la recta proyectada se dirige hacia la parte izquierda y hacia la parte superior de la pantalla de proyección, ya que los valores límite de X y Y son en ambos casos 0.

¿Cómo se trataría la situación en las otras posibilidades?

Por ejemplo, si la recta se dirigiera o hacia abajo o hacia la derecha, los valores límites de x y y proyectados sería resultado de las ecuaciones:

$$\frac{v[p_{1x} + (p_{2x} - p_{1x})t_2] + \frac{S_x}{2}[p_{1z} + (p_{2z} - p_{1z})t_2]}{v + [p_{1z} + (p_{2z} - p_{1z})t_2]} = S_x$$

$$\frac{v[p_{1y} + (p_{2y} - p_{1y})t_4] + \frac{S_y}{2}[p_{1z} + (p_{2z} - p_{1z})t_4]}{v + [p_{1z} + (p_{2z} - p_{1z})t_4]} = S_y$$

Despejando respectivamente t_2 y t_4 , tenemos:

$$v[p_{1x} + (p_{2x} - p_{1x})t_2] + \frac{S_x}{2}[p_{1z} + (p_{2z} - p_{1z})t_2] = S_x[v + [p_{1z} + (p_{2z} - p_{1z})t_2]]$$

$$\left[v(p_{2x} - p_{1x}) - \frac{S_x}{2}(p_{2z} - p_{1z}) \right] t_2 = v(S_x - p_{1x}) + \frac{S_x}{2}p_{1z}$$

$$t_2 = \frac{v(S_x - p_{1x}) + \frac{S_x}{2}p_{1z}}{v(p_{2x} - p_{1x}) - \frac{S_x}{2}(p_{2z} - p_{1z})}$$

Y:

$$t_4 = \frac{v(S_y - p_{1y}) + \frac{S_y}{2}p_{1z}}{v(p_{2y} - p_{1y}) - \frac{S_y}{2}(p_{2z} - p_{1z})}$$

¿Pero cómo sabemos si la recta va hacia un lado o hacia otro, para saber qué valor de t es el que tenemos que calcular para determinar el punto de proyección?

Pues es relativamente fácil:

- Cojamos siempre como referencia el punto que está por detrás del punto de vista. En nuestro ejemplo es siempre p_1 .

- La línea puede tener tres posibles direcciones en el eje x : o hacia la izquierda (hacia el valor 0), o hacia la derecha (hacia el valor S_x), o vertical porque no avance ni hacia un lado ni hacia otro.
 - Si la coordenada x de p_1 es $< \frac{S_x}{2}$ irá hacia la izquierda (hacia el valor 0 de x).
 - Si es mayor, irá hacia la derecha hacia la derecha (hacia el valor S_x de x).
 - y si es igual a $\frac{S_x}{2}$ la decisión habrá que tomarla en función de esa misma coordenada, pero para el punto p_2 , y razonando de manera inversa a como lo hemos hecho para p_1 . Es decir, en este caso si $p_{2x} < \frac{S_x}{2}$ el punto va hacia la derecha (hacia el valor S_x de x) y viceversa.
 - Si además es ese segundo valor es también $= \frac{S_x}{2}$ la proyección en el eje X será perpendicular.
- Otro tanto puede razonarse de manera equivalente para la dirección en el eje y .
- Si según los pasos anteriores, la proyección de la recta fuera vertical y además también horizontal, la proyección de la recta es un punto pues empieza y acaba en la coordenada $(x, y) = (\frac{S_x}{2}, \frac{S_y}{2})$ variando sólo la coordenada z .

Por tanto, ya tenemos el algoritmo completo a utilizar cuando una recta tiene punto final detrás del punto de vista:

Para una línea:

- ¿Los dos puntos de la línea son potencialmente visibles? Dejarlo los puntos originales como están y terminar.
- ¿Los dos puntos son invisibles? Eliminarlos de la lista de puntos a proyectar y terminar.
- ¿Hay uno visible y el otro invisible?
 - Calcular la dirección que tendrá la proyección de la recta en el plano X: izquierda, derecha o vertical (según se ha visto en el párrafo anterior)
 - Lo mismo en el plano Y: arriba, abajo o horizontal.
 - Si la recta es vertical y horizontal en proyección, entonces la recta es, en realidad, perpendicular al plano de proyección r y pasando por el punto $(\frac{S_x}{2}, \frac{S_y}{2})$. Se asigna como nuevo punto $p_1 = (\frac{S_x}{2}, \frac{S_y}{2}, 0)$ para que la proyección salga un punto en el centro de la pantalla.
 - En otro caso:
 - Se definen dos variables intermedias t_x y t_y .
 - Si la proyección de la recta es vertical pero no horizontal a la vez, abandonará la pantalla por el eje Y , luego se calcula $t_y = t_3$ o $t_y = t_4$ en función de si ésta se vaya hacia arriba o hacia abajo. Se elije además como valor de $t_x = 1$.
 - Si la proyección de la recta es horizontal pero no vertical, abandonará la pantalla por el eje X , luego se calcula $t_x = t_1$ o $t_x = t_2$ en función de si ésta vaya hacia la izquierda o hacia la derecha. Se elije como valor de $t_y = 1$.
 - Si la proyección de la recta es vertical y horizontal a la vez, es un punto y podemos fijar el valor de $t_x = t_y = 1$.
 - Si no es ninguno de los dos casos anteriores:
 - Si la recta va hacia la izquierda $t_x = t_1$.

- Si va hacia la derecha $t_x = t_2$.
- Si va hacia arriba $t_y = t_3$.
- Si va hacia abajo $t_y = t_4$.
- Con los valores seleccionados de t_x y t_y se calcula la coordenada del punto z límite de la recta. Se elige el $t(x \text{ o } y)$ que de una coordenada z menor, pero siempre mayor que el mínimo visible.
- Se recalcula el punto p_1 con las coordenadas que se deduzcan del valor de t seleccionado.

19: Algoritmo para limitar los puntos fuera de la zona de visión en una proyección cónica

Veamos aquí un trozo de código en c++ (función lambda) que limita los puntos de una recta, recibida como parámetros, en caso de que alguno de ellos esté por detrás del punto de vista. La función devuelve true si los puntos han sido limitados y false en caso contrario. La función implementa el algoritmo descrito arriba:

```
auto limitLinePoints = [this](QGAMES::Position& p1, QGAMES::Position& p2) -> bool
{
    bool result = true;

    bool p1V = p1.posZ () <= _status._conicLimitZ;
    bool p2V = p2.posZ () <= _status._conicLimitZ;

    // Both are out of the limts...
    if (p1V && p2V)
        p1 = p2 = QGAMES::Position::_noPoint; // Not visible...
    else
        // One them only is out of the limits
        if (p1V || p2V)
        {
            QGAMES::Position& pOut = p1V ? p1 : p2;
            QGAMES::Position& pIn = p1V ? p2 : p1; // The other...

            QGAMES::bdata mW2 = __BD _status._maximumWidth / __BD 2;
            // 1 left, 2 right, 3 vertical...
            unsigned left = pOut posX () < mW2
                ? 1
                : ((pOut posX ()) > mW2)
                    ? 2
                    : (pIn posX () > mW2) ? 1 : ((pIn posX ()) < mW2) ? 2 : 3);

            QGAMES::bdata mH2 = __BD _status._maximumHeight / __BD 2;
            // 1 up, 2 down, 3 horizontal...
            unsigned up = pOut posY () < mH2
                ? 1
                : ((pOut posY ()) > mH2)
                    ? 2
                    : (pIn posY () > mH2) ? 1 : ((pIn posY ()) < mH2) ? 2 : 3);

            // If the projection is vertical and horizontal at the same time
            // then the line is perpendicular to the projection plane
            // and going through the center where the vision point is projected too...
            if (left == 3 && up == 3)
                pOut = QGAMES::Position (mW2, mH2, _status._conicLimitZ); // So the last
            point visible should be the limit...
        }
        else
        {
            QGAMES::bdata tx = __BD 1;
            QGAMES::bdata ty = __BD 1;

            if (left != 3)
                tx = (left == 1) // left?
                    ? (-__BD _status._maximumWidth * (pOut posX () + pOut posZ ()) /
                        (__BD 2 * (pIn posX () - pOut posX ()) +
                        _status._maximumWidth * (pIn posZ () - pOut posZ ())))
                    : ((__BD _status._maximumWidth - pOut posX ()) + pOut posZ ()) /
                        ((pIn posX () - pOut posX ()) - (pIn posZ () - pOut posZ ()));
            if (up != 3)
                ty = (up == 1) // up?
                    ? (-__BD _status._maximumHeight * (pOut posY () + pOut posZ ()) /
                        (__BD 2 * (pIn posY () - pOut posY ()) +
                        _status._maximumHeight * (pIn posZ () - pOut posZ ()))
                    : ((__BD _status._maximumHeight - pOut posY ()) + pOut posZ ()) /
                        ((pIn posY () - pOut posY ()) - (pIn posZ () - pOut posZ ()));
        }
}
```

```

        ty = (up == 1) // up?
            ? (-__BD __status._maximumWidth * (pOut.posY () + (pOut.posZ () /
__status._screenRatio))) /
                ((__BD 2 * (pIn.posY () - pOut.posY ()) + ((__BD
__status._maximumWidth / __status._screenRatio) * (pIn.posZ () - pOut.posZ ())))
                : (((__BD __status._maximumWidth / __status._screenRatio) - pOut.posY
()) + (pOut.posZ () / __status._screenRatio)) /
                    ((pIn.posY () - pOut.posY ()) - ((__BD 1 / __status._screenRatio)
* (pIn.posZ () - pOut.posZ ())));
            assert (tx != __BD 0 && ty != __BD 0);
            QGAMES::bdata pztz = pOut.posZ () + ((pIn.posZ () - pOut.posZ ()) * tx);
            if (pztz < __status._conicLimitZ) pztz = std::numeric_limits
<QGAMES::bdata> ().max ();
            QGAMES::bdata pztz = pOut.posZ () + ((pIn.posZ () - pOut.posZ ()) * ty);
            if (pztz < __status._conicLimitZ) pztz = std::numeric_limits
<QGAMES::bdata> ().max ();
            assert (pztz != std::numeric_limits <QGAMES::bdata> ().max () || pztz !=
std::numeric_limits <QGAMES::bdata> ().max ());
            pOut = (pztz < pztz)
                ? QGAMES::Position (__BD pOut posX () + ((pIn posX () - pOut posX ())
* tx),
                    __BD pOut posY () + ((pIn posY () - pOut posY ()) * tx),
                    pztz)
                : QGAMES::Position (__BD pOut posX () + ((pIn posX () - pOut posX ())
* ty),
                    __BD pOut posY () + ((pIn posY () - pOut posY ()) * ty),
                    pztz);
        }
    }
    else
        result = false;
}

return (result);
};

```

20: Función lambda para recortar los puntos de una línea en proyección cónica

Realicemos, por último, una pequeña comprobación adicional.

¿Puede valer t valer 0 en alguno de los cálculos realizados? ¿Qué significaría?

Tengamos presente, antes de continuar, que el recorte de líneas y, por tanto, los algoritmos comentados anteriormente, sólo son aplicables si el punto p_1 estuviera más allá del punto de vista. De otra manera el recorte no se realiza y los puntos se proyectan de manera normal.

Cojamos, por ejemplo, t_1 :

$$t_1 = \frac{-vp_{1x} - \frac{S_x}{2}p_{1z}}{(p_{2x} - p_{1x}) + \frac{S_x}{2}(p_{2z} - p_{1z})} = 0$$

$$-vp_{1x} - \frac{S_x}{2}p_{1z} = 0$$

$$-vp_{1x} = \frac{S_x}{2}p_{1z} \Rightarrow p_{1x} = -\frac{S_x}{2v}p_{1z}$$

En nuestra habitual convención, significaría que el punto que está fuera de la zona visible se encuentra en una recta:

$$r_{t1} \equiv p_{1x} = -p_{1z}$$

Que representa una recta en el plano XZ.

De igual manera para t_2 :

$$r_{t2} \equiv p_{1y} = -p_{1z}$$

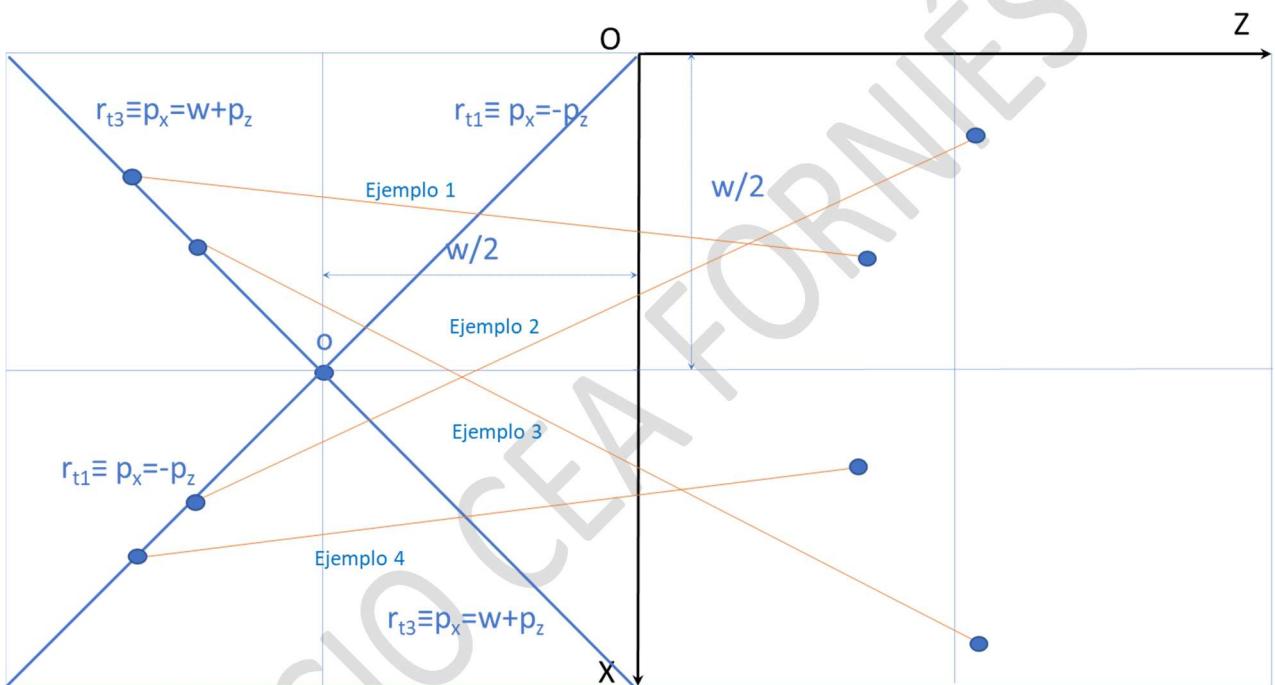
Que representa una recta en el plano YZ.

Y, de manera equivalente para t_3 y t_4 , rectas también en los planos XZ e YZ respectivamente:

$$r_{t3} \equiv p_{1x} = w + p_{1z}$$

$$r_{t4} \equiv p_{1y} = w + p_{1y}$$

En la siguiente figura se ha graficado el valor de esas rectas límite para el caso de t_1 y t_3 (plano XZ). De manera semejante se podrían haber graficado las rectas límite correspondientes a los puntos t_2 y t_4 en el plano YZ:



21: Tipos de rectas que hacen t_1 o t_3 nulo

Pero, el valor de t_1 sólo se debería utilizar cuando la línea se dirige hacia la parte izquierda de la pantalla (hacia el valor 0). En este caso, hubiéramos seleccionado el valor t_2 .

¿Qué valor tendría en nuestra habitual convención:

$$t_2 = \frac{(w - p_{1x}) + p_{1z}}{(p_{2x} - p_{1x}) - (p_{2z} - p_{1z})} = \frac{w + 2p_{1z}}{p_{2x} - p_{2z} + 2p_{1z}}$$

Que sólo es cero cuando:

$$p_{1z} = -\frac{w}{2}$$

Es decir, si el punto final de la recta está justo en el plano en el que se ubica el punto de vista, es decir coincide con 0. Esto quiere decir que la recta termina en la recta perpendicular al plano XZ que pasa por el punto de vista o (recta excepción en la figura).

En este caso, el último punto visible sería un poquito justo por delante de 0. Bastaría con hacer que t nunca pudiera ser menor de un determinado valor.

De igual manera se podría haber razonado entre t_3 y t_4 .

Resumiendo, el valor de t para nuestro caso más habitual, en el que:

$$v = \frac{S_x}{2} = \frac{w}{2}$$

$$\frac{S_x}{S_y} = r_a; S_y = \frac{1}{r_a} S_x = \frac{1}{r_a} 2v; \frac{S_y}{2} = \frac{v}{r_a}$$

Serían:

$$t_1 = \frac{-w(\mathbf{p}_{1x} + \mathbf{p}_{1z})}{2(\mathbf{p}_{2x} - \mathbf{p}_{1x}) + w(\mathbf{p}_{2z} - \mathbf{p}_{1z})}$$

$$t_2 = \frac{(w - \mathbf{p}_{1x}) + \mathbf{p}_{1z}}{(\mathbf{p}_{2x} - \mathbf{p}_{1x}) - (\mathbf{p}_{2z} - \mathbf{p}_{1z})}$$

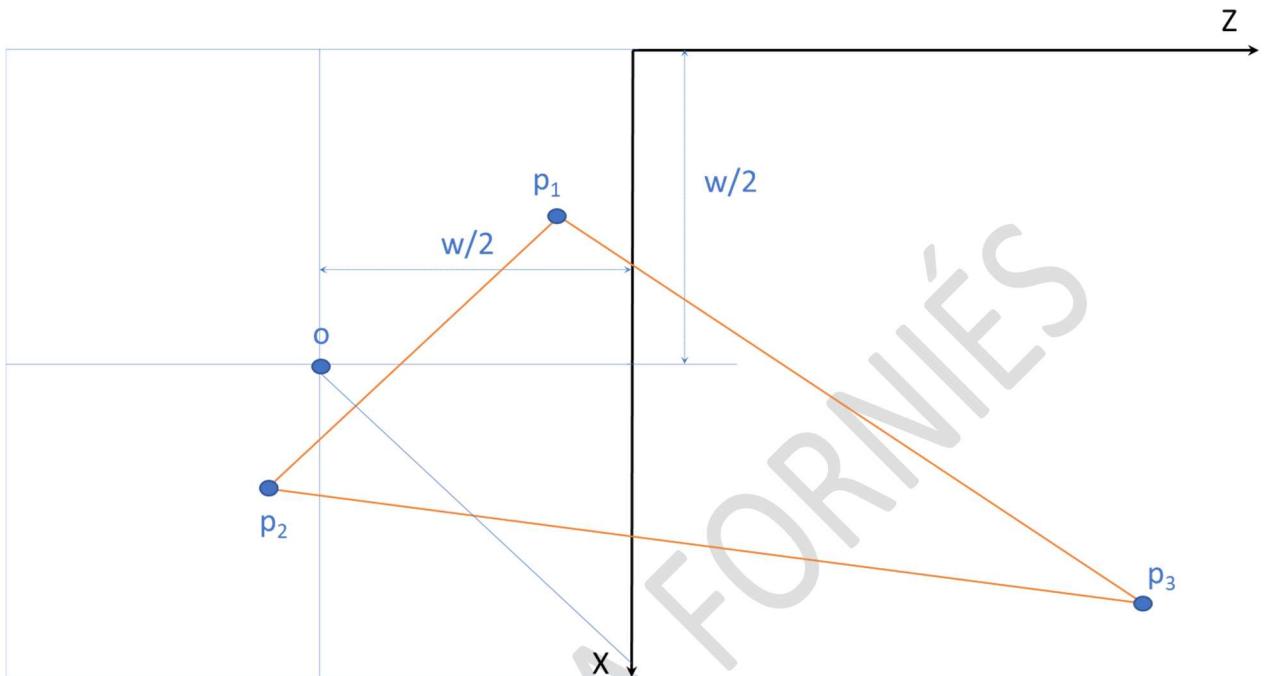
$$t_3 = \frac{-w\left(\mathbf{p}_{1y} + \frac{1}{r_a}\mathbf{p}_{1z}\right)}{2(\mathbf{p}_{2y} - \mathbf{p}_{1y}) + \frac{1}{r_a}w(\mathbf{p}_{2z} - \mathbf{p}_{1z})}$$

$$t_4 = \frac{\left(\frac{1}{r_a}w - \mathbf{p}_{1y}\right) + \frac{1}{r_a}\mathbf{p}_{1z}}{(\mathbf{p}_{2y} - \mathbf{p}_{1y}) - \frac{1}{r_a}(\mathbf{p}_{2z} - \mathbf{p}_{1z})}$$

20: Valores de t para limitar los puntos de una recta más allá del punto de vista en una proyección cónica

Proyección de polígonos con puntos por detrás del punto de vista

¿Cómo tratamos los polígonos en el que alguno de sus puntos se ubique más allá del punto de vista?



22: Figura geométrica con puntos más allá del punto de vista

Una primera reflexión consistiría en tratar según se ha visto en el punto anterior las parejas contiguas de vértices, unidas por una línea.

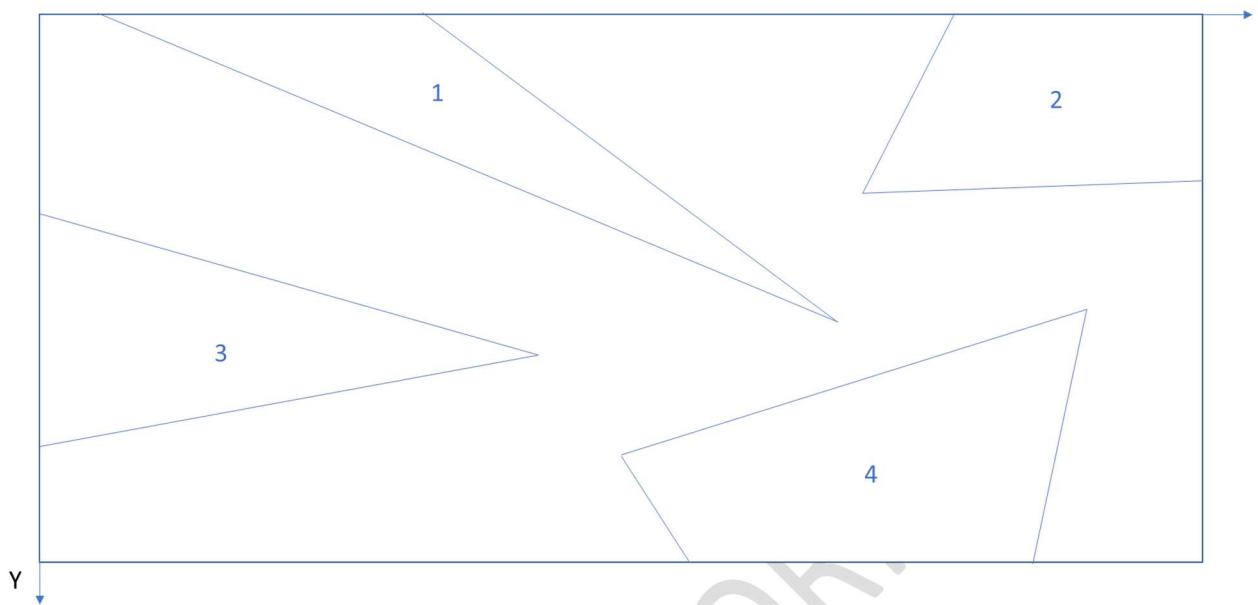
De la línea que une p_2 con p_1 sólo sería visible, según se analizado anteriormente, el trozo que va de p_4 a p_1 , y de la línea que une p_2 con p_3 , sólo sería visible el trozo que va de p_5 a p_1 .

Nuestro polígono quedaría entonces “abierto”. Podríamos imaginar cerrarlo “imaginariamente” con la línea que uniera los puntos p_4 y p_5 , pero nada garantizaría entonces que dicha línea no sea perfectamente visible. Todo dependería de los valores de dichos puntos en el eje Y que no vemos en esta ilustración. Por tanto, cerrar arbitrariamente el polígono con una línea no es la solución definitiva.

Nótese que el punto p_2 , que era único, se ha desdoblado, por otro lado, en 2: uno para la línea de p_2 a p_1 (p_4) y otro para la línea de p_2 a p_3 (p_5).

Podríamos, simplemente dejar el polígono abierto y sólo pintar líneas, pero si éste estuviera relleno no podríamos pintarle, al menos de la forma adecuada. Más aún si se rellena con una textura y no sólo con color.

En la siguiente ilustración se muestran cuatro posibles proyecciones en la pantalla de nuestro triángulo ejemplo:



23: Posibles proyecciones de un triángulo con algún vértice más allá del punto de vista

Como se puede ver, unir los puntos límite en el caso 2, originaría un polígono cerrado, pero no reflejo del triángulo original. Sin embargo, eso sí que ocurriría en los otros tres ejemplos. La diferencia entre ambas situaciones estriba en que los puntos o salen de la pantalla por el mismo eje (X o Y) o por diferentes ejes (X e Y).

Pongamos una primera versión de nuestro algoritmo:

- A partir de la lista de vértices contiguos de la figura geométrica, generamos líneas que los unan por pares. Habrá así tantas líneas como vértices. La lista de líneas estará ordenada y cada línea tendrá su punto de inicio y su punto de fin independiente de las demás.
- Calculamos los límites visibles de todas esas líneas según el algoritmo desgranado en el punto anterior (cambiarán, por tanto, el valor de algunos puntos). Y algunas líneas aparecerán como totalmente invisibles porque su punto de inicio y fin están por detrás del punto de vista.
- Colocamos todos los puntos proyectados visibles en una lista.
- Los proyectamos.
- Creamos una lista resultado vacía.
- Declaramos los puntos esquina con coordenadas $(0,0)$, $(w,0)$, (w,h) y $(0,h)$ respectivamente.
- Iteramos sobre la lista de punto proyectados visibles:
 - Si el punto no está en la lista resultado:
 - Si el nuevo punto no está ubicado en el límite de la pantalla, insertarlo.
 - Si está ubicado en el límite de la pantalla, compararlo con el anterior (si hubiere):
 - Si ese anterior está también ubicado en el límite de la pantalla:
 - Si ambos puntos están en el mismo lado límite, insertar el nuevo punto sin más en la lista resultado.

- Si están en diferentes lados (uno en el X y el otro en el Y o viceversa), insertar tantos nuevos puntos esquina como estén entre los dos lados a los que pertenecen los dos puntos. Insertar luego el punto.
- Si no está ubicado en el límite de la pantalla, insertarlo.
- Si el punto está ya en la lista resultado, no habría que hacer nada:

Al final tendremos una lista de puntos que forman vértices de un nuevo polígono, esta vez proyectado.

Proyección en perspectiva cilíndrica ortogonal (ortogonal)

La proyección ortogonal es la más fácil de realizar, sin duda.

En este caso, el punto p y el punto p' proyectado ocupan posiciones semejantes. La única diferencia está en que, debido a la proyección cilíndrica ortogonal, la coordenada z del punto proyectado (que está en 2D) será siempre cero. Por tanto:

$$p' = (p_x, p_y)$$

Al ser la proyección ortogonal una proyección cilíndrica, los puntos con coordenada $z < 0$ se proyectarán de la misma manera sobre el plano de proyección que aquellos equivalentes con $z > 0$. En una proyección cilíndrica la proyección tiene la misma forma que si se observara desde el infinito. Por tanto, no es necesario realizar ninguna adaptación cuando los puntos a proyectar tienen una coordenada $z < 0$.

Proyección en perspectiva axonométrica isométrica (isométrica)

La proyección axonométrica isométrica involucra en realidad dos proyecciones.

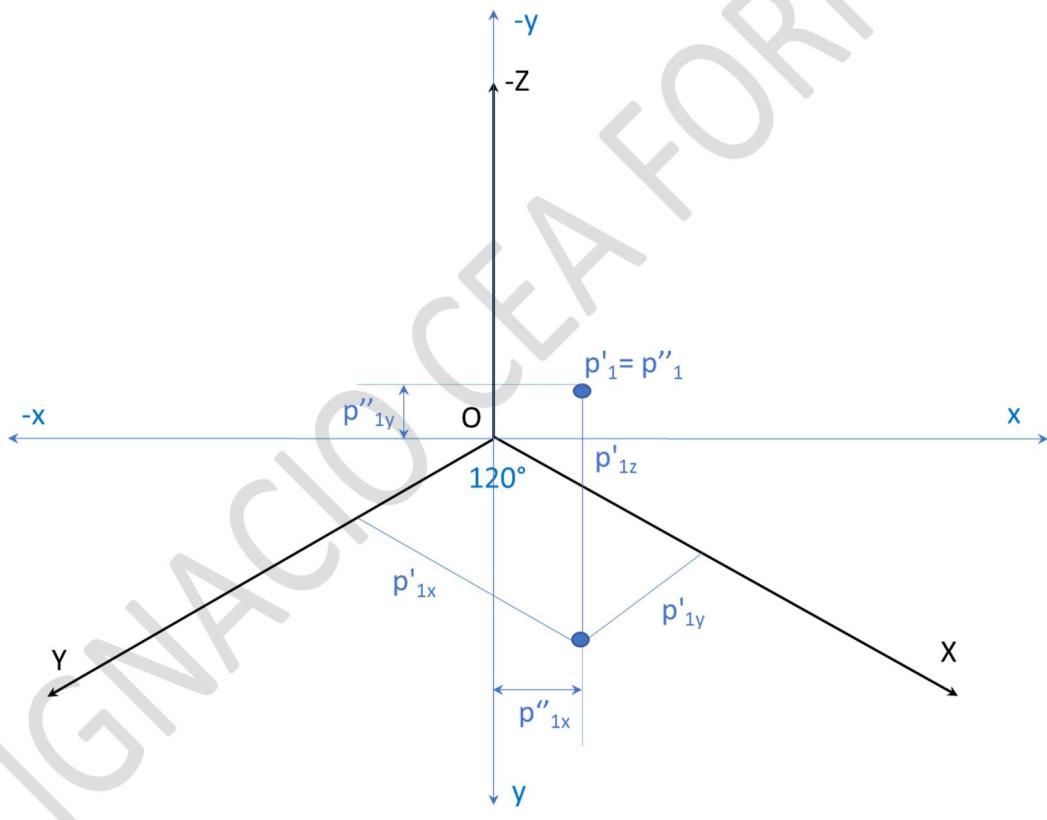
La primera, en la que un punto p es proyectado sobre un plano que pasa por el origen de coordenadas y que forma ángulos α , β y γ respectivamente con los ejes \overrightarrow{OX} , \overrightarrow{OY} y \overrightarrow{OZ} . El punto proyectado puede ponerse en función de sus coordenadas sobre los ejes proyectados. Así:

$$p' = (p_x \cos \alpha, p_y \cos \beta, p_z \cos \gamma)$$

Y, si todos los ángulos son iguales:

$$p' = 0,816(p_x, p_y, p_z)$$

A partir de ahí se realiza una proyección cilíndrica ortogonal sobre un plano paralelo al anterior. Solo hay que hacer entonces un cambio de coordenadas para calcular el punto final proyectado:



24: Proyección isométrica. Calculo en la pantalla de proyección

$$p''_x = p'_{1x} \cos 30^\circ - p'_{1y} \cos 30^\circ = \frac{\sqrt{3}}{2}(p'_{1x} - p'_{1y})$$

$$p''_y = p'_{1x} \sin 30^\circ + p'_{1y} \sin 30^\circ + p'_{1z} = \frac{1}{2}(p'_{1x} + p'_{1y}) + p'_{1z}$$

Sabiendo además que p'_{1x} , p'_{1y} y p'_{1z} tienen como valor los de la posición original multiplicado por 0,816 (tal y como se vio anteriormente).

Luego el punto proyectado estaría finalmente sobre las coordenadas de la pantalla (generalizando a cualquier p):

$$\mathbf{p}'' = \mathbf{0}, \mathbf{816} \left(\frac{\sqrt{3}}{2} (\mathbf{p}_{1x} - \mathbf{p}_{1y}), \frac{1}{2} (\mathbf{p}_{1x} + \mathbf{p}_{1y}) + \mathbf{p}_{1z} \right)$$

Intentemos algunas aproximaciones que nos pueden ser útiles más adelante.

La primera es prescindir de los valores de reducción:

$$\mathbf{p}'' = \left(\frac{\sqrt{3}}{2} (\mathbf{p}_{1x} - \mathbf{p}_{1y}), \frac{1}{2} (\mathbf{p}_{1x} + \mathbf{p}_{1y}) + \mathbf{p}_{1z} \right)$$

Y la segunda, suponer que además que $\frac{\sqrt{3}}{2} \approx 1$. Entonces:

$$\mathbf{p}'' = \left(\mathbf{p}_{1x} - \mathbf{p}_{1y}, \frac{1}{2} (\mathbf{p}_{1x} + \mathbf{p}_{1y}) + \mathbf{p}_{1z} \right)$$

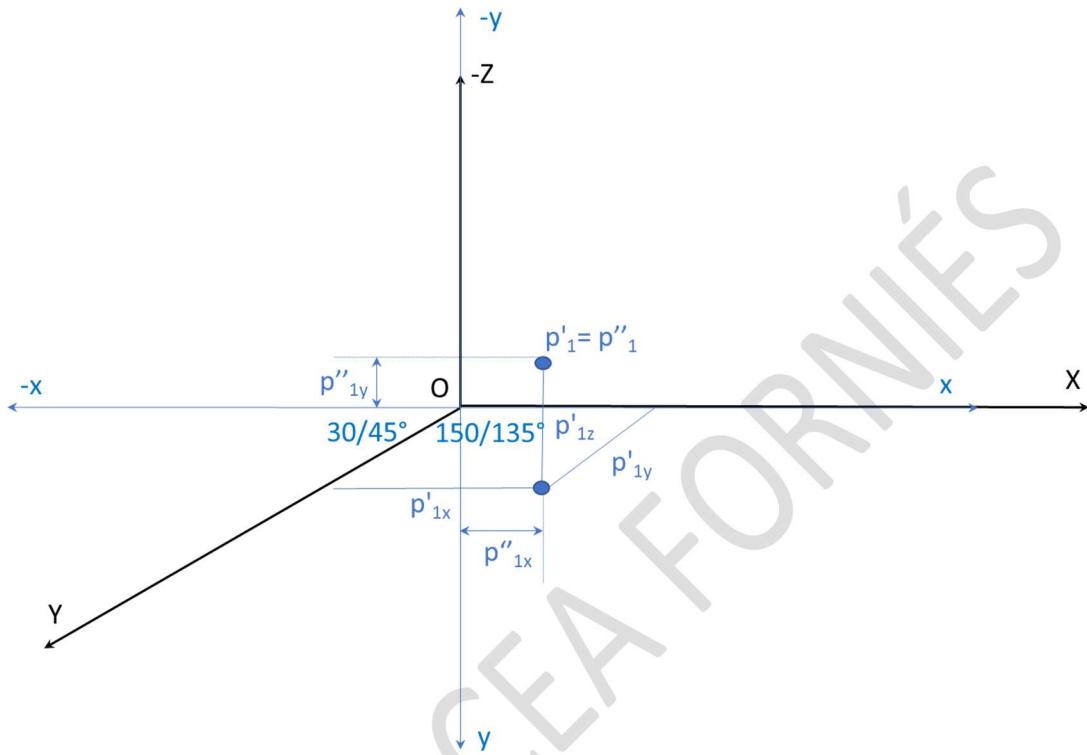
21: Coordenadas del punto proyectado en perspectiva axonométrica isométrica

Llamaremos a esta perspectiva, perspectiva isométrica rápida o sencilla.

Al ser la proyección isométrica una proyección cilíndrica, los puntos con coordenada $z < 0$ se proyectarán de la misma manera sobre el plano de proyección que aquellos equivalentes con $z > 0$. En una proyección cilíndrica la proyección tiene la misma forma que si se observara desde el infinito. Por tanto, no es necesario realizar ninguna adaptación cuando los puntos a proyectar tienen una coordenada $z < 0$.

Proyección en perspectiva axonométrica caballera (caballera)

Esta proyección es bastante parecida a la anterior, pero al ser una proyección cilíndrica oblicua, varían los ángulos de los ejes axonométricos, y con ello los valores de la proyección sobre la referencia de la pantalla:



25: Proyección caballera. Cálculo en la pantalla de proyección

Cojamos como referencia el ángulo de 45 grados:

$$p''_{1x} = p'_{1x} - p'_{1y} \cos 45^\circ = p'_{1x} - \frac{\sqrt{2}}{2} p'_{1y}$$

$$p''_{1y} = p'_{1y} \sin 45^\circ + p'_{1z} = \frac{\sqrt{2}}{2} p'_{1y} + p'_{1z}$$

$$\mathbf{p}'' = 0,816 \left(p'_{1x} - \frac{\sqrt{2}}{2} p'_{1y}, \frac{\sqrt{2}}{2} p'_{1y} + p'_{1z} \right)$$

Aplicando el mismo racional que en la proyección anterior:

$$\mathbf{p}'' = \left(p_{1x} - \frac{1}{2} p_{1y}, \frac{1}{2} p_{1y} + p_{1z} \right)$$

22: Coordenadas proyectadas de un punto en perspectiva axonométrica caballera

Llamaremos a esta perspectiva, perspectiva caballera rápida o sencilla.

Al ser la proyección isométrica una proyección cilíndrica, los puntos con coordenada $z < 0$ se proyectarán de la misma manera sobre el plano de proyección que aquellos equivalentes con $z > 0$. En una proyección cilíndrica la proyección tiene la misma forma que si se observara

desde el infinito. Por tanto, no es necesario realizar ninguna adaptación cuando los puntos a proyectar tienen una coordenada $z < 0$.

IGNACIO CEA FORNIÉS

Cambio de sistema de referencia

Antes de proyectar cualquier punto del mundo real referido al sistema de referencia universal s , que sabemos que es el permanente y constante, y según lo podemos deducir de los puntos anteriores, es necesario haberlo referido al sistema de referencia que marca el plano de proyección y .

Cambio de base

Antes de profundizar en las implicaciones de un cambio de sistema de referencia, que es lo que en el fondo se persigue, es necesario abordar los cambios de base.

Supongamos que tenemos dos bases cualesquiera $U = \{\vec{u}_1, \vec{u}_2, \vec{u}_3\}$ y $V = \{\vec{v}_1, \vec{v}_2, \vec{v}_3\}$. Sea \vec{w} un vector cualquiera que puede expresarse en ambas bases. Se tiene entonces que:

$$\vec{w} = \alpha_1 \vec{u}_1 + \alpha_2 \vec{u}_2 + \alpha_3 \vec{u}_3$$

O:

$$\vec{w} = \beta_1 \vec{v}_1 + \beta_2 \vec{v}_2 + \beta_3 \vec{v}_3$$

Y supongamos que, además los vectores de la base v se pueden poner en función de los de la base u , de la siguiente forma:

$$\begin{aligned}\vec{v}_1 &= \lambda_{11} \vec{u}_1 + \lambda_{21} \vec{u}_2 + \lambda_{31} \vec{u}_3 \\ \vec{v}_2 &= \lambda_{12} \vec{u}_1 + \lambda_{22} \vec{u}_2 + \lambda_{32} \vec{u}_3 \\ \vec{v}_3 &= \lambda_{13} \vec{u}_1 + \lambda_{23} \vec{u}_2 + \lambda_{33} \vec{u}_3\end{aligned}$$

O bien, puesto en forma matricial:

$$[\vec{v}_1 \quad \vec{v}_2 \quad \vec{v}_3] = [\vec{u}_1 \quad \vec{u}_2 \quad \vec{u}_3] \begin{bmatrix} \lambda_{11} & \lambda_{21} & \lambda_{31} \\ \lambda_{12} & \lambda_{22} & \lambda_{32} \\ \lambda_{13} & \lambda_{23} & \lambda_{33} \end{bmatrix} \Rightarrow V = UR$$

Donde llamamos R a la matriz de cambio de base.

Volviendo ahora al planteamiento inicial:

$$w = \beta_1 \vec{v}_1 + \beta_2 \vec{v}_2 + \beta_3 \vec{v}_3 = [\vec{v}_1 \quad \vec{v}_2 \quad \vec{v}_3] \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = [\vec{u}_1 \quad \vec{u}_2 \quad \vec{u}_3] R \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = [\vec{u}_1 \quad \vec{u}_2 \quad \vec{u}_3] \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$$

De donde se deduce que:

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = R \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

O bien que:

$$\begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = R^{-1} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$$

Si U representara la base de nuestro sistema de referencia universal fijo (u), y V la base del sistema de referencia del plano de proyección (v) y por ello asociado a la cámara, lo que toca siempre hacer es buscar son los parámetros beta (β) en función de los parámetros alfa (α). Es decir, lo que en nuestro racional representa la matriz R^{-1} . Volveremos a este importante concepto más adelante.

En nuestro caso, además, es necesario tener presente los dos sistemas de referencia (u y v) son ortonormales; esto es, que forman entre si un ángulo de 90 grados y además el módulo de cada vector de la base es la unidad.

En ese caso:

$$U^{-1} = U^{T_{10}}$$

Y:

$$\text{Si } V = UR \Rightarrow R = U^{-1}V = U^T V$$

Y extendiendo:

$$R = \begin{bmatrix} \vec{u}_1 \\ \vec{u}_2 \\ \vec{u}_3 \end{bmatrix} [\vec{v}_1 \quad \vec{v}_2 \quad \vec{v}_3] = \begin{bmatrix} \vec{u}_1 \cdot \vec{v}_1 & \vec{u}_1 \cdot \vec{v}_2 & \vec{u}_1 \cdot \vec{v}_3 \\ \vec{u}_2 \cdot \vec{v}_1 & \vec{u}_2 \cdot \vec{v}_2 & \vec{u}_2 \cdot \vec{v}_3 \\ \vec{u}_3 \cdot \vec{v}_1 & \vec{u}_3 \cdot \vec{v}_2 & \vec{u}_3 \cdot \vec{v}_3 \end{bmatrix}$$

Es decir, que la matriz R de cambio de base está compuesta por los productos escalares entre los diferentes vectores que forman las dos bases.

Y dado que, a su vez, el producto escalar es:

$$\vec{u}_i \cdot \vec{v}_j = \|\vec{u}_i\| \|\vec{v}_j\| \cos \varphi_{ij}$$

Y el módulo de los vectores de la base es 1, al ser éstos ortonormales, los elementos de la matriz de cambio de base son el coseno de los ángulos que forman unos vectores con otros:

$$\vec{u}_i \cdot \vec{v}_j = \cos \varphi_{ij}$$

Podemos encadenar múltiples cambios de base y entonces se demuestra que:

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \dots R_3 R_2 R_1 \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = R_T \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

Siendo R_1, R_2, R_3 cambios consecutivos. O bien:

$$\begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = R_T^{-1} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = R_1^{-1} R_2^{-1} R_3^{-1} \dots \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

De mucha utilidad como veremos más adelante.

Cambio de sistema de referencia

Un sistema de referencia es el conjunto de un punto (origen) y una base de referencia.

¹⁰ Algebra básica.

En un juego manejamos, básicamente, dos sistemas de referencia: Uno fijo, llamado universal (u) y el otro móvil (v), asociado al movimiento de la cámara.

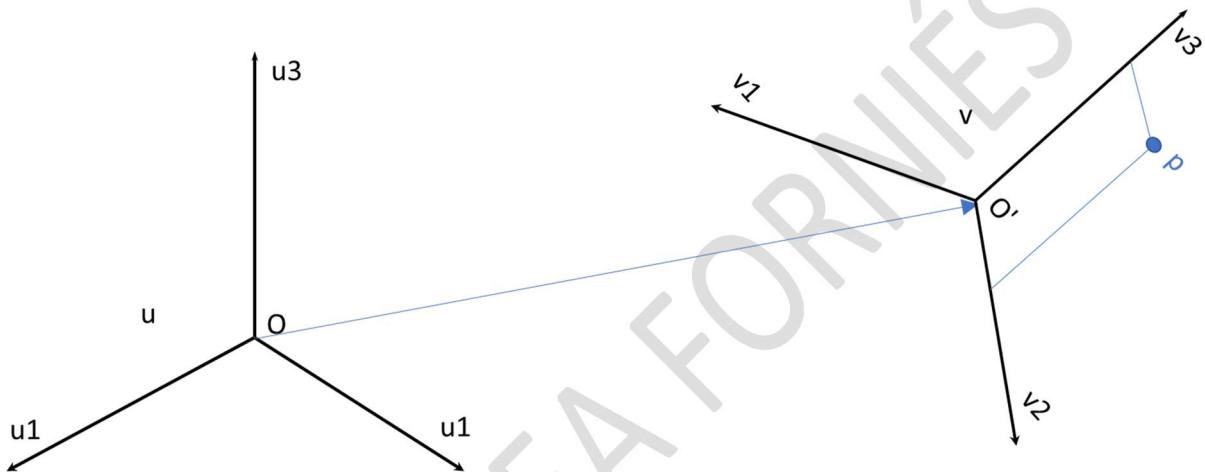
Sea, por tanto:

$$u = \{O, [\vec{u}_1 \quad \vec{u}_2 \quad \vec{u}_3]\}$$

El sistema de referencia universal al que se refieren todos los objetos del espacio, y:

$$v = \{O', [\vec{v}_1 \quad \vec{v}_2 \quad \vec{v}_3]\}$$

El sistema de referencia asociado al plano de proyección (y) y con ello a la cámara y a la pantalla en sí misma. Ambos de vectores ortonormales (ángulo 90 grados y módulo 1):



26: Cambio de sistema de referencia

Sea un punto cualquier del espacio P . Podríamos escribir que:

$$\overrightarrow{OP} \equiv P - O = \alpha_1 \vec{u}_1 + \alpha_2 \vec{u}_2 + \alpha_3 \vec{u}_3 = [\vec{u}_1 \quad \vec{u}_2 \quad \vec{u}_3] \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$$

Y que:

$$\overrightarrow{O'P} \equiv P - O' = \beta_1 \vec{v}_1 + \beta_2 \vec{v}_2 + \beta_3 \vec{v}_3 = [\vec{v}_1 \quad \vec{v}_2 \quad \vec{v}_3] \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

Cogiendo el primero:

$$\overrightarrow{OP} = \overrightarrow{OO'} + \overrightarrow{O'P} = \overrightarrow{OO'} + [\vec{v}_1 \quad \vec{v}_2 \quad \vec{v}_3] \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

E igualando ambas:

$$[\vec{u}_1 \quad \vec{u}_2 \quad \vec{u}_3] \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \overrightarrow{OO'} + [\vec{v}_1 \quad \vec{v}_2 \quad \vec{v}_3] \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = \overrightarrow{OO'} + V \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

$$U \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = V \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} + \overrightarrow{OO'}$$

Despejando, y dado que las bases son ortonormales:

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = R \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} + \overrightarrow{OO'}$$

23: Coordenadas universales de un punto a partir de las mismas respecto a otro sistema de referencia

Y, en este caso:

$$R^{-1} \left(\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} - \overrightarrow{OO'} \right) = R^{-1}R \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

Luego:

$$\begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} = R^{-1} \left(\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} - \overrightarrow{OO'} \right)$$

24: Coordenadas de un punto en un sistema de referencia a partir de sus equivalentes en un sistema universal

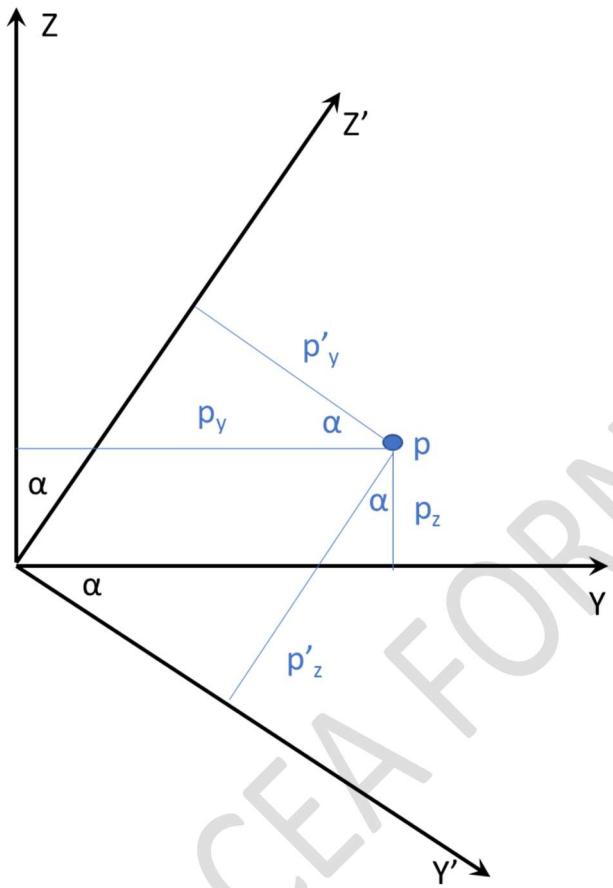
Emplearemos la anterior fórmula para, dados los componentes de un punto cualquiera en el sistema de referencia universal y la posición del origen del sistema de referencia del plano de proyección, calcular las coordenadas de ese punto en el sistema v , antes de proyectarlo sobre el plano de proyección y .

Matrices de giro. Ángulos de Euler

Supongamos que el sistema v mencionado anteriormente se obtiene mediante el giro de un determinado ángulo α alrededor de alguno de los ejes del sistema de referencia u (representado por \vec{u}_1, \vec{u}_2 o \vec{u}_3).

Llamemos a los ejes representados por esos vectores respectivamente X, Y y Z en el sistema u , y X', Y' y Z' en el sistema v . Entonces cualquier punto del espacio p referido al sistema u , puede escribirse referido al sistema v, p' como sigue.

Vamos a considerar sentido positivo de giro el de las agujas del reloj desde el eje en el que se produce el giro (a derechas). Así:



27: Giro sobre el eje X (yaw)

$$p_x = p'_x$$

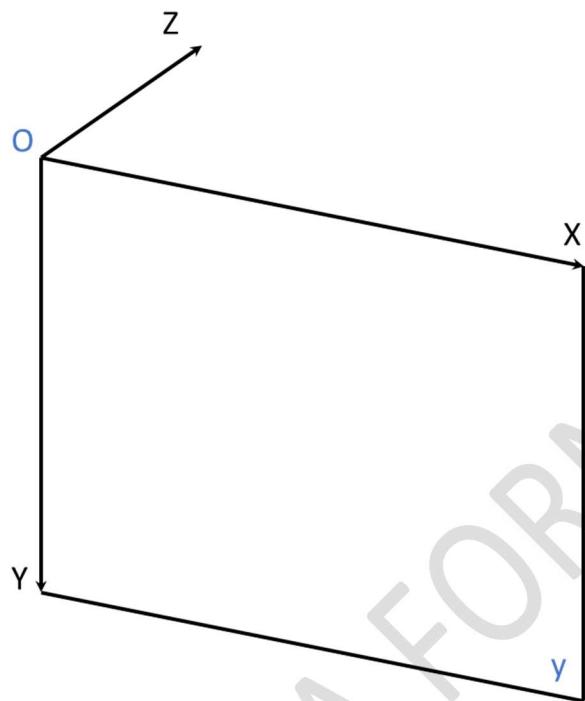
$$p_y = p'_y \cos \alpha + p'_z \sin \alpha$$

$$p_z = -p'_y \sin \alpha + p'_z \cos \alpha$$

O en forma matricial:

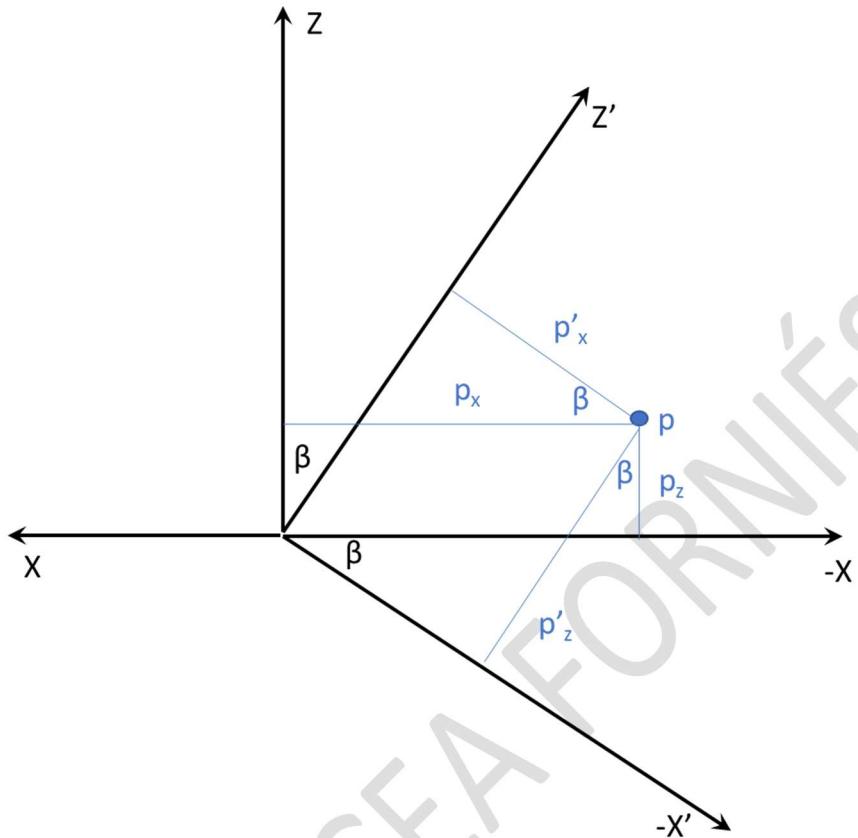
$$\begin{bmatrix} p_x & p_y & p_z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha \\ 0 & -\sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = G_x \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix}$$

Este movimiento equivale, teniendo en cuenta la orientación de ejes que venimos considerando como base de nuestra caja de proyecciones, a inclinar la cabeza la cabeza a derechas y a izquierdas. Movimiento que se conoce como *yaw* en inglés.



28: Referencia de proyección en juegos

De igual manera si el giro para producir el sistema v se hubiera producido alrededor del eje Y y con un ángulo $-\beta$:



29: Giro sobre el eje Y (pitch)

Téngase en cuenta que, en este caso, el giro en el sentido de las agujas del reloj se produce con un ángulo hacia la parte negativa del eje X , por eso se denomina el ángulo girado con el signo negativo.

$$p_x = p'_x \cos -\beta + p'_z \sin -\beta = p'_x \cos \beta - p'_z \sin \beta$$

$$p_y = p'_y$$

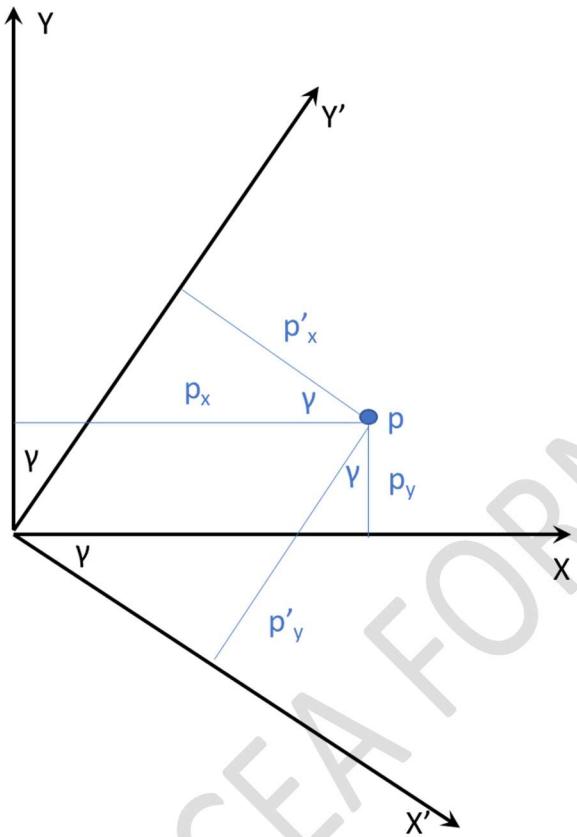
$$p_z = -p'_x \sin -\beta + p'_z \cos -\beta = p'_x \sin \beta + p'_z \cos \beta$$

O, en forma matricial:

$$\begin{bmatrix} p_x & p_y & p_z \end{bmatrix} = \begin{bmatrix} \cos \beta & 0 & -\sin \beta \\ 0 & 1 & 0 \\ \sin \beta & 0 & \cos \beta \end{bmatrix} \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = G_y \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix}$$

Este giro equivaldría a mover de abajo arriba la cabeza.

Y, alrededor del eje Z, con un ángulo γ :



30: Giro sobre el eje Z (roll)

$$p_x = p'_x \cos \gamma + p'_y \sin \gamma$$

$$p_y = -p'_x \sin \gamma + p'_y \cos \gamma$$

$$p_z = p'_z$$

O, en forma matricial:

$$\begin{bmatrix} p_x & p_y & p_z \end{bmatrix} = \begin{bmatrix} \cos \gamma & \sin \gamma & 0 \\ -\sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = G_z \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix}$$

Y este último movimiento equivale a mover la cabeza a derecha o a izquierda.

Cualquier posición en el espacio se puede conseguir mediante una combinación de los tres giros. Si los aplicamos secuencialmente tal y como dedujimos en el punto anterior:

Nuestra matriz de cambio de sistema de referencia podría ponerse como:

$$\begin{bmatrix} p_x & p_y & p_z \end{bmatrix} = G_z G_y G_x \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix} = G \begin{bmatrix} p'_x \\ p'_y \\ p'_z \end{bmatrix}$$

La matriz:

$$G = G_z G_y G_x$$

Es la matriz de giro. Y a los ángulos α , β y γ los ángulos de Euler.

Por tanto, considerando ahora que el sistema de referencia v puede moverse con respecto al sistema universal u hasta un punto O , la fórmula para calcular los puntos universales en relación con nuestro sistema v , que ya hemos dicho que es paso previo para proyectarlos sobre el plano de proyección es:

$$[\mathbf{p}'_x \quad \mathbf{p}'_y \quad \mathbf{p}'_z] = G^{-1} \left(\begin{bmatrix} \mathbf{p}_x \\ \mathbf{p}_y \\ \mathbf{p}_z \end{bmatrix} - \mathbf{o}' \right)$$

25: Ecuación de cálculo de un punto cualquiera referido al sistema de proyección v

Sabiendo que:

$$G^{-1} = (G_z G_y G_x)^{-1} = G_x^{-1} G_y^{-1} G_z^{-1} = G_x^T G_y^T G_z^T$$

Al ser G_i una matriz ortogonal compuesta por filas y columnas de vectores ortonormales.

En resumen:

$$G_x^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha \\ 0 & -\sin \alpha & \cos \alpha \end{bmatrix}^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix}$$

$$G_y^T = \begin{bmatrix} \cos \beta & 0 & -\sin \beta \\ 0 & 1 & 0 \\ \sin \beta & 0 & \cos \beta \end{bmatrix}^T = \begin{bmatrix} \cos \beta & 0 & \sin \beta \\ 0 & 1 & 0 \\ -\sin \beta & 0 & \cos \beta \end{bmatrix}$$

$$G_z^T = \begin{bmatrix} \cos \gamma & \sin \gamma & 0 \\ -\sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} \cos \gamma & -\sin \gamma & 0 \\ \sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

26: Matrices de giro de un punto cualquiera referido al sistema de proyección

Un par de ejemplos

Imaginemos un sistema de referencia s' ubicado en el origen O , igual que el sistema de referencia universal u , pero que ha girado sobre el eje \overrightarrow{OY} (yaw) 90° ($\frac{\pi}{2}$ radianes) a la izquierda. En este caso:

$$\alpha = 0, \beta = \frac{\pi}{2} \text{ y } \gamma = 0$$

Por lo que las matrices de Euler son:

$$G_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

$$G_z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Por lo que:

$$G = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

Supongamos un punto ubicado en la coordenada $(-1 \ 2 \ 3)$ referida al sistema universal e inicialmente no visible en el plano de proyección (antes del movimiento). En el nuevo sistema de coordenadas dicho punto tendrá de coordenadas:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \\ 3 \end{bmatrix} = [3 \ 2 \ 1]$$

Es decir que tras girar la cabeza el punto estará justo delante de nosotros. Fíjese que el valor de la coordenada x de antes es ahora la z y en positivo (delante de nosotros), la de y es ahora la z , y la de z la y , consecuencia del giro.

Imaginemos ahora que el sistema s' estuviera en la posición $v = (5 \ 0 \ 0)$. En este caso, las coordenadas del punto son:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -6 \\ 2 \\ 3 \end{bmatrix} = [3 \ 2 \ 6]$$

Es decir, el punto estaría enfrente nuestro pero un poco más lejos.

Bloque 3: Las hojas de fotogramas. Interpretación en 2.5D

IGNACIO CEA FORNIÉS

Introducción

Fue quizás *Knight Lore*® en 1984 para *Spectrum*® (https://es.wikipedia.org/wiki/Knight_Lore), el primer juego que simulaba 3 dimensiones mediante proyecciones ortogonales en 2. Una forma inteligente (y barata en la utilización de recursos para aquella época ciertamente escasos) de conseguir efectos tridimensionales.

La perspectiva utilizada en aquellas primeras representaciones era básicamente isométrica (se analizará más adelante). Proyección ortogonal que describe bien la dimensión real de los objetos en el espacio y permite al observador hacerse una buena idea de éstos y su posición relativa.

Poco a poco se generalizó esta técnica extendiéndola a otro tipo de proyecciones como la caballera, más sencilla y rápida de calcular
([https://en.wikipedia.org/wiki/Entombed_\(video_game\)](https://en.wikipedia.org/wiki/Entombed_(video_game)))

Los juegos modernos basan mucho su lógica en la utilización de ficheros de gráficos (*sprite sheets*). Para poder ejecutar bien la lógica de un juego basado en proyecciones ortogonales, es necesario poder inferir cuales son las dimensiones y posiciones relevantes de un fotograma al entenderlas desde una perspectiva 3D.

Hojas de fotogramas. “Sprite Sheets”

En los juegos 3D, las entidades que participan en el mismo están definidas en complejos ficheros (.mesh por ejemplo en *OGRE3D®*) en la que se describen uno a uno los diferentes triángulos que conforman su contorno. Así como las diferentes formas de dicha entidades en diferentes fotogramas. Los motores 3D (como *Unity®*, *Unreal®* o *OGRE3D®*) dibujan esas entidades (y sus diferentes formas) como cualquier otra forma ubicada en el mundo que les rodee (una línea, un círculo, etc.).

Esto no es habitual en los juegos cuya base es una proyección ortogonal y típicamente los de dos dimensiones.

En su lugar las entidades que participan en un juego se definen en *Sprite Sheets*, u hojas de fotogramas, como el siguiente:



31: Hoja de fotogramas para un juego ortogonal

En esos ficheros (tipo .jpg, .png,...) aparecen todas los diferentes aspectos que una entidad puede adoptar en cada una de las diferentes circunstancias por las que puede pasar en el transcurso del juego. Dibujando diferentes aspectos por cada bucle del juego pueden conseguirse efectos de movimiento, como si de fotogramas de una película se tratara.

Ahora bien, esas hojas de fotogramas llevan implícita la perspectiva utilizada en el juego en el que participan. Por ejemplo, es obvio que los fotogramas de arriba representan un ninja en un juego 2D ortogonal. También llevan implícito el tamaño con el que allí aparecen. Pero sin duda su utilización, en lugar, de definiciones generales basadas por ejemplo en triángulos, acelera la ejecución de los juegos y simplifica su desarrollo.

Por ejemplo, este otro:



32: Hoja de fotogramas para un juego en perspectiva isométrica

Representa un caballero, preparado para participar en juego en perspectiva isométrica. Y un tipo de hoja no puede ser empleado en una proyección diferente si se quiere conseguir efecto de realidad.

Pero, ahora bien, en este tipo de juegos, y al igual que somos capaces de determinar la ubicación de un objeto inmerso en el (un cubo, una línea...) debemos de ser capaces de determinar diferentes posiciones dentro de cada fotograma, para así poder determinar, por ejemplo, posiciones relativas con aquellos objetos o colisiones con ellos u otras entidades.

Es decir, si, por ejemplo, hemos de considerar un paralelepípedo que contenga al caballero anterior, ¿Cuáles serían sus coordenadas?

Esas dimensiones y puntos son imprescindibles para tareas tan fundamentales como la detección de intersecciones entre elementos del juego o para calcular si una entidad hay que dibujarla antes que otra.

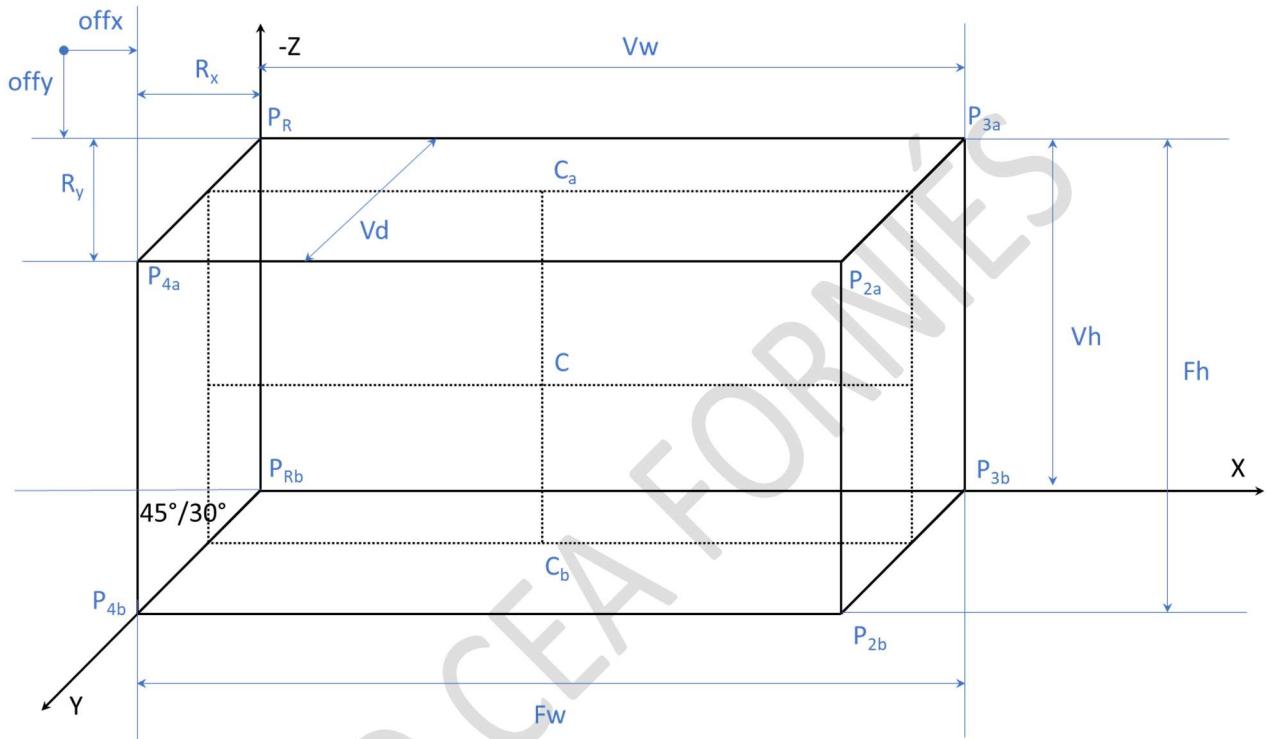
Vamos a suponer que cualquier entidad, y cualquiera de los fotogramas que la representan en una hoja de gráficos, puede enmarcarse en un cubo 3D (o mejor 2.5D). Manejar las intersecciones entre objetos o su ubicación relativa en el espacio, es mucho más eficiente con esta suposición, por otro lado, bastante cercana a la realidad.

La figura en sí, y su representación, dentro de la hoja de gráficos, no necesariamente comienza siempre en la esquina superior izquierda de la misma. Las herramientas para la construcción de hojas de gráficos identifican ese punto con sendos valores de offset (*offsetx* hacia la derecha y *offsety* hacia abajo). No es así, sin embargo, como se calculan los valores 3D relevantes, pero debe ser tenidas en cuenta.

Analicemos las particularidades y cálculos adecuados para cada tipo de perspectiva.

Perspectiva caballera 30 grados aproximada

En esta perspectiva, el eje \overrightarrow{OZ} 3D y el eje \overrightarrow{OX} (según la convención elegida) 3D se disponen ortogonalmente en la pantalla, mientras que el eje \overrightarrow{OY} 3D se dispone formando 45° con el eje $-\overrightarrow{OX}$ o 135° con el $-\overrightarrow{OZ}$ (nuevamente según la convención elegida)¹¹. Detallando todas las variables:



33: Paralelepípedo en perspectiva caballera ($45^\circ/35^\circ$)

En dónde:

- $offx$, es la distancia en el eje horizontal 2D hasta el primer punto de definición útil del fotograma (aparecen típicamente marcadas en las hojas de gráficos),
- $offy$, lo mismo en el eje vertical 2D.
- F_w es el ancho del fotograma según está definido en el fichero gráfico,
- F_h es su alto.

Estos 4 parámetros son básicos en la definición de un fotograma y, por tanto, dados. Por otro lado:

- V_d es la profundidad aparente visual de ese fotograma en perspectiva caballera cuando se observa desde una perspectiva tridimensional,
- V_w es su ancho aparente visual,
- Y V_h es el alto aparente visual.

Como de profundo es la representación plana de una imagen tridimensional, no parece algo que pueda calcularse simplemente a partir de las dimensiones del fotograma (planas). Vamos

¹¹ Se elige esta convención de X e Y por ser la que más se aproxima a como se disponen las coordenadas gráficas

por tanto a suponer, inicialmente, que la profundidad 3D será un parámetro adicional proporcionado junto al resto de valores que representan la proyección ortogonal de un fotograma.

En consecuencia, V_w y V_h deben poderse calcular en función de V_d y de F_w y F_h , así:

$$V_w = F_w - \cos 45^\circ V_d = F_w - \frac{\sqrt{2}}{2} V_d \approx F_w - 0,7071 V_d$$

$$V_h = F_h - \sin 45^\circ V_d = F_h - \frac{\sqrt{2}}{2} V_d \approx F_h - 0,7071 V_d$$

El cálculo de estos valores se presupone una actividad constante en la ejecución de un juego y, por tanto, debe hacerse lo más eficiente posible.

Multiplicar y dividir números reales es mucho más ineficiente y lento que multiplicar y dividir números enteros, y esto es más ineficiente a su vez que multiplicar y dividir por potencias de 2 dado que, para realizar estas últimas operaciones, basta coger su representación en bits y desplazar hacia la izquierda o la derecha tanto bits como indique la potencia de 2 asociada al cálculo a realizar. Por último, desplazar dos veces los bits de una dirección de memoria lleva el doble de tiempo que hacerlo sólo una, por lo que incluso este tipo de operaciones debe minimizarse para maximizar el rendimiento de la ejecución.

Tratemos, por tanto, de acercar los cálculos anteriores a esa situación mediante diversas simplificaciones y / o aproximaciones.

En primer lugar: Si el ángulo formado por el eje $\overrightarrow{-OX}$ con respecto al eje \overrightarrow{OY} no fueran 45° , sino 30° , el valor de V_w y V_h sería, respectivamente:

$$V_w = F_w - \cos 30^\circ V_d = F_w - \frac{\sqrt{3}}{2} V_d \approx F_w - 0,866 V_d$$

$$V_h = F_h - \sin 30^\circ V_d = F_h - \frac{1}{2} V_d$$

Este último mucho más fácil y rápido de calcular que su equivalente en 45° . Sin embargo V_w sería, al menos, igual de complicado y lento de calcular que en caso anterior.

Si, por otro lado, aproximáramos $\sqrt{3}/2$ al racional de entero más próximo posible con un denominador potencia de 2, así. Sea:

$$V'_w = F_w - \frac{7}{8} V_d$$

Se introduciría un error de:

$$e_w = V'_w - V_w = \left(-\frac{7}{8} + \frac{\sqrt{3}}{2} \right) V_d \approx 0,008 V_d$$

Es decir, que con esta aproximación se calcularía siempre una anchura visible aparente ligeramente más grande de lo que debiera ser en la perspectiva caballera 30 grados adoptada. El error introducido sería tanto más grande cuanto más grande fuera la profundidad aparente de la entidad representada, pero en términos generales parece asumible. Veámoslo.

Dado el ancho concreto de un fotograma, y según éste represente una imagen en perspectiva caballera más o menos alargada en el eje \overrightarrow{OX} 3D, podríamos fijar V_d en función de ese ancho, de la siguiente manera:

$$V_d = kF_w$$

Cuanto más grande fuera k , más alargada sería la figura, y cuanto más bajo fuera, más ancha sería. El valor mínimo de k es 0, pero su valor máximo depende de las dimensiones del fotograma. Una cota superior viene determinada por el ancho de este:

$$\text{Proy}V_d \text{ en } \overrightarrow{OX} = \frac{7}{8}V_d = F_w \Rightarrow \frac{7}{8}kF_w = F_w \Rightarrow k = \frac{8}{7}$$

Y la otra cota superior vendrá determinada por el alto de este:

$$\text{Proy}V_d \text{ en } \overrightarrow{OY} = \frac{1}{2}V_d = F_h \Rightarrow \frac{1}{2}kF_w = F_h \Rightarrow k = 2\frac{F_h}{F_w}$$

Luego k puede variar entre:

$$0 \leq k \leq \min\left(\frac{8}{7}, 2\frac{F_h}{F_w}\right)$$

Calculando así k , el error se puede acotar en función del ancho del fotograma, de la siguiente manera:

$$e_w \approx 0,008kF_w$$

$$0 \leq e_w \leq 0,008\frac{8}{7}F_w \Rightarrow 0 \leq e_w \leq 0,0091F_w^{12}$$

Es decir, que el error es tanto más grande cuanto más ancho (2D) sea el fotograma para representar una figura y tanto más pequeño cuando más alto (2D) sea éste. Sin embargo, en el peor de las situaciones, y suponiendo por ejemplo un fotograma de 1000 pixeles, cometeríamos un error de tan sólo 9. El error es, por tanto, despreciable y, con ello podemos asumir que:

$$V'_w \approx V_w$$

Por lo que:

$$V_w = F_w - \frac{7}{8}V_d = F_w - V_d + \frac{1}{8}V_d^{13}$$

Más fácil para ser rápidamente calculado por el ordenador.

Así, cuando $k = 0$, la figura es, en realidad plana en el plano $\overrightarrow{OX} \times \overrightarrow{OZ}$ (no tendría profundidad, sólo anchura y altura). Pero si se quisiera representar un fotograma que fuera plano $\overrightarrow{OY} \times \overrightarrow{OZ}$:

$$V_w = 0 \Rightarrow F_w - \frac{7}{8}kF_w = 0 \Rightarrow 1 - k\frac{7}{8} = 0 \Rightarrow k = \frac{8}{7}$$

¹² Cogiendo el máximo posible de $\min\left(\frac{8}{7}, 2\frac{F_w}{F_h}\right)$

¹³ Son fórmulas más fáciles de calcular por un ordenador.

Pero entonces:

$$2 \frac{F_h}{F_w} = \frac{8}{7} \Rightarrow F_h = \frac{8}{14} F_w = \frac{4}{7} F_w$$

Para que de verdad el fotograma represente una figura plana en ese eje.

Tomemos, por tanto, esta convección y, resumiendo:

$$\begin{aligned} V_d &= kF_w \text{ donde } 0 \leq k \leq \min\left(\frac{8}{7}, 2 \frac{F_h}{F_w}\right) \\ V_w &= F_w - kF_w + \frac{1}{8}kF_w \\ V_h &= F_h - \frac{1}{2}kF_w \end{aligned}$$

27: Dimensiones visuales aproximadas generales (k) de un fotograma en perspectiva caballera 30 grados

En este caso:

$$V_h \geq 0 \Rightarrow F_h \geq \frac{1}{2}kF_w$$

28: Verificación a realizar para valores genéricos de k entre las dimensiones del fotograma en caballera

Tratemos de simplificar algo más:

Si bien k puede adoptar valores continuos entre 0 y $\min\left(\frac{8}{7}, 2 \frac{F_w}{F_h}\right)$, consideremos que sólo pudiera adoptar valores equivalentes a una fracción potencia de 2 de la forma: $k = \frac{1}{2^p}$ y limitemos los valores de p a 0, 1, 2 o 3 (discretos) para representar respectivamente figuras largas ($k = 0$), cuadradas ($k = 1$) anchas ($k = 2$) y muy anchas ($k = 3$). Es una simplificación que podemos permitirnos a la hora de definir sprites.

Con ninguno de esos valores se superarían los límites de k definidos anteriormente, pero tampoco podríamos conseguir figuras completamente planas en el plano $\overrightarrow{OY} \times \overrightarrow{OZ}$, aunque si en el $\overrightarrow{OX} \times \overrightarrow{OZ}$.

En cualquier caso, parece una aproximación adecuada.

Entonces podemos admitir:

$$\begin{aligned} V_d &= \frac{1}{2^p} F_w = F_w \gg p; \quad p = \{0, 1, 2, 3\} \\ V_w &= F_w - \frac{1}{2^p} F_w + \frac{1}{2^{(p+3)}} F_w = F_w - (F_w \gg p) + (F_w \gg (p+3)) \\ V_h &= F_h - \frac{1}{2^{(p+1)}} F_w = F_h - (F_w \gg (p+1)) \end{aligned}$$

29: Dimensiones visuales aproximadas (p) generales de un fotograma en perspectiva caballera 30 grados.

Dónde $>> p$ es el operador “right bit shift”, equivalente a dividir por 2^p un número entero. Todos los cálculos son apropiados para operaciones enteras. Calculando todas las dimensiones en función de los parámetros que definen el fotograma.

Y que podemos establecer de la siguiente manera:

	V_w	V_d	V_h
$p = 0$	$F_w \gg 3$	F_w	$F_h - (F_w \gg 1)$
$p = 1$	$F_w - (F_w \gg 1) + (F_w \gg 4)$	$F_w \gg 1$	$F_h - (F_w \gg 2)$
$p = 2$	$F_w - (F_w \gg 2) + (F_w \gg 5)$	$F_w \gg 2$	$F_h - (F_w \gg 3)$
$p = 3$	$F_w - (F_w \gg 3) + (F_w \gg 6)$	$F_w \gg 3$	$F_h - (F_w \gg 4)$

30: Cuadro de dimensiones para diferentes valores de p en perspectiva caballera

Y en ningún caso V_h puede ser negativo luego; se debe verificar que, por ejemplo, para $p = 0$:

$$F_h - F_w \gg 1 \geq 0 \Rightarrow F_h \geq F_w \gg 1 \Rightarrow F_h \geq \frac{1}{2} F_w$$

De igual manera:

$p = 0$	$F_h \geq \frac{1}{2} F_w$
$p = 1$	$F_h \geq \frac{1}{4} F_w$
$p = 2$	$F_h \geq \frac{1}{8} F_w$
$p = 3$	$F_h \geq \frac{1}{16} F_w$

31: Valores mínimos de la dimensión vertical según diferentes valores de p en perspectiva caballera

Comprobaciones que habría que hacer al cargar cada fotograma.

Calculemos ahora los puntos representativos del cubo en caballera que imaginariamente envolvería a cada fotograma (tal útil, por ejemplo, para el cálculo de puntos de colisión con otros elementos del juego) en función de las dimensiones V_d , V_w y V_h y de los valores que definen al fotograma (F_w y F_h).

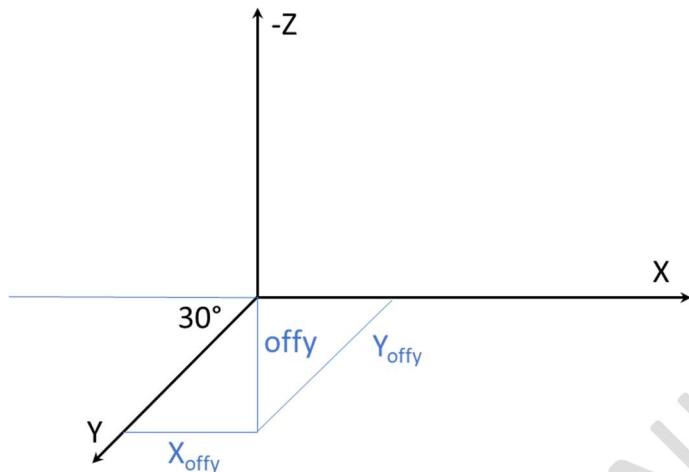
Se va a tomar como base de los puntos representativos de ese cubo, la esquina superior izquierda del fotograma. Para ello nos hace falta conocer el valor de R_x :

$$R_x = F_w - V_w$$

Podríamos entonces pensar que el punto $P_{1a} = P_R$ tuviera de coordenadas:

$$P_{1a} = P_R = \begin{pmatrix} offx + R_x \\ offy \\ 0 \end{pmatrix}$$

Sin embargo, esas coordenadas han sido calculadas con una perspectiva 2D y no una caballera. Trasformemos esos valores en coordenadas de los ejes \overrightarrow{OX} y \overrightarrow{OY} . Según la figura:



34: Coordenadas en caballera del offset

Sus coordenadas estarían relacionadas según las siguientes ecuaciones:

$$Y_{offy} \cos 30^\circ = X_{offy}$$

$$Y_{offy} \sin 30^\circ = offy$$

Y despejando:

$$Y_{offy} = \frac{1}{\sin 30^\circ} offy = 2offy$$

$$X_{offy} = 2offy \frac{\sqrt{3}}{2} \approx \frac{7}{4} offy = 2offy - \frac{1}{4} offy$$

O en formato potencias de dos:

$$Y_{offy} = offy \ll 1$$

$$X_{offy} = (offy \ll 1) - (offy \gg 2)$$

Por tanto, las coordenadas del punto de referencia serían:

$$R_x = F_w - V_w$$

$$P_{1a} = P_R = \begin{pmatrix} (offx + R_x) + (offy \ll 1) - (offy \gg 2) \\ offy \ll 1 \\ 0 \end{pmatrix}$$

Con lo que las coordenadas de los puntos del cubo en perspectiva caballera que rodea a la figura serían lo siguientes:

$$R_x = F_w - V_w$$

$$\mathbf{P}_{1a} = \mathbf{P}_R = \begin{pmatrix} (\text{offx} + R_x) + (\text{offy} \ll 1) - (\text{offy} \gg 2) \\ \text{offy} \ll 1 \\ \mathbf{0} \end{pmatrix}$$

$$\mathbf{P}_{2a} = \mathbf{P}_R + \begin{pmatrix} V_d \\ V_w \\ \mathbf{0} \end{pmatrix}$$

$$\mathbf{P}_{3a} = \mathbf{P}_R + \begin{pmatrix} \mathbf{0} \\ V_w \\ \mathbf{0} \end{pmatrix}$$

$$\mathbf{P}_{4a} = \mathbf{P}_R + \begin{pmatrix} V_d \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}$$

$$\mathbf{P}_{xb} = \mathbf{P}_{xa} - \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ V_h \end{pmatrix}; \text{ donde } x = \{1, 2, 3, 4\}$$

32: Puntos característicos del paralelepípedo que envuelve a un fotograma en caballera en representación 3D

Calculando todos los posibles valores de R_x :

	\mathbf{V}_w	R_x
$p = 0$	$F_w \gg 3$	$F_w - (F_w \gg 3)$
$p = 1$	$F_w - (F_w \gg 1) + (F_w \gg 4)$	$(F_w \gg 1) - (F_w \gg 4)$
$p = 2$	$F_w - (F_w \gg 2) + (F_w \gg 5)$	$(F_w \gg 2) - (F_w \gg 5)$
$p = 3$	$F_w - (F_w \gg 3) + (F_w \gg 6)$	$(F_w \gg 3) - (F_w \gg 6)$

Y el punto central del mismo sería:

$$\mathbf{P}_c = \frac{(\mathbf{P}_{2b} - \mathbf{P}_{1a})}{2} = \frac{(\mathbf{P}_R + \begin{pmatrix} V_d \\ V_w \\ V_h \end{pmatrix} - \mathbf{P}_R)}{2} = \frac{1}{2} \begin{pmatrix} V_d \\ V_w \\ V_h \end{pmatrix}$$

En visualización pura 2D, esos puntos serían:

$$R_x = F_w - V_w = V_d \cos 30^\circ = \frac{\sqrt{3}}{2} V_d \approx \frac{7}{8} V_d = \frac{7}{8} kF_w = kF_w - \frac{1}{8} kF_w = \frac{1}{2^p} F_w - \frac{1}{2^{(p+3)}} F_w$$

$$R_y = V_d \sin 30^\circ = \frac{1}{2} V_d = \frac{1}{2} kF_w = \frac{1}{2^{(p+1)}} F_w$$

Y, en conclusión:

$$R_x = (F_w \gg p) - (F_w \gg (p+3))$$

$$R_y = (F_w \gg (p+1))$$

$$P_{1a} = P_R = \begin{pmatrix} offx + R_x \\ offy \end{pmatrix}$$

$$P_{2a} = P_R + \begin{pmatrix} V_w - R_x \\ R_y \end{pmatrix} = \begin{pmatrix} offx + F_w - R_x \\ offy + R_y \end{pmatrix}$$

$$P_{3a} = P_R + \begin{pmatrix} V_w \\ 0 \end{pmatrix} = \begin{pmatrix} offx + F_w \\ offy \end{pmatrix}$$

$$P_{4a} = P_R + \begin{pmatrix} -R_x \\ R_y \end{pmatrix} = \begin{pmatrix} offx \\ offy + R_y \end{pmatrix}$$

$$P_{xb} = P_{xa} + \begin{pmatrix} 0 \\ F_h - R_y \end{pmatrix} = \text{donde } x = \{1,2,3,4\}$$

$$P_c = \frac{(P_{2b} - P_{1a})}{2} = \frac{(P_R + \begin{pmatrix} offx + F_w - R_x \\ offy + F_h \end{pmatrix} - P_R - \begin{pmatrix} offx + R_x \\ offy \end{pmatrix})}{2} = \frac{1}{2} \begin{pmatrix} F_w \\ F_h \end{pmatrix}$$

33: Puntos característicos 2D de un fotograma en perspectiva caballera

Calculado en función de los parámetros que definen el fotograma.

Perspectiva caballera 30 grados rápida

Introducimos aquí el término de la perspectiva caballera rápida.

Si la aproximación que se hizo a $\cos 30^\circ$, hubiera sido 1 en lugar de $\frac{7}{8}$, el error introducido en el cálculo de V_w hubiera sido:

$$e_w = V'_w - V_w = \left(-1 + \frac{\sqrt{3}}{2}\right) V_d \approx -0,1339 V_d$$

Que vamos a considerar, de momento, aceptable.

Entonces, si:

$$V_d = k F_w$$

El valor de k oscilaría entre:

$$0 \leq k \leq \min\left(1,2 \frac{F_w}{F_h}\right)$$

Y las dimensiones visuales tendrían como fórmula:

$$\begin{aligned} V_w &= F_w - V_d = F_w - k F_w \\ V_h &= F_h - \frac{1}{2} V_d = F_h - \frac{1}{2} k F_w \end{aligned}$$

Y si ponemos k como una potencia de 2, entonces:

$$\begin{aligned} V_d &= \frac{1}{2^p} F_w = F_w \gg p; \quad p = \{0, 1, 2, 3\} \\ V_w &= F_w - \frac{1}{2^p} F_w = F_w - (F_w \gg p) \\ V_h &= F_h - \frac{1}{2^{(p+1)}} F_w = F_h - (F_w \gg (p+1)) \end{aligned}$$

34: Ecuaciones de las dimensiones de un fotograma en perspectiva caballera 30 grados rápida

En este caso, las tablas de cálculo quedarían:

	V_w	V_d	V_h
$p = 0$	$F_w \gg 2$	F_w	$F_h - (F_w \gg 1)$
$p = 1$	$F_w - (F_w \gg 1)$	$F_w \gg 1$	$F_h - (F_w \gg 2)$
$p = 2$	$F_w - (F_w \gg 2)$	$F_w \gg 2$	$F_h - (F_w \gg 3)$
$p = 3$	$F_w - (F_w \gg 3)$	$F_w \gg 3$	$F_h - (F_w \gg 4)$

35: Ecuaciones de las dimensiones visuales de un fotograma en perspectiva caballera 30 grados rápida en base a p

Mientras que la de límites seguiría siendo igual:

$p = 0$	$F_h \geq \frac{1}{2}F_w \Rightarrow F_h \geq (F_w \gg 1)$
$p = 1$	$F_h \geq \frac{1}{4}F_w \Rightarrow F_h \geq (F_w \gg 2)$
$p = 2$	$F_h \geq \frac{1}{8}F_w \Rightarrow F_h \geq (F_w \gg 3)$
$p = 3$	$F_h \geq \frac{1}{16}F_w \Rightarrow F_h \geq (F_w \gg 4)$

Y la de valores de R_x :

	V_w	R_x
$p = 0$	$F_w \gg 2$	$F_w - (F_w \gg 2)$
$p = 1$	$F_w - (F_w \gg 1)$	$F_w \gg 1$
$p = 2$	$F_w - (F_w \gg 2)$	$F_w \gg 2$
$p = 3$	$F_w - (F_w \gg 3)$	$F_w \gg 3$

Las ecuaciones para determinar los puntos del paralelepípedo que rodea al fotograma son casi las mismas. En este caso podríamos prescindir de la aproximación máxima de $offy$, dejándolo en:

$$Y_{offy} = offy \ll 1$$

$$X_{offy} = offy \ll 1$$

Por lo que:

$$P_{1a} = P_R = \begin{pmatrix} (offx + R_x) + (offy \ll 1) \\ offy \ll 1 \\ 0 \end{pmatrix}$$

Para calcular la posición de los puntos relativos en 2D, tendríamos que hacer algo semejante. En dónde:

$$R_x = F_w \gg p.$$

El resto de los cálculos seguirían igual.

Esta aproximación es válida mientras los fotogramas sean suficientemente pequeños. En un juego basado en teselas, estas suelen ser de tamaños relativamente pequeños.

Ejemplos:

Para el caso en el que la tesela sea considerada ancha; es decir que $p = 2$:

$$V_d = F_w \gg 2; V_w = F_w - (F_w \gg 2); V_h = F_h - (F_w \gg 3)$$

Imaginemos una tesela de $F_w = 64$; y $F_h = 8$, entonces:

$$V_d = 16; V_w = 48; V_h = 0$$

Que representa, por ejemplo, una lámina en el suelo (no tiene grosor)

Un fotograma con una altura menor que 8 no sería viable.

Si $p = 1$; es decir, fuera “cuadrada”:

$$V_d = F_w \gg 1; V_w = F_w - (F_w \gg 1); V_h = F_h - (F_w \gg 2)$$

Con ese valor, imaginemos ahora una tesela de $F_w = 64$; y $F_h = 32$, entonces:

$$V_d = 32; V_w = 32; V_h = 16$$

Ahora representando una placa cuadrada con cierto grosor.

Un fotograma con una altura menor que 16, no sería viable.

Como puede apreciar estas fórmulas permiten el dibujo de teselas y hojas de fotogramas de las mismas mucho más rápidamente que las anteriores. Siendo por ello las más habitualmente utilizadas en el desarrollo de juegos.

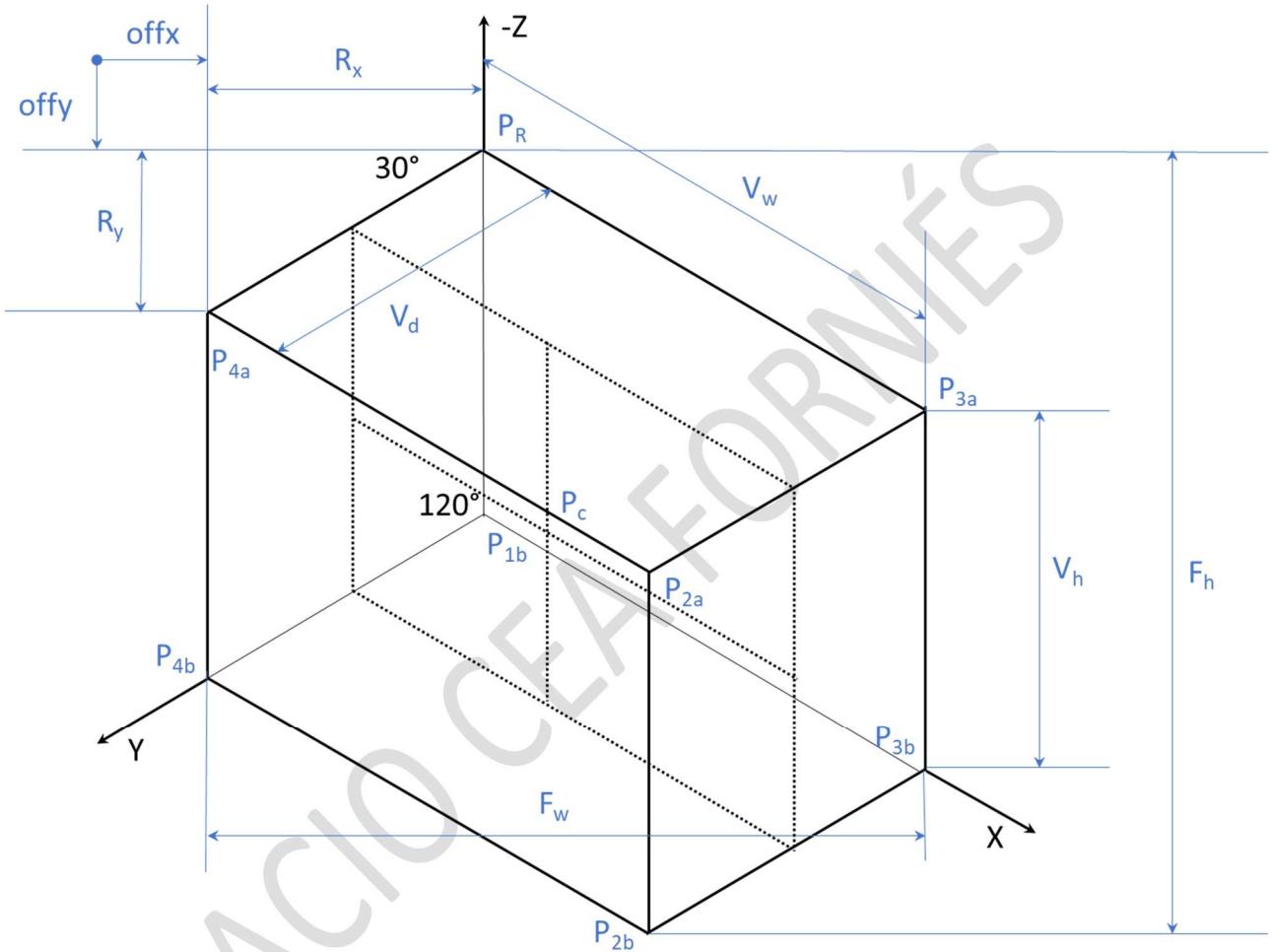
Posición relativa en Perspectiva caballera

A la hora de dibujar diferentes elementos en un juego basado en perspectiva caballera (o en cualquier otra) es necesario poder decidir de manera sencilla cual pintar primero y cual después.

Para

Isométrica aproximada

En esta perspectiva, los 3 ejes 3D se disponen sobre el plano formando ángulo de 120° entre sí. Detallando todas sus variables:



35: Paralelepípedo en perspectiva isométrica

En dónde las diferentes variables tienen un significado equivalente a los que se describieron en la perspectiva caballera.

Y, de igual manera a como sucedía en aquella, las dimensiones fundamentales de la visualización isométrica también se pueden calcular en función de los parámetros que definen los fotogramas de la hoja de gráficos. Calculemos algunas equivalencias que pueden deducirse de la observación de la figura (5):

$$F_w = \cos 30^\circ V_w + \cos 30^\circ V_d = \frac{\sqrt{3}}{2} (V_w + V_d)$$

$$F_h = V_h + \sin 30^\circ V_w + \sin 30^\circ V_d = V_h + \frac{1}{2}(V_w + V_d)$$

O todo en función de V_d :

$$V_w = \frac{2}{\sqrt{3}}F_w - V_d \approx 1,154F_w - V_d$$

$$V_h = F_h - \frac{1}{2}(V_w + V_d) = F_h - \frac{1}{2}\left(\frac{2}{\sqrt{3}}F_w - V_d + V_d\right) = F_h - \frac{1}{\sqrt{3}}F_w \approx F_h - 0,577F_w$$

Intentemos aproximar la primera expresión a un racional en el que se usen en la medida de lo posible potencias de 2. Al igual que sucedió en el caso de la perspectiva caballera, esa aproximación nos servirá luego para plantear cálculos más rápidos de ejecutar en un ordenador. Sea:

$$V'_w = \frac{9}{8}F_w - V_d$$

El error introducido es:

$$e_w = V'_w - V_w = \left(\frac{9}{8} - \frac{2}{\sqrt{3}}\right)F_w \approx 0,029F_w$$

Es decir, que con esta aproximación se calcularía siempre una anchura visible aparente ligeramente más grande de lo que debiera ser en la perspectiva isométrica adoptada. El error introducido sería tanto más grande cuanto más grande fuera el ancho del fotograma. De tal manera que para un fotograma de 1000 pixeles de ancho introduciríamos un error de casi 30. Un error, sin duda, mucho mayor que el que se introducía en la perspectiva caballera 30 grados con una aproximación semejante (punto anterior). En cualquier caso, lo vamos a considerar despreciable al no ser habitual fotogramas de entidades tan grandes como el indicado. Es decir:

$$V'_w \approx V_w$$

De esta aproximación se deduce que:

$$\frac{9}{8} \approx \frac{2}{\sqrt{3}} \Rightarrow \frac{9}{16} \approx \frac{1}{\sqrt{3}}$$

Por lo que:

$$V_w = \frac{9}{8}F_w - V_d$$

$$V_h = F_h - \frac{9}{16}F_w$$

Dado el ancho concreto de un fotograma, y según éste represente una imagen en perspectiva isométrica más o menos alargada en el eje \vec{OX} 3D, podríamos fijar V_d en función de ese ancho, de la siguiente manera:

$$V_d = kF_w$$

Cuanto más grande fuera k , más alargada sería la figura, y cuanto más baja fuera, más ancha sería. En ese caso:

$$V_w = \frac{9}{8}F_w - kF_w = F_w + \frac{1}{8}F_w - kF_w$$

$$V_h = F_h - \frac{9}{16}F_w = F_h - \frac{1}{2}F_w - \frac{1}{16}F_w$$

Como se puede apreciar V_h no depende de V_d , ni del valor de k . Además, nunca puede ser negativa, por lo que:

$$V_h \geq 0 \Rightarrow F_h \geq \frac{9}{16}F_w$$

Tampoco lo puede ser V_w , por lo que:

$$V_w \geq 0 \Rightarrow \frac{9}{8}F_w - kF_w \geq 0 \Rightarrow k \leq \frac{9}{8}$$

Por tanto:

$$0 \leq k \leq \frac{9}{8}$$

Cuando k es 0 representa una figura plana ubicada en el plano $\overrightarrow{OX} \times \overrightarrow{OZ}$, mientras que cuando es $\frac{9}{8}$ representa lo mismo, pero en plano $\overrightarrow{OY} \times \overrightarrow{OZ}$.

Y si también pusiéramos k como fracción de una potencia de 2.

$$V_d = \frac{1}{2^p}F_w = F_w \gg p; p = \{0, 1, 2, 3\}$$

$$V_w = F_w + \frac{1}{2^3}F_w - \frac{1}{2^p}F_w = F_w + (F_w \gg 3) - (F_w \gg p)$$

$$V_h = F_h - \frac{1}{2}F_w - \frac{1}{2^4}F_w = F_h - (F_w \gg 1) - (F_w \gg 4)$$

36:Ecuaciones de las dimensiones de un fotograma en perspectiva isométrica aproximada

En este caso, podría surgir la pregunta del valor de p óptimo en el que $V_d = V_w$:

$$kF_w = F_w + \frac{1}{8}F_w - kF_w$$

$$k = \frac{1}{2} \left(1 + \frac{1}{8}\right); k = \frac{9}{16}; k \approx \frac{1}{2}; \text{ ó } p \approx 1$$

Por tanto, en este caso, haciendo variar el valor de p entre 0, 1, 2 o 3, representaríamos igualmente figuras alargadas, cuadradas, anchas y muy anchas.

En este caso, la tabla de dimensiones para los 4 valores de p , sería:

	V_w	V_d	V_h
$p = 0$	$F_w \gg 3$	F_w	
$p = 1$	$F_w + (F_w \gg 3) - (F_w \gg 1)$	$F_w \gg 1$	
$p = 2$	$F_w + (F_w \gg 3) - (F_w \gg 2)$	$F_w \gg 2$	$F_h - (F_w \gg 1) - (F_w \gg 4)$
$p = 3$	F_w	$F_w \gg 3$	

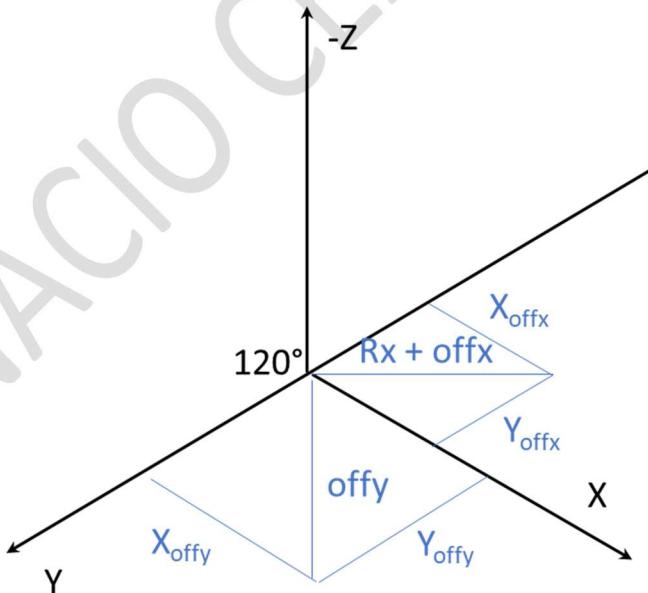
Y como, en ningún caso V_h puede ser negativa se debe verificar siempre (para cualquier valor de p) que:

$$F_h - (F_w \gg 1) - (F_w \gg 4) \geq 0 \Rightarrow F_h \geq (F_w \gg 1) + (F_w \gg 4)$$

Calculemos ahora los puntos del cubo isométrico que rodea imaginariamente a cada fotograma. De igual manera que nos sucedía en perspectiva caballera, tomaremos como referencia el punto superior izquierdo.

$$R_x = F_w - V_w \cos 30^\circ \approx F_w - V_w + (V_w \gg 3)$$

Y transformado esos valores en coordenadas isométricas según se puede deducir de la siguiente figura:



36: Representación isométrica de los puntos offset

En el eje OX:

$$X_{offx} \cos 30^\circ - Y_{offx} \cos 30^\circ = R_x + offx$$

$$X_{offx} \sin 30^\circ = -Y_{offx} \sin 30^\circ$$

Y despejando:

$$X_{offx} = -Y_{offx}$$

$$X_{offx} = \frac{1}{\sqrt{3}}(R_x + offx) \approx \frac{9}{16}(R_x + offx) = \frac{1}{2}(R_x + offx) + \frac{1}{16}(R_x + offx)$$

O en formato de potencia de dos:

$$X_{offx} = ((R_x + offx) \gg 1) + ((R_x + offx) \gg 4)$$

$$Y_{offx} = -X_{offx}$$

Otro tanto podría hacerse con el offset en el eje \overrightarrow{OY} :

$$X_{offy} \cos 60^\circ + Y_{offy} \cos 60^\circ = offy$$

$$X_{offy} = Y_{offy}$$

Y despejando:

$$X_{offy} = offy$$

$$Y_{offy} = X_{offy}$$

O en forma de potencias de dos:

$$X_{offy} = offy$$

$$Y_{offy} = X_{offy}$$

Por lo que las coordenadas del punto de referencia serían:

$$R_x = (F_w \gg p) + (F_w \gg 6) - (F_w \gg (p+3))$$

$$P_{1a} = P_R = \begin{pmatrix} ((R_x + offx) \gg 1) + ((R_x + offx) \gg 4) + offy \\ -((R_x + offx) \gg 1) - ((R_x + offx) \gg 4) + offy \\ 0 \end{pmatrix}$$

Con lo que las coordenadas de todos los puntos quedan

$$R_x = (F_w \gg p) + (F_w \gg 6) - (F_w \gg (p+3))$$

$$P_{1a} = P_R = \begin{pmatrix} ((R_x + offx) \gg 1) + ((R_x + offx) \gg 4) + offy \\ -((R_x + offx) \gg 1) - ((R_x + offx) \gg 4) + offy \\ 0 \end{pmatrix}$$

$$P_{2a} = P_R + \begin{pmatrix} V_d \\ V_w \\ 0 \end{pmatrix}$$

$$P_{3a} = P_R + \begin{pmatrix} 0 \\ V_w \\ 0 \end{pmatrix}$$

$$P_{4a} = P_R + \begin{pmatrix} V_d \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}$$

$$P_{xb} = P_{xa} - \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ V_h \end{pmatrix}; \text{ donde } x = \{1, 2, 3, 4\}$$

37: Puntos característicos del paralelepípedo que envuelve a un fotograma en isométrica en representación 3D

Calculando todos los posibles valores de R_x :

	V_w	R_x
$p = 0$	$F_w \gg 3$	$F_w + (F_w \gg 6) - (F_w \gg 3)$
$p = 1$	$F_w + (F_w \gg 3) - (F_w \gg 1)$	$(F_w \gg 1) + (F_w \gg 6) - (F_w \gg 4)$
$p = 2$	$F_w + (F_w \gg 3) - (F_w \gg 2)$	$(F_w \gg 2) + (F_w \gg 6) - (F_w \gg 5)$
$p = 3$	F_w	$F_w \gg 3$

Y las coordenadas de esos puntos dentro del fotograma (2D), serían, en función de las dimensiones de este:

$$R_x = (F_w \gg p) + (F_w \gg 6) - (F_w \gg (p + 3))$$

$$R_y = (F_w \gg (p + 1))$$

$$P_{1a} = P_R = \begin{pmatrix} offx + R_x \\ offy \end{pmatrix}$$

$$P_{2a} = P_R + \begin{pmatrix} V_w - R_x \\ R_y \end{pmatrix} = \begin{pmatrix} offx + F_w - R_x \\ offy + R_y \end{pmatrix}$$

$$P_{3a} = P_R + \begin{pmatrix} V_w \\ 0 \end{pmatrix} = \begin{pmatrix} offx + F_w \\ offy \end{pmatrix}$$

$$P_{4a} = P_R + \begin{pmatrix} -R_x \\ R_y \end{pmatrix} = \begin{pmatrix} offx \\ offy + R_y \end{pmatrix}$$

$$P_{xb} = P_{xa} + \begin{pmatrix} 0 \\ F_h - R_y \end{pmatrix} = \text{donde } x = \{1, 2, 3, 4\}$$

$$P_c = \frac{(P_{2b} - P_{1a})}{2} = \frac{(P_R + \begin{pmatrix} offx + F_w - R_x \\ offy + F_h \end{pmatrix} - P_R - \begin{pmatrix} offx + R_x \\ offy \end{pmatrix})}{2} = \frac{1}{2} \begin{pmatrix} F_w \\ F_h \end{pmatrix}$$

38: Puntos característicos 2D de un fotograma en perspectiva isométrica

Perspectiva isométrica rápida

Al igual que hicimos en el caso de la perspectiva caballera, si hacemos que:

$$1 \approx \frac{2}{\sqrt{3}} \Rightarrow \frac{1}{2} \approx \frac{1}{\sqrt{3}}$$

Entonces:

$$V_w = F_w - V_d$$

$$V_h = F_h - \frac{1}{2}F_w$$

Y nuevamente si:

$$V_d = kF_w$$

Las fórmulas de las dimensiones quedarían:

$$V_d = \frac{1}{2^p}F_w = F_w \gg p; p = \{0, 1, 2, 3\}$$

$$V_w = F_w - \frac{1}{2^p}F_w = F_w - (F_w \gg p)$$

$$V_h = F_h - \frac{1}{2}F_w = F_h - (F_w \gg 1)$$

39: Ecuaciones de las dimensiones visuales de un fotograma en perspectiva isométrica rápida

Las tablas de cálculo quedarían ahora:

	V_w	V_d	V_h
$p = 0$	0	F_w	
$p = 1$	$F_w - (F_w \gg 1)$	$F_w \gg 1$	
$p = 2$	$F_w - (F_w \gg 2)$	$F_w \gg 2$	$F_h - (F_w \gg 1)$
$p = 3$	$F_w - (F_w \gg 3)$	$F_w \gg 3$	

40: Ecuaciones de las dimensiones visuales de un fotograma en perspectiva isométrica rápida en base a p

En este caso, las coordenadas del paralelepípedo que le envuelve cambian, al cambiar el valor de $R_x = F_w - V_w$, por lo que se quedaría como:

	V_w	R_x
$p = 0$	0	F_w
$p = 1$	$F_w - (F_w \gg 1)$	$F_w \gg 1$
$p = 2$	$F_w - (F_w \gg 2)$	$F_w \gg 2$
$p = 3$	$F_w - (F_w \gg 3)$	$F_w \gg 3$

Y, en este caso, las coordenadas del punto de referencia serían:

$$P_{1a} = P_R = \begin{pmatrix} ((R_x + offx) \gg 1) + offy \\ -((R_x + offx) \gg 1) + offy \\ 0 \end{pmatrix}$$

Y el resto se calcularían igual:

Nuevamente esta aproximación es válida mientras los fotogramas sean suficientemente pequeños. En un juego basado en teselas, estás suelen ser de tamaños relativamente pequeños.

Ejemplos:

Para el caso en el que la tesela sea considerada ancha; es decir que $p = 2$:

$$V_d = F_w \gg 2; V_w = F_w - (F_w \gg 2); V_h = F_h - (F_w \gg 1)$$

Imaginemos una tesela de $F_w = 64$; y $F_h = 32$, entonces:

$$V_d = 16; V_w = 48; V_h = 0$$

Que representa, por ejemplo, una lámina en el suelo (no tiene grosor)

Un fotograma con una altura menor que 32 no sería viable.

Si $p = 1$; es decir, fuera “cuadrada”:

$$V_d = F_w \gg 1; V_w = F_w - (F_w \gg 1); V_h = F_h - (F_w \gg 1)$$

Con ese valor, imaginemos ahora la misma tesela de $F_w = 64$; y $F_h = 64$, entonces:

$$V_d = 32; V_w = 32; V_h = 32$$

Ahora representando una placa cuadrada con cierto grosor.

Un fotograma con una altura menor que 32, no sería viable.

Como puede apreciar estas fórmulas permiten el dibujo de teselas y hojas de fotogramas de las mismas mucho más rápidamente que las anteriores. Siendo por ello las más habitualmente utilizadas en el desarrollo de juegos en perspectiva isométrica.

Posición relativa en Isométrica

A la hora de dibujar en isométrica es determinante decidir qué elementos están delante o detrás de que otros. Es decir, decidir el orden en el que van a ser dibujados para transmitir mejor la sensación de realismo.

Algoritmo básico

Para determinar el algoritmo más sencillo que mejor puede servir a este propósito, fijemos dos hipótesis iniciales: Que todos los elementos que forman parte de la escena tienen un tamaño más o menos comparable; es decir que no hay objetos enormes frente a objetos minúsculos en términos de píxeles ocupados en la pantalla (y con respecto a la escala de la proyección isométrica), y que cualquiera de esos elementos puede ser identificado por un punto representativo (por ejemplo, su centro).

En una proyección isométrica el plano de proyección tiene de vector normal:

$$n = \frac{1}{\sqrt{3}}(1 \quad 1 \quad 1)$$

Y el punto de vista, dado que se trata de una proyección cilíndrica, lo podemos ubicar en algún punto, tendiendo al infinito, dentro de ese vector normal, dado por la expresión:

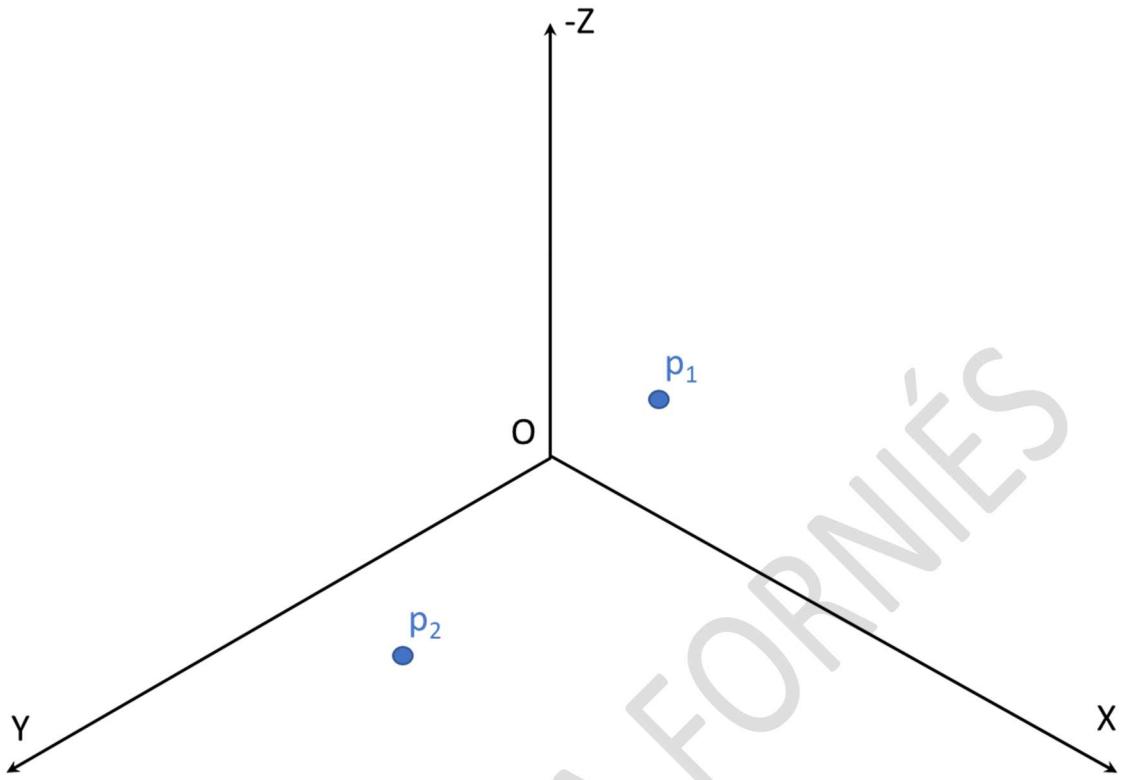
$$F = d(1 \quad 1 \quad 1)$$

Supongamos dos puntos cualesquiera dentro del espacio a proyectar:

$$p_a = (x_a \quad y_a \quad z_a)$$

$$p_b = (x_b \quad y_b \quad z_b)$$

Según la figura:



Sus distancias relativas al punto de vista (F), serían entonces:

$$d_a = |\overrightarrow{p_a F}|^2 = (d - x_a)^2 + (d - y_a)^2 + (d - z_a)^2$$

$$d_b = |\overrightarrow{p_b F}|^2 = (d - x_b)^2 + (d - y_b)^2 + (d - z_b)^2$$

Y si d_a es mayor que d_b , p_a estará más lejos del punto de vista que p_b y por tanto habrá que dibujarlo primero. Este razonamiento es la base de nuestro algoritmo. Profundicemos en ello.
¿En qué casos d_a es mayor que d_b ?

$$d_a > d_b$$

$$(d - x_a)^2 + (d - y_a)^2 + (d - z_a)^2 > (d - x_b)^2 + (d - y_b)^2 + (d - z_b)^2$$

$$x_a^2 + y_a^2 + z_a^2 - 2d(x_a + y_a + z_a) > x_b^2 + y_b^2 + z_b^2 - 2d(x_b + y_b + z_b)$$

Cuando d tiende a infinito, los términos que no dependen de él se vuelven irrelevantes, por lo que eliminándolos y, entonces, dividiendo ambos miembros de la inecuación por $-2d$, tenemos (ojo que la inecuación cambia de sentido):

$$d_a > d_b \Rightarrow (x_a + y_a + z_a) < (x_b + y_b + z_b)$$

41: Ecuación para decidir qué elemento está detrás de otro en una proyección isométrica

Muy sencilla de recordar y programar.

De cara a decidir si pinto el elemento a o el b primero, será aquél que tenga un menor valor de la suma de las tres coordenadas de su centro de referencia. Para poder calcular con facilidad el centro de referencia de cualquier elemento, lo vamos a suponer envuelto en un

paralelepípedo de referencia que es justo lo que hemos tratado a lo largo de todos los puntos anteriores.

Por tanto, el algoritmo para decidir el orden para dibujar una lista de elementos es un sencillo método de ordenación en el que la fórmula de comparación de cada par de elementos es la descrita anteriormente.

¿Y si hubiera dos elementos en el que la suma mencionada fuera igual? Si se cumplen estrictamente las hipótesis fijadas al principio de este punto, es más que probable que ninguno de los elementos se superponga con el otro en ninguna parte, por lo que sería indiferente cual pintar. Como, en los algoritmos de ordenación es siempre obligatorio incorporar una decisión de desempate, podríamos fijar ésta basándose en el identificador de cada elemento, por lo que es importante que cada uno tenga uno diferente.

Y, ¿si hubiera algún elemento dentro de la escena mucho más grande que otro? Nuestras hipótesis no serían entonces válidas, por lo que, para seguir empleando esta fórmula, deberíamos procurar que esto no ocurriera dividiendo, por ejemplo, el elemento a dibujar en varios sub – elementos. En un juego basado en tiles y hojas de gráficos definidos en simples ficheros de gráficos es propio para esto.

Algoritmos evolucionados

En lugar de tomar como referencia un punto representativo de cada elemento, tomemos los planos que lo delimitan que, por consenso, estimaremos alineados con los ejes de coordenadas. Todo elemento tiene 6 planos que lo delimitan. Pero tomemos aquellos cuya parte visible se orienta según los vectores normales que definen cada eje. Según esta nueva definición tendríamos 3 rectángulos límite: el de la derecha (eje OX), el de abajo (eje OY) y el arriba (eje -OZ).

Posición relativa en otras perspectivas

En otro tipo de proyecciones, la posición relativa entre objetivos es mucho más sencilla de determinar. En perspectiva cónica es obvio. Estará más lejos del observador, y por tanto, habrá de ser dibujado en primera instancia, el objeto cuyo punto representativo se encuentre a mayor distancia del punto de observación. En perspectiva caballera, estará más lejos del observador aquel cuya coordenada y esté más cerca de 0 (sea menor);

IGNACIO CEA FORNIÉS

Bloque 4: Detección de colisiones en juegos

IGNACIO CEA FORNIÉS

Introducción

Otra de las tareas clave en un juego es la detección de colisiones entre los diferentes elementos que formen parte de él. Llevar a cabo esta tarea entre figuras que pueden tener formas y aspectos muy diversos puede ser complicada y, sin duda, que hay que buscar formas simplificadas de llevarlo a cabo.

Dentro de esa simplificación podemos entender que cualquier figura esté enmarcada en un paralelepípedo en el caso de figuras 3D o 2.5D, o en un rectángulo, en el caso de figuras 2D.

Empecemos analizando en detalle este tipo de figuras.

Rectángulos

Como decíamos en la introducción, los rectángulos juegan un papel muy importante en el funcionamiento de un juego, sobre todo para simplificar el cálculo de intersecciones entre los diversos objetos que se mueven en él.

En un mundo 2D, por ejemplo, podemos considerar cada fotograma contenido en un rectángulo, por lo que determinar las intersecciones de éste con otros es detectar intersecciones entre rectángulos.

Rectángulos orientados, ubicados en un plano y alineados con los ejes

Supongamos inicialmente rectángulos r situados en el plano XY o paralelo ($\vec{n} = [0,0, \neq 0]$). Sean A, B, C, D sus 4 vértices, orientados estos en el sentido de las agujas del reloj cuando se miran desde el extremo del vector \vec{n} que pasa por su centro, y empezando por la esquina superior izquierda. Sean $\overrightarrow{AB}, \overrightarrow{BC}, \overrightarrow{CD}$ y \overrightarrow{DA} vectores que representan sus 4 lados y también orientados según las agujas del reloj.

Supongamos esos lados inicialmente también alineados según los ejes X (\overrightarrow{OX}) e Y (\overrightarrow{OY}); es decir: $\overrightarrow{AB} \parallel \overrightarrow{OY}$ y $\overrightarrow{BC} \parallel \overrightarrow{OX}$ y $\overrightarrow{CD} \parallel \overrightarrow{OY}$ y $\overrightarrow{DA} \parallel \overrightarrow{OX}$.

Podemos empezar a realizar algunas operaciones con ellos.

- **Punto medio de un rectángulo:**

El punto central de un rectángulo es el punto de corte las diagonales \overrightarrow{AC} y \overrightarrow{BD} , y coincide con el punto medio de \overrightarrow{AC} o el de \overrightarrow{BD} ; es decir:

$$\overrightarrow{OP} (\equiv P) = \overrightarrow{OA} (\equiv A) + \frac{1}{2} \overrightarrow{AC}$$

42: Punto medio de un rectángulo

Esta fórmula es también válida si el rectángulo no está en ningún plano de referencia e incluso si sus lados no están alineados con los ejes.

- **Punto contenido en un rectángulo:**

Sea P un punto cualquiera del plano XY o paralelo. P estará contenido en el rectángulo r orientado, también situado en el plano XY o paralelo y de lados alineado con los ejes XY , sí y sólo si:

$$P \in r \Leftrightarrow A_{rx} \leq P_x \text{ y } B_{rx} \geq P_x \text{ y } A_{ry} \leq P_y \text{ y } C_{ry} \geq P_y \text{ y } A_{rz} = C_{rz} = P_z$$

43: Verificar si un punto está dentro de un rectángulo orientado, paralelo al plano XY y alineado con sus ejes

- **¿El rectángulo se corta con otro?:**

Sea dos rectángulos r_1 y r_2 , ambos orientados, situados en el mismo plano XY o paralelo a éste, y alineados con los mismos ejes XY . r_1 y r_2 se cortan, si cualquiera de los vértices de r_1 está en r_2 o cualquiera de los de r_2 en r_1 ; es decir si y sólo si:

$$r_1 \cap r_2 \neq \emptyset \Leftrightarrow \begin{cases} A_{r1} \in r_2 \text{ ó} \\ B_{r1} \in r_2 \text{ ó} \\ C_{r1} \in r_2 \text{ ó} \\ D_{r1} \in r_2 \text{ ó} \end{cases} \text{ ó} \begin{cases} A_{r2} \in r_1 \text{ ó} \\ B_{r2} \in r_1 \text{ ó} \\ C_{r2} \in r_1 \text{ ó} \\ D_{r2} \in r_1 \text{ ó} \end{cases}$$

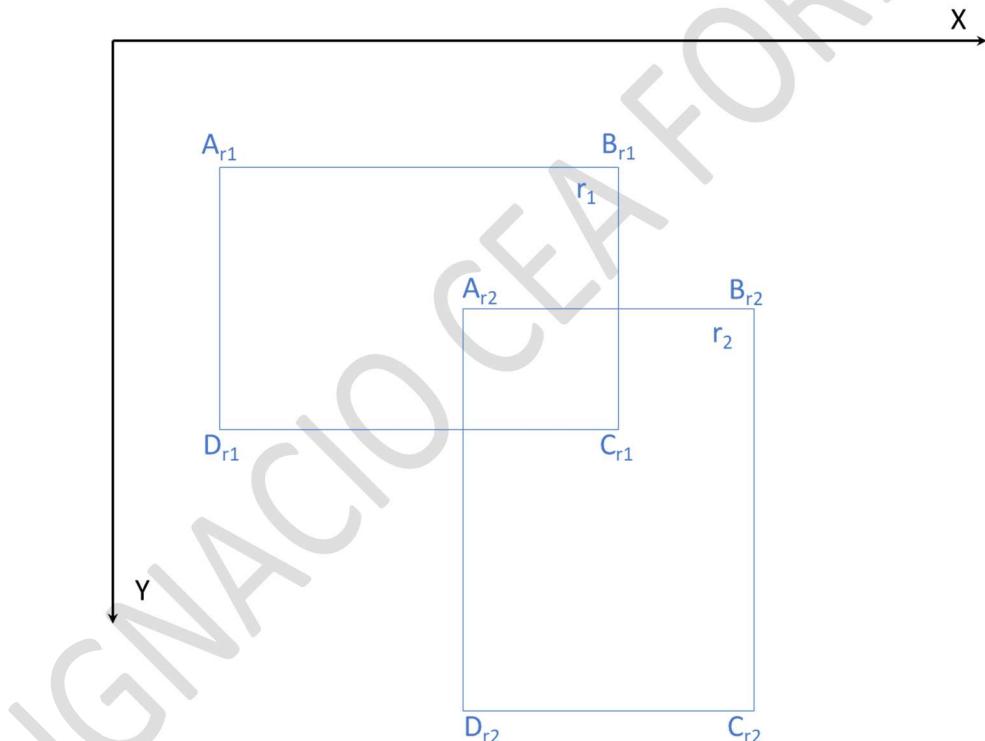
44: Verificar si dos rectángulos orientados, paralelos al plano XY y alineados con sus ejes se cortan

Algo de fácil y rápida computación, pues sólo implica comparaciones y no cálculos.
Comparaciones en las que, además, basta con que se verifique una de ellas.

Veremos, sin embargo, formas más sencillas de llevarlo a cabo después.

- **Intersección entre dos rectángulos:**

La intersección de dos rectángulos orientados y situados ambos en el mismo plano XY o paralelo, y con los lados alineados según los ejes, es otro rectángulo igualmente orientado y situado también en el mismo plano y en el que las coordenadas x, y de sus vértices son las intermedias de los rectángulos originales, y la coordenada z la de cualquiera de los puntos que intervienen (al estar todos en el plano XY o paralelo).



37: Cuadrados alineados, orientados en un plano XY y con los lados alineados con los ejes

Es decir que, para calcular el rectángulo de intersección, basta ordenar de menor a mayor las coordenadas x de los puntos A y B y lo mismo con las coordenadas y de los puntos A y C y quedarse con los valores intermedios para con ellos formar los nuevos cuatro puntos A', B', C' y D' tomando como coordenada z la de cualquier punto original.

45: Intersección de 2 rectángulos en el mismo plano XY, orientados y con los lados alineados con los ejes

Nuevamente, de fácil computación y que no involucra ningún cálculo.

Si ambos cuadrados estuvieran situados no en el plano XY, sino en el XZ o en el YZ, los razonamientos y fórmulas anteriores seguirían siendo válidas cambiando la coordenada y por z o la x por la y respectivamente.

Puede parecer muy simple, pero todo lo visto es válido para todos los juegos en 2D.

Extender la reflexión a los planos XZ e YZ permite, lo veremos más adelante, atacar los juegos 3D.

Rectángulos orientados y girados un mismo ángulo en un plano

Escalemos un poco más en el nivel de complejidad y supongamos dos rectángulos r orientados, ubicados en uno de los planos fundamentales (XY, XZ o YZ), pero no alineados con los ejes, sino girados ángulo θ respecto a un eje que pasara por su centro y fuera perpendicular al plano que los contiene.

- **Punto contenido en un rectángulo:**

Para comprobar, en este caso, si el punto P está contenido en el rectángulo r girado θ , habría varios métodos:

Uno, basado en todo lo construido anteriormente, podría consistir en girar el rectángulo r y el punto P un ángulo $-\theta$, respecto a un eje perpendicular a r común a ambos (por ejemplo, que pasara por el origen por sencillez) y comprobar entonces, con las fórmulas anteriores, si el nuevo punto P' está contenido en r' .

Antes habría que garantizar, en cualquier caso, que la coordenada z de P coincide con la coordenada z de cualquiera de los vértices del rectángulo (están de verdad en el mismo plano).

Rotar el rectángulo y el punto implica realizar bastantes cálculos, y no menores, de coma flotante.

Busquemos un camino alternativo:

La línea más corta entre los extremos del rectángulo r orientado es la que une los vértices A y C (diagonal). La más larga es la suma de la distancia entre A y B y la que une B y C (catetos de un triángulo rectángulo). Y se verifica que:

$$\|\vec{AC}\| \leq \|\vec{AB}\| + \|\vec{BC}\|$$

Sólo habrá igualdad cuando el rectángulo, en realidad, fuera un único punto.

Si tomamos cualquier punto P del interior del rectángulo, la suma de la distancia de P a los vértices A y C , cumplirá que:

$$\|\vec{AC}\| \leq \|\vec{AP}\| + \|\vec{PC}\| \leq \|\vec{AB}\| + \|\vec{BC}\|$$

46: Ecuación para verificar si un punto P pertenece a un rectángulo

Más rápido de calcular. Esta fórmula también sería válida para el anterior caso (rectángulos alineados y no girados), pero quizás sea un poco más lenta.

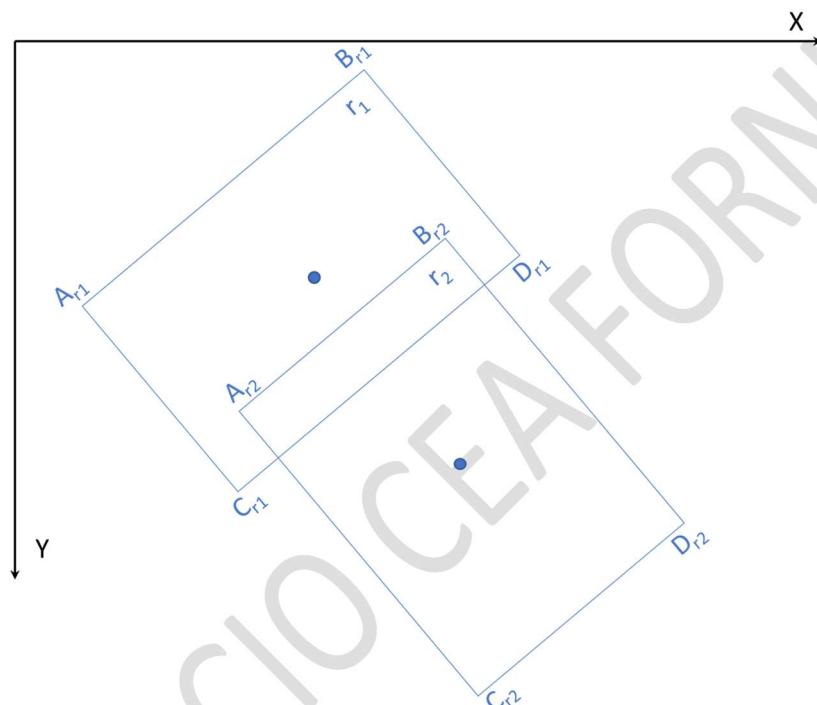
- **¿El rectángulo se corta con otro?:**

Para calcular si hay o no intersección entre 2 rectángulos r_1 y r_2 girados (pero orientados y en el mismo plano), bastaría con aplicar la comparación anterior de todos los puntos de un rectángulo en el otro y viceversa.

Con que sólo una lo cumpliera, los dos rectángulos tendrían intersección.

- **Intersección entre dos rectángulos:**

Como los ángulos de giro de ambos rectángulos son iguales ($\theta_1 = \theta_2 = \theta$), para calcular el rectángulo de intersección lo haríamos sobre los rectángulos rotados $-\theta$ para, el resultado, volverlo a rotar, utilizando el mismo eje, de nuevo un ángulo θ .



38: Cuadrados orientados en un plano XY y con los lados girados con respecto a los ejes

Rectángulos en diferentes planos

Por último, imaginemos como siguiente complejidad, que ambos rectángulos estuvieran en el mismo plano, pero este no fuera ninguno de los planos de referencia; es decir: ni el XY, ni el XZ, ni el YZ. Entonces

- **Punto contenido en un rectángulo:**

La fórmula del caso anterior seguiría siendo válida, pero habría que verificar antes que el punto efectivamente está en el mismo plano del rectángulo. Cosa que antes lo podríamos garantizar simplemente comparando las coordenadas z (o equivalente) del punto con cualquiera de los del rectángulo.

Sea \vec{n} el vector del plano en el que está el rectángulo. Entonces para que punto y rectángulo estén en el mismo plano se ha de verificar que:

$$(\overrightarrow{AB} \times \overrightarrow{AD}) \times (\overrightarrow{AP} \times \overrightarrow{AD}) = \vec{0}$$

Es decir que el que contienen a tres puntos del plano y el que contiene a dos del plano y el punto P , son paralelos. Tras esto habrá que verificar, como antes, que el punto no sólo está en el mismo plano, sino que además está dentro del rectángulo.

- **¿El rectángulo se corta con otro?:**

Igual que antes bastaría con verificar que al menos un punto de uno de ellos cumple las anteriores reglas, pero habría primero que verificar que, efectivamente ambos rectángulos están en el mismo plano, por lo que se debe verificar que:

$$(\overrightarrow{AB_1} \times \overrightarrow{AD_1}) \times (\overrightarrow{AB_2} \times \overrightarrow{AD_2}) = \vec{0}$$

Es decir, que los vectores de los planos que contienen a ambos rectángulos son el mismo o paralelos, pero además que:

$$(\overrightarrow{AB_1} \times \overrightarrow{AD_1}) \times (\overrightarrow{A_1B_2} \times \overrightarrow{A_1D_2}) = \vec{0}$$

Efectivamente están en el mismo plano y no en paralelos.

- **Intersección entre dos rectángulos:**

Al igual que sucedía anteriormente, la intersección entre ambos rectángulos sería otro rectángulo, siempre y cuando los ángulos girados por ambos fueran iguales. Para calcular dicho rectángulo bastaría, primero colocar ambos rectángulos en un plano “amigo” y en una posición “amiga”, para luego calcular el rectángulo de intersección. Con ese resultado deshacer los movimientos realizados en orden inverso. Así:

1. $\vec{n} = \overrightarrow{AB} \times \overrightarrow{AD}$
2. $\vec{v} = \overrightarrow{OZ} \times \vec{n}$
3. $\varphi = \cos^{-1} \frac{\overrightarrow{OZ} \cdot \vec{n}}{\|\overrightarrow{OZ}\|}$
4. Rotar $ABCD$ alrededor del eje \vec{v} (pasando por el origen) un ángulo $-\varphi$
5. Rotar $ABCD$ alrededor del eje \vec{n} (pasando por el centro de $ABCD$) un ángulo $-\theta$
6. Calcular la intersección = $ABCD'$
7. Deshacer sobre $ABCD'$ los movimientos realizados

Si los ángulos que están girados los rectángulos no fueran iguales y hubiera intersección, ésta no podría ser un rectángulo.

También obviamos la necesidad de calcularlo.

Las situaciones descritas son suficientes para cubrir la mayor parte de las necesidades de un juego, al menos, de 2 dimensiones.

Definición de un rectángulo

En este punto podemos extraer como grandes conclusiones sobre los rectángulos, las siguientes:

1. **Un rectángulo se puede definir en función de 4 parámetros:**

- **A:** Las coordenadas de su esquina superior izquierda
- **B:** Las coordenadas de su esquina inferior derecha

- \vec{n} : El vector normalizado (módulo 1) del plano que contiene al rectángulo
- El ángulo θ que el rectángulo está girado respecto de una alineación perfecta de sus lados a los ejes fundamentales más cercanos, alrededor del eje que pasa por su centro y tiene como dirección la del vector del plano que lo contiene.

Los otros dos puntos del rectángulo pueden calcularse a partir de los datos descritos.

Y almacenar toda la información descrita como parte de la definición del rectángulo puede ayudar a acelerar los cálculos alrededor de cualquier cuadrado.

2. En un juego se necesita hacer básicamente tres operaciones con un rectángulo:

- Determinar si un punto está o no dentro de él.
- Determinar si otro rectángulo tiene o no intersección con él.
- Determinar el polígono de intersección en caso de que esa existiera. Ese polígono sería una línea en el caso de existir y de que ambos rectángulos estuvieran en planos diferentes y no paralelos.

Parece que el número de tareas a realizar para cada una de esas operaciones, y con ello el tiempo de computación involucrado, puede variar mucho según la posición relativa de los rectángulos y de estos respecto a los ejes y planos fundamentales XY, XZ o YZ.

Por tanto, parece lógico que, en cada una de esas operaciones sigamos un árbol de decisión semejante a:

- ¿Es $\vec{n} = (1,0,0)$ o $n = (0,1,0)$ o $n = (0,0,1)$?
 - SI: ¿Es $\theta = 0$?
 - SI: Verificar si el punto pertenece al cuadrado según el primer bloque de razonamientos.
 - NO: Según el segundo bloque
 - NO: Según el tercer bloque de razonamientos

Toda esta funcionalidad se representa en *QGAMES* en la clase *QGAMES::Rectangle*:

```
/** \ingroup Game */
/** \ingroup Foundation */
/*@{*/

/**
 * @file
 * File: rectangle.hpp \n
 * Framework: Community Game Library (CGL) \n
 * Author: Ignacio Cea Forniés (Community Networks) \n
 * Creation Date: 01/12/2014 \n
 * Description: Defines a rectangle and the methods to manage it. \n
 * Versions: 1.0 Initial
 */

#ifndef __QGAMES_RECTANGLE__
#define __QGAMES_RECTANGLE__

#include <Common/position.hpp>
#include <Common/polygon.hpp>
#include <vector>

namespace QGAMES
{
    class Rectangle;
    typedef std::vector<Rectangle> Rectangles;
```

```

/** A Rectangle can exist also in the space. */
class Rectangle
{
public:
    typedef enum { _XY = 0, _XZ = 1, _YZ = 2, _OTHER = 3 } Plane;

    static const Rectangle _noRectangle;

    Rectangle ();
    Rectangle (const Position& p1, const Position& p2,
               const Vector& o = Vector (_BD 0, _BD 0, _BD 1), bdata a = _BD 0 /* Over 45
*/);
    Rectangle& operator = (const Rectangle& r);

    const Position& pos1 () const
        { return (_position1); }
    const Position& pos2 () const
        { return (_position2); }
    const Position& pos3 () const
        { return (_position3); }
    const Position& pos4 () const
        { return (_position4); }
    const Position& orientation () const
        { return (_orientation); }
    Position center () const;
    bdata width () const
        { return ((_position3 - _position1).module ()); }
    bdata height () const
        { return ((_position4 - _position1).module ()); }
    bdata diagonal () const
        { return ((_position2 - _position1).module ()); }
    Plane plane () const
        { return (_plane); }
    bdata angle () const
        { return (_angle); }

    bool hasIn (const Position& p) const;
    bool isIn (const Rectangle& r) const;
    bool intersectsWith (const Rectangle& r) const;
    bool intersectsWith (const Position& p, const Vector& v) const // with a line...
        { return (std::abs (v.dotProduct (_orientation)) < __OGAMES_ERROR
&&
        std::abs ((_position1 - p).dotProduct (_orientation)) <
__OGAMES_ERROR); }
    Rectangle intersectionWith (const Rectangle& r) const;
    std::vector <Position> intersectionWith (const Position& p, const Vector& v)
const;
    Rectangle rotate (const Position& e1, const Position& e2, bdata a) const;
    Rectangle rotateFromCenter (bdata a) const;
    Polygon projectOver (const Position& c, const Vector& o) const;
    Rectangle& toBase (); // Projects the rectangle on a plane parallel to its but
passing by the origin.
    Rectangle bigger (bdata d) const; // Returns a rectangle d unit bigger, in the
same
                                // plane, and with the same basic rotation angle than
the original one...
    Rectangle scale (bdata d) const;

    bool operator == (const Rectangle& r) const;
    bool operator != (const Rectangle& r) const
        { return (!(*this == r)); }
    Rectangle operator + (const Position& p) const
        { return (Rectangle (_position1 + p, _position2 + p,
                            _position3 + p, _position4 + p, _orientation, _angle)); }
    friend Rectangle operator + (const Position& p, const Rectangle& r)
        { return (r + p); }
    Rectangle& operator += (const Position& p)
        { *this = *this + p; return (*this); }
    Rectangle operator - (const Position& p) const
        { return (Rectangle (_position1 - p, _position2 - p,
                            _position3 - p, _position4 - p, _orientation, _angle)); }
    friend Rectangle operator - (const Position& p, const Rectangle& r)
        { return (r - p); }
    Rectangle& operator -= (const Position& p)
        { *this = *this - p; return (*this); }
    friend std::ostream& operator << (std::ostream& o, const Rectangle& r)

```

```

        { return (o << r._pos1 () << " | " << r._pos2 () << " | "
            << r._pos3 () << " | " << r._pos4 ()) <<
            "[" << r._orientation << " | " << r._angle << "]"; }

    private:
    // Implementation
    // Very internal constructor...
    Rectangle (const Position& p1, const Position& p2, const Position& p3,
               const Position& p4, const Vector& o, bdata a);

    // To simplify the calculations of points 3 & 4 in a plane...
    void calculateRestOfPointsInAPlane (bdata& p3A, bdata& p3B, bdata& p4A, bdata&
p4B,
                                         bdata p1A, bdata p1B, bdata p2A, bdata p2B, bdata a);
    // Same but ordering...
    void orderMainPointsInAPlane (bdata& p1A, bdata& p1B, bdata& p2A, bdata& p2B);

    private:
    // Usually:
    // (1)----- (3)
    //   |       |
    //   |       |
    // (4)----- (2)
    // Imaging anyway the different positions in the space the rectangle can has...
    Position _position1, _position2, _position3, _position4;
    Vector _orientation; // Could be redundant...
    // Usually too:
    // (Z)x----- (x)
    //   |
    //   |
    //   |
    //   |
    //     v (y)
    // z = x.crossproduct (y)

    // Implementation
    // To help the treatment...
    Plane _plane;
    bdata _angle;
};

#endif

// End of the file
/*@}*/

```

39: Clase Rectangle. Definición .hpp

```

#include <Common/rectangle.hpp>
#include <Common/genalgorithms.hpp>
#include <vector>
#include <algorithm>

// A no rectangle...
const QGAMES::Rectangle QGAMES::_noRectangle = QGAMES::Rectangle ();

// ---
QGAMES::Rectangle::Rectangle ()
: _position1 (), _position2 (), _position3 (), _position4 (),
  _orientation (QGAMES::Vector::zNormal),
  _plane (QGAMES::Rectangle::Plane::XY),
  _angle (_BD 0)
{
    // Nothing else to do...so far
}

// ---
QGAMES::Rectangle::Rectangle (const QGAMES::Position& p1, const QGAMES::Position& p2,
                           const QGAMES::Vector& o, QGAMES::bdata a)
: _position1 (p1), _position2 (p2),
  _position3 (QGAMES::Position::_cero), _position4 (QGAMES::Position::_cero),
  _orientation (o),
  _plane (QGAMES::Rectangle::Plane::OTHER),
  _angle (a - (_BD 2 * __PI * __BD ((int) (a / (_BD 2 * __PI))))) // To align the
angle into the first loop!

```

```

{
    // This method corrects all "defects" in parameters
    // Put the variables in the right way to allow comparations and checks later
    // Take into account little imperfection in the data.
    // Those imperfections can be the outcome of previous calculations (rotations
    mainly).
    // In the same way verifies whether the axis is right, and correct it if needed.
    // Doing so later comparations (e.g to verify whether a point is in the rectangle)
    are possible

    // Is it parallel to plane XY?
    if (abs (_position1.posZ () - _position2.posZ ()) <= QGAMES::__QGAMES_ERROR &&
        (abs (_orientation.angleWith (QGAMES::Vector::__zNormal)) <= QGAMES::__QGAMES_ERROR
    ||
        abs (_orientation.angleWith (-QGAMES::Vector::__zNormal)) <=
    QGAMES::__QGAMES_ERROR))
        // Basic 2D Rectangle...
    {
        _orientation = QGAMES::Vector::__zNormal;
        _plane = QGAMES::Rectangle::Plane::__XY;
        QGAMES::bdata p1A, p1B, p2A, p2B, pZ;
        pZ = _position1.posZ ();
        p1A = _position1.posX (); p1B = _position1.posY ();
        p2A = _position2.posX (); p2B = _position2.posY ();
        orderMainPointsInAPlane (p1A, p1B, p2A, p2B);
        // Order the points...
        // To order the point is important for further checks
        // e.g: To check the intersection between two rectangles or to check
        // whether a point is in the rectangle.
        // Ordering the points could alter the angle, but maintaining
        // is not relevant in terms of checking!.
        QGAMES::bdata p3A, p3B, p4A, p4B;
        calculateRestOfPointsInAPlane (p3A, p3B, p4A, p4B, p1A, p1B, p2A, p2B, a);
        _position1 = QGAMES::Position (p1A, p1B, pZ);
        _position2 = QGAMES::Position (p2A, p2B, pZ);
        _position3 = QGAMES::Position (p3A, p3B, pZ);
        _position4 = QGAMES::Position (p4A, p4B, pZ);
    }
    else
        // ...or it is parallel to plane XZ?
        if (abs (_position1.posY () - _position2.posY ()) <= QGAMES::__QGAMES_ERROR &&
            (abs (_orientation.angleWith (QGAMES::Vector::__yNormal)) <= QGAMES::__QGAMES_ERROR
        ||
            abs (_orientation.angleWith (-QGAMES::Vector::__yNormal)) <=
        QGAMES::__QGAMES_ERROR))
        {
            _orientation = QGAMES::Vector::__yNormal;
            _plane = QGAMES::Rectangle::Plane::__XZ;
            QGAMES::bdata p1A, p1B, p2A, p2B, pY;
            pY = _position1.posY ();
            p1A = _position1.posX (); p1B = _position1.posZ ();
            p2A = _position2.posX (); p2B = _position2.posZ ();
            orderMainPointsInAPlane (p1A, p1B, p2A, p2B);
            // Order the points (see explanation above)
            QGAMES::bdata p3A, p3B, p4A, p4B;
            calculateRestOfPointsInAPlane (p3A, p3B, p4A, p4B, p1A, p1B, p2A, p2B, a);
            _position1 = QGAMES::Position (p1A, pY, p1B);
            _position2 = QGAMES::Position (p2A, pY, p2B);
            _position3 = QGAMES::Position (p3A, pY, p3B);
            _position4 = QGAMES::Position (p4A, pY, p4B);
        }
        else
            // ...or it is parallel to plane YZ?
            if (abs (_position1.posX () - _position2.posX ()) <= QGAMES::__QGAMES_ERROR &&
                (abs (_orientation.angleWith (QGAMES::Vector::__xNormal)) <= QGAMES::__QGAMES_ERROR
            ||
                abs (_orientation.angleWith (-QGAMES::Vector::__xNormal)) <=
            QGAMES::__QGAMES_ERROR))
            {
                _orientation = QGAMES::Vector::__xNormal;
                _plane = QGAMES::Rectangle::Plane::__YZ;
                QGAMES::bdata p1A, p1B, p2A, p2B, pX;
                pX = _position1.posX ();
                p1A = _position1.posY (); p1B = _position1.posZ ();
                p2A = _position2.posY (); p2B = _position2.posZ ();
                orderMainPointsInAPlane (p1A, p1B, p2A, p2B);
                // Order the points (see explanation above)
            }
    }
}

```

```

QGAMES::bdata p3A, p3B, p4A, p4B;
calculateRestOfPointsInAPlane (p3A, p3B, p4A, p4B, p1A, p1B, p2A, p2B, a);
_position1 = QGAMES::Position (px, p1A, p1B);
_position2 = QGAMES::Position (px, p2A, p2B);
_position3 = QGAMES::Position (px, p3A, p3B);
_position4 = QGAMES::Position (px, p4A, p4B);
}
// Other...
else
{
    // Which would the angle with the z axis be?
    QGAMES::bdata agAxis = _orientation.angleWith (QGAMES::Vector::_zNormal);
    // Which would the center of the future rectangle be?
    QGAMES::Position mP = QGAMES::Position (
        (_position1 posX ()) + (_position2 posX ()) / __BD 2,
        (_position1 posY ()) + (_position2 posY ()) / __BD 2,
        (_position1 posZ ()) + (_position2 posZ ()) / __BD 2);

    // Set the points known to a position in the plane XY and aligned with the axis
    XY...
    QGAMES::Position p1R = _position1.rotate (mP, mP + _orientation, -_angle);
    p1R = p1R.rotate (mP, mP + QGAMES::Vector::_yNormal, -agAxis);
    QGAMES::Position p2R = _position2.rotate (mP, mP + _orientation, -_angle);
    p2R = p2R.rotate (mP, mP + QGAMES::Vector::_yNormal, -agAxis);

    // Rotating the position we will have 2 points in a plane parallel to the z axis
    // ...and aligned with the rest of the axis...
    // So the rest of the points can be calculated!
    QGAMES::bdata p1A, p1B, p2A, p2B, pZ;
    pZ = p1R.posZ ();
    p1A = p1R posX (); p1B = p1R posY ();
    p2A = p2R posX (); p2B = p2R posY ();
    orderMainPointsInAPlane (p1A, p1B, p2A, p2B);
    // Order the points (see explanation above)
    QGAMES::bdata p3A, p3B, p4A, p4B;
    calculateRestOfPointsInAPlane (p3A, p3B, p4A, p4B, p1A, p1B, p2A, p2B, a);
    _position1 = QGAMES::Position (p1A, p1B, pZ).
        rotate (mP, mP + QGAMES::Vector::_yNormal, agAxis).
        rotate (mP, mP + _orientation, _angle);
    _position2 = QGAMES::Position (p2A, p2B, pZ).
        rotate (mP, mP + QGAMES::Vector::_yNormal, agAxis).
        rotate (mP, mP + _orientation, _angle);
    _position3 = QGAMES::Position (p3A, p3B, pZ).
        rotate (mP, mP + QGAMES::Vector::_yNormal, agAxis).
        rotate (mP, mP + _orientation, _angle);
    _position4 = QGAMES::Position (p4A, p4B, pZ).
        rotate (mP, mP + QGAMES::Vector::_yNormal, agAxis).
        rotate (mP, mP + _orientation, _angle);
    }
}

// ---
QGAMES::Rectangle& QGAMES::Rectangle::operator = (const QGAMES::Rectangle& r)
{
    _position1 = r._position1;
    _position2 = r._position2;
    _position3 = r._position3;
    _position4 = r._position4;
    _orientation = r._orientation;
    _plane = r._plane;
    _angle = r._angle;
    return (*this);
}

// ---
QGAMES::Position QGAMES::Rectangle::center () const
{
    return ((_position2 + _position1) / 2);
}

// ---
bool QGAMES::Rectangle::hasIn (const QGAMES::Position& p) const
{
    bool result = false;
    // Parallel to plane XY?
    if (_plane == QGAMES::Rectangle::Plane::_XY)
    {

```

```

        if (abs (_angle) <= QGAMES::__QGAMES_ERROR) // The rectangle is not twisted
    {
        result = abs (p.posZ () - _position1.posZ ()) <= QGAMES::__QGAMES_ERROR && //
at the same level (necessary condition)
            p posX () >= (_position1 posX () - QGAMES::__QGAMES_ERROR) &&
            p posX () <= (_position2 posX () + QGAMES::__QGAMES_ERROR) &&
            p posY () >= (_position1 posY () - QGAMES::__QGAMES_ERROR) &&
            p posY () <= (_position2 posY () + QGAMES::__QGAMES_ERROR);
    }
else
{
    // Rotate the rectangle to be aligned..
    // ...and then makes the same test!
    QGAMES::Position c = center ();
    QGAMES::Position cF = c + QGAMES::Vector::__zNormal;
    result = rotate (c, cF, -_angle).
        hasIn (p.rotate (c, cF, -_angle)); // If true, then true!
}
}
else
// Parallel to plane XZ?
if (_plane == QGAMES::Rectangle::Plane::__XZ)
{
    if (abs (_angle) <= QGAMES::__QGAMES_ERROR) // The rectangle is not twisted...
    {
        result = abs (p.posY () - _position1.posY ()) <= QGAMES::__QGAMES_ERROR && //
at the same level (necesary condition)
            p posX () >= (_position1 posX () - QGAMES::__QGAMES_ERROR) &&
            p posX () <= (_position2 posX () + QGAMES::__QGAMES_ERROR) &&
            p posZ () >= (_position1 posZ () - QGAMES::__QGAMES_ERROR) &&
            p posZ () <= (_position2 posZ () + QGAMES::__QGAMES_ERROR);
    }
else
{
    // Rotate the rectangle to be aligned..
    // ...and then makes the same test!
    QGAMES::Position c = center ();
    QGAMES::Position cF = c + QGAMES::Vector::__yNormal;
    result = rotate (c, cF, -_angle).
        hasIn (p.rotate (c, cF, -_angle)); // If true, then true!
}
}
else
if (_plane == QGAMES::Rectangle::Plane::__YZ)
{
    if (abs (_angle) <= QGAMES::__QGAMES_ERROR) // The rectangle is not twisted
    {
        result = abs (p posX () - _position1 posX ()) <= QGAMES::__QGAMES_ERROR && //
at the same level (necessary condition)
            p posY () >= (_position1 posY () - QGAMES::__QGAMES_ERROR) &&
            p posY () <= (_position2 posY () + QGAMES::__QGAMES_ERROR) &&
            p posZ () >= (_position1 posZ () - QGAMES::__QGAMES_ERROR) &&
            p posZ () <= (_position2 posZ () + QGAMES::__QGAMES_ERROR);
    }
else
{
    // Rotate the rectangle to be aligned..
    // ...and then makes the same test!
    QGAMES::Position c = center ();
    QGAMES::Position cF = c + QGAMES::Vector::__xNormal;
    result = rotate (c, cF, -_angle).
        hasIn (p.rotate (c, cF, -_angle)); // If true, then true!
}
}
// Other?
else
{
    // Angle with the zaxis?
    QGAMES::bdata agAxis = _orientation.angleWith (QGAMES::Vector::__zNormal);
    // Wich is the center?
    QGAMES::Position mP = center ();
    // Creates a rectangle perpendicular to the z axis,
    // ...and aligned to the x and y axis!
    QGAMES::Position p1R = _position1.rotate (mP, mP + QGAMES::Vector::__yNormal, -
agAxis);
    QGAMES::Position p2R = _position2.rotate (mP, mP + QGAMES::Vector::__yNormal, -
agAxis);
}

```

```

    QGAMES::Rectangle rR (p1R, p2R, QGAMES::Vector::zNormal, _angle);
    // Same with the point to check
    QGAMES::Position pR = p.rotate (mP, mP + QGAMES::Vector::yNormal, -agAxis);
    pR.rotate (mP, mP + QGAMES::Vector::zNormal, -_angle);
    result = rR.hasIn (pR);
}

return (result);
}

// ---
bool QGAMES::Rectangle::isIn (const QGAMES::Rectangle& r) const
{
    // Both rectangles have to be in the same plane...
    // It means that they normal vectors are parallel
    if (_orientation.crossProduct (r._orientation).module2 () > QGAMES::__QGAMES_ERROR)
        return (false); // ...or almost! (module2 always positive)

    // Once we have test they are in the same plane
    // One is other if all their points are in...
    return (r.hasIn (pos1 ()) && r.hasIn (pos2 ()) &&
            r.hasIn (pos3 ()) && r.hasIn (pos4 ()));
}

// ---
bool QGAMES::Rectangle::intersectsWith (const Rectangle& r) const
{
    bool result = false;

    // If they are in the same plane, and that plane is "simple", let's say parallel
    // to one of the basic planes, the formula is too easy!
    // The same formula could be used even when the planes are not parallel
    // to one of the coordinates, because the hasIn method takes into account this
    // situation, but it is better to do it in another way! (read later why)
    if (_plane == r._plane && _plane != QGAMES::Rectangle::Plane::__OTHER)
    {
        // The intersection is only possible when the rectangles are in the same "level",
        // Otherwise they wouldn't cross ever!
        if (_plane == QGAMES::Rectangle::Plane::XY &&
            abs (_position1.posZ () - r._position1.posZ ()) <= QGAMES::__QGAMES_ERROR)
        ||
        (_plane == QGAMES::Rectangle::Plane::XZ &&
            abs (_position1.posY () - r._position1.posY ()) <= QGAMES::__QGAMES_ERROR)
        ||
        (_plane == QGAMES::Rectangle::Plane::YZ &&
            abs (_position1 posX () - r._position1 posX ()) <= QGAMES::__QGAMES_ERROR))
            result = hasIn (r._position1) || hasIn (r._position2) ||
                    hasIn (r._position3) || hasIn (r._position4) ||
                    r.hasIn (_position1) || r.hasIn (_position2) ||
                    r.hasIn (_position3) || r.hasIn (_position4) ||
                    hasIn (r.center ()) || r.hasIn (center ());
    }
    else
    {
        if (_orientation.crossProduct (r._orientation).module2 () <=
QGAMES::__QGAMES_ERROR)
        {
            // It could be done as:
            // result = hasIn (r._position1)...
            // because the hasIn method takes into account whether the
            // rectangle is rotated
            // But doing so, the rotation of the rectangle would be done
            // many times. This way is then faster!

            // Which would the angle with the z axis be?
            QGAMES::bdata agAxis = _orientation.angleWith (QGAMES::Vector::zNormal);
            // Set the points known to a position in the plane XY and aligned with the axis
            XY...
            QGAMES::Rectangle r1R = rotate (QGAMES::Position::_cero, _orientation, -
            _angle);
            rotate (QGAMES::Position::_cero, QGAMES::Vector::yNormal, -agAxis);
            QGAMES::Rectangle r2R = r.rotate (QGAMES::Position::_cero, _orientation, -
            _angle);
            rotate (QGAMES::Position::_cero, QGAMES::Vector::yNormal, -agAxis);
            result = r1R.intersectsWith (r2R);
        }
    }
}

```

```

{
    // Calculates the vector of the intersection line between
    // the two planes containing the rectangles.
    // It doesn't mean that the rectangles intersect yet!
    int tp = -1;
    QGAMES::Vector rI = _orientation.crossProduct (r._orientation); // The
orientation of the line
    QGAMES::bdata A = __BD_0; QGAMES::bdata B = __BD_0; QGAMES::bdata C = __BD_0;
    QGAMES::bdata Ap = __BD_0; QGAMES::bdata Bp = __BD_0; QGAMES::bdata Cp = __BD
0;
    QGAMES::bdata l = _orientation.dotProduct (_position1); // The indepent term of
the first plane equation
    QGAMES::bdata lp = r._orientation.dotProduct (r._position1); // The independent
term of the second!
    if (abs (_orientation.posY ()) > QGAMES::__QGAMES_ERROR &&
        abs (r._orientation.posY ()) > QGAMES::__QGAMES_ERROR &&
        abs ((_orientation.posX () * r._orientation.posY ()) +
             (_orientation.posY () * r._orientation.posX ()) ) >=
QGAMES::__QGAMES_ERROR)
    {
        tp = 0;
        A = _orientation.posY (); Ap = r._orientation.posY ();
        B = _orientation.posX (); Bp = r._orientation.posX ();
    }
    else
    if (abs (_orientation.posY ()) > QGAMES::__QGAMES_ERROR &&
        abs (r._orientation.posY ()) > QGAMES::__QGAMES_ERROR &&
        abs ((_orientation.posZ () * r._orientation.posY ()) +
             (_orientation.posY () * r._orientation.posZ ()) ) >=
QGAMES::__QGAMES_ERROR)
    {
        tp = 1;
        A = _orientation.posY (); Ap = r._orientation.posY ();
        B = _orientation.posZ (); Bp = r._orientation.posZ ();
    }
    else
    if (abs (_orientation.posX ()) > QGAMES::__QGAMES_ERROR &&
        abs (r._orientation.posX ()) > QGAMES::__QGAMES_ERROR &&
        abs ((_orientation.posZ () * r._orientation.posX ()) +
             (_orientation.posX () * r._orientation.posZ ()) ) >=
QGAMES::__QGAMES_ERROR)
    {
        tp = 2;
        A = _orientation.posX (); Ap = r._orientation.posX ();
        B = _orientation.posZ (); Bp = r._orientation.posZ ();
    }

    QGAMES::bdata a = (l - (lp * A / Ap)) / (B - (Bp * A / Ap));
    QGAMES::bdata b = (l - (a * B)) / A;
    QGAMES::Position pI = QGAMES::Position::_cero;
    if (tp == 0) pI = QGAMES::Position (a, b, __BD_0);
    else if (tp == 1) pI = QGAMES::Position (__BD_0, a, b);
    else if (tp == 2) pI = QGAMES::Position (a, __BD_0, b);

    // Now I have already definede the vector defining the intersection line (rI)
    // ...and one point the line pass over (pI)
    // The it is time to calculate another points of it
    QGAMES::Position pIp = rI + pI;

    // They would intersect if one point of the line belonging to both rectangles
can be found!
    // If the distance between one corner of any rectangle and the intersection
line calculated before
    // is less than the side of the right rectangle, it would mean that
    // the intersection line is in the rectangle
    QGAMES::bdata d1 = (_position1 - pI).crossProduct (_position1 - pIp)).module2
() /
    (pIp - pI).module2 (); // http://mathworld.wolfram.com/Point-LineDistance3-
Dimensional.html
    QGAMES::bdata d2 = ((r._position1 - pI).crossProduct (r._position1 -
pIp)).module2 () /
        (pIp - pI).module2 ();
    result = d1 <= (_position4 - _position1).module2 () &&
            d2 <= (r._position4 - r._position1).module2 ();

    // Probably they will intersect, but not in a rectangle...a line maximum!
}

```

```

        }

        return (result);
    }

// ---
QGAMES::Rectangle QGAMES::Rectangle::intersectionWith (const QGAMES::Rectangle& r) const
{
    // If there is no intersection, then result is a no rectangle...
    if (!intersectsWith (r))
        return (QGAMES::Rectangle::_noRectangle);

    // The intersection is, in general a polygon...
    // To be a rectangle they have to be in the same plane
    // and they have to be rotated equally...
    if (_plane != r._plane ||
        _orientation.crossProduct (r._orientation).module2 () > QGAMES::__QGAMES_ERROR)
        return (QGAMES::Rectangle::_noRectangle);
    if (_angle != r._angle)
        return (QGAMES::Rectangle::_noRectangle);
    // If the intersection is going to be not a rectangle
    // this is not the right method to process it,
    // so a no rectangle is returned!

    // Then it is time to calculate the result rectangle
    // First of all in easy way, when both rectangles are parallel to any axis!
    QGAMES::Rectangle result = QGAMES::Rectangle::_noRectangle;
    if (_plane == QGAMES::Rectangle::Plane::_XY)
    {
        // First, the orientation has to be the same...
        if (abs (_angle) <= QGAMES::__QGAMES_ERROR)
        {
            std::vector <bdata> pA; pA.resize (4);
            std::vector <bdata> pB; pB.resize (4);
            pA [0] = _position1.posX (); pA [1] = _position2.posX ();
            pA [2] = r._position1.posX (); pA [3] = r._position2.posX ();
            pB [0] = _position1.posY (); pB [1] = _position2.posY ();
            pB [2] = r._position1.posY (); pB [3] = r._position2.posY ();
            std::sort (pA.begin (), pA.end()); // Short the point from less to more...
            std::sort (pB.begin (), pB.end()); // Same...
            result = QGAMES::Rectangle (QGAMES::Position (pA [1], pB [1], _position1.posZ
                ()),
                QGAMES::Position (pA [2], pB [2], _position1.posZ ()),
                QGAMES::Vector::_zNormal);
        }
        // Otherwise, it corrects the orientation and recalculates...
        else
        {
            QGAMES::Position c = center ();
            QGAMES::Position cF = c + QGAMES::Vector::_zNormal;
            QGAMES::Rectangle rR =
                rotate (c, cF, -_angle).intersectionWith (r.rotate (c, cF, -_angle));
            // Everything has to be rotated around the same center (it could be any, but
            the same)
            return (rR.rotate (c, cF, _angle));
            // Put it back in the original position!
        }
    }
    else
    if (_plane == QGAMES::Rectangle::Plane::_XZ)
    {
        // First, the orientation has to be the same...
        if (abs (_angle) <= QGAMES::__QGAMES_ERROR)
        {
            std::vector <bdata> pA; pA.resize (4);
            std::vector <bdata> pB; pB.resize (4);
            pA [0] = _position1.posX (); pA [1] = _position2.posX ();
            pA [2] = r._position1.posX (); pA [3] = r._position2.posX ();
            pB [0] = _position1.posZ (); pB [1] = _position2.posZ ();
            pB [2] = r._position1.posZ (); pB [3] = r._position2.posZ ();
            std::sort (pA.begin (), pA.end()); // Short the point from less to more...
            std::sort (pB.begin (), pB.end()); // Same...
            result = QGAMES::Rectangle (QGAMES::Position (pA [1], _position1.posY (), pB
                [1]),
                QGAMES::Position (pA [2], _position1.posY (), pB [2]),
                QGAMES::Vector::_yNormal);
        }
    }
}

```

```

    // Otherwise, it corrects the orientation and recalculates...
else
{
    QGAMES::Position c = center ();
    QGAMES::Position cF = c + QGAMES::Vector::_yNormal;
    QGAMES::Rectangle rR =
        rotate (c, cF, -_angle).intersectionWith (r.rotate (c, cF, -_angle));
    // Everything has to be rotated around the same center (it could be any, but
the same)
    return (rR.rotate (c, cF, _angle));
    // Put it back in the original position!!
}
}
else
if (_plane == QGAMES::Rectangle::Plane::_YZ)
{
    // First, the orientation has to be the same...
    if (abs (_angle) <= QGAMES::_QGAMES_ERROR)
    {
        std::vector <bdata> pA; pA.resize (4);
        std::vector <bdata> pB; pB.resize (4);
        pA [0] = _position1.posY (); pA [1] = _position2.posY ();
        pA [2] = r._position1.posY (); pA [3] = r._position2.posY ();
        pB [0] = _position1.posZ (); pB [1] = _position2.posZ ();
        pB [2] = r._position1.posZ (); pB [3] = r._position2.posZ ();
        std::sort (pA.begin (), pA.end()); // Short the point from less to more...
        std::sort (pB.begin (), pB.end()); // Same...
        result = QGAMES::Rectangle (QGAMES::Position (_position1 posX (), pA [1], pB
[1]),
            QGAMES::Position (_position1 posX (), pA [2], pB [2]),
            QGAMES::Vector::_xNormal);
    }
    // Otherwise, it corrects the orientation and recalculates...
else
{
    QGAMES::Position c = center ();
    QGAMES::Position cF = c + QGAMES::Vector::_xNormal;
    QGAMES::Rectangle rR =
        rotate (c, cF, -_angle).intersectionWith (r.rotate (c, cF, -_angle));
    // Everything has to be rotated around the same center (it could be any, but
the same)
    return (rR.rotate (c, cF, _angle));
    // Put it back in the original position!!
}
}
// At this point the rectangles are not paralell to any axis,
// but trhey are in teh same axis, so there is a complex way to calculate
// the rectangle interscetion!
else
{
    QGAMES::bdata agAxis = _orientation.angleWith (QGAMES::Vector::_zNormal);
    // Set the points known to a position in the plane XY and aligned with the axis
XY...
    QGAMES::Rectangle r1R = rotate (QGAMES::Position::_cero, _orientation, -_angle).
        rotate (QGAMES::Position::_cero, QGAMES::Vector::_yNormal, -agAxis);
    QGAMES::Rectangle r2R = r.rotate (QGAMES::Position::_cero, _orientation, -_angle).
        rotate (QGAMES::Position::_cero, QGAMES::Vector::_yNormal, -agAxis);
    result = r1R.intersectionWith (r2R);
    result = result.rotate (QGAMES::Position::_cero, QGAMES::Vector::_yNormal,
agAxis),
        rotate (QGAMES::Position::_cero, _orientation, _angle); // Put it back...
}
return (result);
}

// ---
std::vector <QGAMES::Position> QGAMES::Rectangle::intersectionWith (const
QGAMES::Position& p, const QGAMES::Vector& v) const
{
    std::vector <QGAMES::Position> result { };

    // If there is no intersection, then result are no points
    if (!intersectsWith (p, v))
        return (std::vector <QGAMES::Position> { /* Empty */ });

    // If the plane is one of the main three main...

```

```

QGAMES::Position p0, p1;
p0 = p1 = QGAMES::Position::_noPoint;
if (_plane == QGAMES::Rectangle::Plane::_OTHER)
{
    // Convert everything it into an XY rectangle rotating it...
    QGAMES::Rectangle cR (*this);
    QGAMES::Position cP (p);
    QGAMES::Vector cV (v);

    QGAMES::Position cRect = center ();
    QGAMES::bdata aRect = _orientation.angleWith (QGAMES::Vector::_xNormal);
    cR = cR.rotate (cRect, cRect + QGAMES::Vector::_xNormal, -aRect);
        rotate (cRect, cRect + QGAMES::Vector::_zNormal, -_angle);
    cP = cP.rotate (cRect, cRect + QGAMES::Vector::_xNormal, -aRect);
        rotate (cRect, cRect + QGAMES::Vector::_zNormal, -_angle);
    cV = cV.rotate (cRect, cRect + QGAMES::Vector::_xNormal, -_angle).
        rotate (cRect, cRect + QGAMES::Vector::_zNormal, -_angle);
    std::vector <QGAMES::Position> iR = cR.intersectionWith (cP, cV);
    for (auto i : iR) /* Pure C++11 */
        result.push_back
            (i.rotate (cRect, cRect + QGAMES::Vector::_zNormal, _angle).
             rotate (cRect, cRect + QGAMES::Vector::_xNormal, aRect));
}
else
{
    QGAMES::bdata p0A, p0B, p1A, p1B;
    p0A = p0B = p1A = p1B = __BD 0;

    QGAMES::bdata vA, vB;
    vA = vB = __BD 0;
    QGAMES::bdata MA, NA, MB, NB;
    MA = MB = NA = NB = __BD 0;
    QGAMES::bdata pA, pB;
    pA = pB = __BD 0;
    if (_plane == QGAMES::Rectangle::Plane::_XY)
    {
        vA = v posX (); vB = v posY ();
        MA = _position1 posX (); MB = _position2 posX ();
        NA = _position1 posY (); NB = _position2 posY ();
        pA = p posX (); pB = p posY ();
    }
    else
    if (_plane == QGAMES::Rectangle::Plane::_XZ)
    {
        vA = v posX (); vB = v posZ ();
        MA = _position1 posX (); MB = _position2 posX ();
        NA = _position1 posZ (); NB = _position2 posZ ();
        pA = p posX (); pB = p posZ ();
    }
    else // Plane YZ
    {
        vA = v posY (); vB = v posZ ();
        MA = _position1 posY (); MB = _position2 posY ();
        NA = _position1 posZ (); NB = _position2 posZ ();
        pA = p posY (); pB = p posZ ();
    }

    if (vA != __BD 0.0 && vB != __BD 0.0)
    {
        QGAMES::bdata d = (-vB / vA * pA) + pB;

        p0A = MA;
        p0B = (vB / vA * MA) + d;
        if (p0B > NB)
        {
            p0A = vA / vB * (NB - d);
            p0B = NB;
        }
        else
        if (p0B < NA)
        {
            p0A = vA / vB * (NA - d);
            p0B = NA;
        }
    }

    p1A = MB;
    p1B = (vB / vA * MB) + d;
}

```

```

        if (p1B > NB)
        {
            p1A = vA / vB * (NB - d);
            p1B = NB;
        }
        else
        if (p1B < NA)
        {
            p1A = vA / vB * (NA - d);
            p1B = NA;
        }
    }
    else
    if (vA == __BD 0.0)
    {
        p0A = pA;
        p0B = NA;
        p1A = pA;
        p1B = NB;
    }
    else // vB == __BD 0.0
    {
        p0A = MA;
        p0B = pB;
        p1A = MB;
        p1B = pB;
    }

    if (p0A >= MA && p0A <= MB && p1A >= MA && p1A <= MB &&
        p0B >= NA && p0B <= NB && p1B >= NA && p1B <= NB)
    {
        // Fits the rectangle...
        if (_plane == QGAMES::Rectangle::Plane::_XY)
            { p0 = QGAMES::Position (p0A, p0B, __BD 0); p1 = QGAMES::Position (p1A, p1B,
__BD 0); }
        else if (_plane == QGAMES::Rectangle::Plane::_XZ)
            { p0 = QGAMES::Position (p0A, __BD 0, p0B); p1 = QGAMES::Position (p1A, __BD
0, p1B); }
        else // Plane YZ
            { p0 = QGAMES::Position (__BD 0, p0A, p0B); p1 = QGAMES::Position (__BD 0,
p1A, p1B); }

        // So, put them into the result...
        result.push_back (p0); result.push_back (p1);
    }
    // Otherwise the intersection points are out of the rectangle...
}

return (result);
}

// ---
QGAMES::Rectangle QGAMES::Rectangle::rotate (const QGAMES::Position& e1,
    const QGAMES::Position& e2, bdata a) const
{
    // If no angle... no rotation...
    if (a == __BD 0)
        return (*this);

    // If the rotation axis passes through the center of the rectangle
    // and it is parallel to the orientation,
    // We have just only to che the angle!
    // First condition: rotation axis and orientation has to be parallel...so:
    bool rP = ((e2 - e1).crossProduct (_orientation).module2 () <=
QGAMES::__QGAMES_ERROR);
    // Second condition: The vector from the center to one point of the axis
    // has to be parallel to the axis itself:
    bool rC = ((e2 - e1).crossProduct (e2 - center ()).module2 () <=
QGAMES::__QGAMES_ERROR);

    // If the rotation axis is parallel to the orientation of the rectangle
    // the angle of the rectangle respect the plane it is on, has to be actualized
    // and it won't be needed to change the orientation itself.
    // In any case, just two points are rotated, and the rest are calculated
    return (Rectangle (
        _position1.rotate (e1, e2, a),

```

```

        _position2.rotate (e1, e2, a),
        rP ? _orientation : _orientation.rotate (e1, e2, a),
        rP ? _angle + a : _angle));
    }

// ---
QGAMES::Rectangle QGAMES::Rectangle::rotateFromCenter (QGAMES::bdata a) const
{
    QGAMES::Position c = center ();
    return (rotate (c, c + _orientation, a));
}

// ---
QGAMES::Polygon QGAMES::Rectangle::projectOver (const Position& c, const Vector& o)
const
{
    // Independent of the type of rectangle...
    QGAMES::Positions pts;
    pts.push_back (_position1);
    pts.push_back (_position3); // In order...
    pts.push_back (_position2);
    pts.push_back (_position4);
    return (QGAMES::Polygon (pts).projectOver (c, o));
}

// ---
QGAMES::Rectangle& QGAMES::Rectangle::toBase ()
{
    QGAMES::Polygon ply = projectOver (QGAMES::Position::_zero, _orientation);
    _position1 = ply.point (0); _position3 = ply.point (1); _position2 = ply.point (2);
    _position4 = ply.point (3);

    return (*this);
}

// ---
QGAMES::Rectangle QGAMES::Rectangle::bigger (QGAMES::bdata d) const
{
    QGAMES::Vector vH = ((_position3 - _position1).normalize ()) * d; // Vector from 1 to
2...
    QGAMES::Vector vV = ((_position4 - _position1).normalize ()) * d; // Vector from 1 to
3...
    return (QGAMES::Rectangle (_position1 - vH - vV, _position2 + vH + vV,
        _position3 + vH - vV, _position4 - vH + vV, _orientation, _angle));
}

// ---
QGAMES::Rectangle QGAMES::Rectangle::scale (QGAMES::bdata d) const
{
    QGAMES::bdata dL = (_position2 - _position1).module () * (_BD 1 - d) / _BD 2; // What to reduce the diagonal from a corner...
    QGAMES::Vector dG = (_position2 - _position1).normalize ();
    return (QGAMES::Rectangle (_position1 + (dG * dL), _position2 - (dG * dL),
        _orientation, _angle));
}

// ---
bool QGAMES::Rectangle::operator == (const QGAMES::Rectangle& r) const
{
    return (_position1 == r._position1 &&
        _position2 == r._position2 &&
        _position3 == r._position3 &&
        _position4 == r._position4 &&
        _orientation == r._orientation &&
        abs (_angle - r._angle) <= QGAMES::__QGAMES_ERROR);
}

// ---
QGAMES::Rectangle::Rectangle (const QGAMES::Position& p1, const QGAMES::Position& p2,
    const QGAMES::Position& p3, const QGAMES::Position& p4, const QGAMES::Vector& o,
    QGAMES::bdata a)
    : _position1 (p1), _position2 (p2), _position3 (p3), _position4 (p4),
    _orientation (o), _angle (a)
{
    // This constructor has to be used really carefully
    // The constructor takes the points as something good
    // No checks are done
}

```

```

// If the points are not in the right order,
// further invocations to other methods could fail!

// The plane is the only variable pending to be fixed...
QGAMES::Vector oN = _orientation.normalize ();
if (oN == QGAMES::Vector::_zNormal)
    _plane = QGAMES::Rectangle::Plane::_XY;
else
if (oN == QGAMES::Vector::_yNormal)
    _plane = QGAMES::Rectangle::Plane::_XZ;
else
if (oN == QGAMES::Vector::_xNormal)
    _plane = QGAMES::Rectangle::Plane::_YZ;
else
    _plane = QGAMES::Rectangle::Plane::_OTHER;
}

// ---
void QGAMES::Rectangle::calculateRestOfPointsInAPlane (QGAMES::bdata& p3A,
QGAMES::bdata& p3B,
QGAMES::bdata& p4A, QGAMES::bdata& p4B,
QGAMES::bdata p1A, QGAMES::bdata p1B, QGAMES::bdata p2A, QGAMES::bdata p2B,
QGAMES::bdata a)
{
    if (a == __BD 0)
    {
        p3A = p2A; p3B = p1B;
        p4A = p1A; p4B = p2B;
    }
    else
    {
        // It is going to simulate the point in the plane XY...
        // Because the point is already in a plane, it is really the same
        // The plane we play with!
        QGAMES::Position P1 (p1A, p1B, __BD 0); // Copy of the point 1 in plane XY
        QGAMES::Position P2 (p2A, p2B, __BD 0); // Copy of the point 2 in plane XY
        QGAMES::Position PC ((p1A + p2A) / __BD 2, (p1B + p2B) / __BD 2, __BD 0); // The
center of the diagonal joining both...
        QGAMES::Position P1R = P1.rotate (PC, PC + QGAMES::Vector::_zNormal, -a); // Rotate the P1 around the center to "align" the diagonal
        QGAMES::Position P2R = P2.rotate (PC, PC + QGAMES::Vector::_zNormal, -a); // The same with P2...
        QGAMES::Position P3 (P2R posX (), P1R posY (), __BD 0); // Now the P3 can be calculated as above
        QGAMES::Position P4 (P1R posX (), P2R posY (), __BD 0); // Same with P4
        QGAMES::Position P3R = P3.rotate (PC, PC + QGAMES::Vector::_zNormal, a); // Rotate back P3
        QGAMES::Position P4R = P4.rotate (PC, PC + QGAMES::Vector::_zNormal, a); // ...and P4
        p3A = P3R posX (); p3B = P3R posY (); // The result coordinates are now the underrotated points...
        p4A = P4R posX (); p4B = P4R posY ();
    }
}

// ---
void QGAMES::Rectangle::orderMainPointsInAPlane (QGAMES::bdata& p1A, QGAMES::bdata& p1B,
QGAMES::bdata& p2A, QGAMES::bdata& p2B)
{
    std::vector <bdata> pA; pA.resize (2);
    std::vector <bdata> pB; pB.resize (2);
    pA [0] = p1A; pA [1] = p2A;
    pB [0] = p1B; pB [1] = p2B;
    std::sort (pA.begin (), pA.end()); // Short the point from less to more...
    std::sort (pB.begin (), pB.end()); // Same...
    p1A = pA [0]; p2A = pA [1];
    p1B = pB [0]; p2B = pB [1];
}

```

40: Clase Rectangle. Definición src

Paralelepípedos

En un juego 3D los paralelepípedos juegan un papel semejante al que juegan los rectángulos en el mundo 2D. Suelen emplearse como figuras geométricas que contienen a las entidades para detectar colisiones entre ellas y con otros elementos del juego.

Para definir un paralelepípedo basta con definir, por ejemplo, el rectángulo de su base y su altura.

Todo esto se refleja en la clase *QGAMES::Box* de *QGAMES*, así:

```
/** \ingroup Game */
/** \ingroup Foundation */
/*@{*/

<**
 * @file
 * File: box.hpp \n
 * Framework: Commy Game Library (CGL) \n
 * Author: Ignacio Cea Forniés (Community Networks) \n
 * Creation Date: 18/05/2019 \n
 * Description: Defines a box and the methods to manage it. \n
 * Versions: 1.0 Initial
 */

#ifndef __QGAMES_BOX__
#define __QGAMES_BOX__

#include <Common/position.hpp>
#include <Common/rectangle.hpp>
#include <vector>

namespace QGAMES
{
    /** Class to represent a box in the space. \n
     * The box is made up of six rectangles parallel two to two. */
    class Box
    {
        public:
            static Box _noBox;

            Box ()
                : _origin (Position::_noPoint),
                  _vX (Vector::_noPoint), _vY (Vector::_noPoint), _vZ (Vector::_noPoint),
                  _wX (_BD 0), _wY (_BD 0), _wZ (_BD 0),
                  _rBase (Rectangle::_noRectangle), _rTop (Rectangle::_noRectangle),
                  _rFront (Rectangle::_noRectangle), _rBack (Rectangle::_noRectangle),
                  _rRight (Rectangle::_noRectangle), _rLeft (Rectangle::_noRectangle)
                { }

        /**
         * The data representing the box are:
         * x,y,z doesn't mean necessary those axis. It is only a way to simplify the
         explanation.
         * @param o  The back left point of the cube in the space.
         * @param vX The direction of one axis. It'll be normalized.
         * @param vY The direction of the other axis. It'll be normalized later.
         * @param wx Length in one axis.
         * @param wy Length in the second axis.
         * @param wz Length in the third axis.
         */
        Box (const Position& o, const Vector& vx, const Vector& vy, bdata wx, bdata wy,
              bdata wz)
            : _origin (o),
              _vX (vx), _vY (vy), _vZ (vx.crossProduct (vy)),
              _wX (wx), _wY (wy), _wZ (wz)
            // The internal points are calculated in the method calculateInternalPoints
            { _vX.normalize (); _vY.normalize (); _vZ.normalize ();
              calculateInternalPoints (); }

        Box (const Rectangle& bottom, bdata wz);
    };
}
```

```

const Position& origin () const
    { return (_origin); }
const Vector& vX () const
    { return (_vX); }
const Vector& vY () const
    { return (_vY); }
const Vector& vZ () const
    { return (_vZ); }
const bdata width () const
    { return (_wX); }
const bdata height () const
    { return (_wY); }
const bdata depth () const
    { return (_wZ); }
Position center () const
    { return ((_rTop.pos1 () + _rBase.pos2 ()) / __BD 2); }

const Rectangle& topRectangle () const
    { return (_rTop); }
const Rectangle& bottomRectangle () const
    { return (_rBase); }
const Rectangle& frontRectangle () const
    { return (_rFront); }
const Rectangle betweenFrontAndBackRectangle () const
    { return (Rectangle (_rFront.pos1 () + _rBack.pos1 ()) / __BD 2,
                    (_rFront.pos2 () + _rBack.pos2 ()) / __BD 2)); }
const Rectangle& backRectangle () const
    { return (_rBack); }
const Rectangle& rightRectangle () const
    { return (_rRight); }
const Rectangle& leftRectangle () const
    { return (_rLeft); }

Box rotate (const Position& e1, const Position& e2, bdata a) const
    { return (Box (_origin.rotate (e1, e2, a), _vX.rotate (e1, e2, a),
_vY.rotate (e1, e2, a),
_wX, _wY, _wZ)); }
Box rorateFromCenter (const Vector& v, bdata a) const
    { return (rotate (center (), center () + v, a)); }

bool operator == (const Box& b) const
    { return (_origin == b._origin &&
            _vX == b._vX && _vY == b._vY && _vZ == b._vZ &&
            _wX == b._wX && _wY == b._wY && _wZ == b._wZ); }
bool operator != (const Box& b) const
    { return (!(*this == b)); }
Box operator + (const Vector& v) const
    { return (Box (_origin + v, _vX, _vY, _wX, _wY, _wZ)); }
friend Box operator + (const Vector& v, const Box& b)
    { return (b + v); }
Box& operator += (const Vector& v)
    { *this = v + *this; return (*this); }
/** Only two boxes with the same orientation can be added, and the result is a new
one containing both.
    This method is very important to control specific methods in composity entities
e.g. */
Box operator + (const Box& b) const;
Box& operator += (const Box& b)
    { *this = b + *this; return (*this); }
Box operator - (const Vector& v) const
    { return (*this + (-v)); }
friend Box operator - (const Vector& v, const Box& b)
    { return (b - v); }
Box& operator -= (const Vector& v)
    { *this = -v + *this; return (*this); }
friend std::ostream& operator << (std::ostream& o, const Box& b)
    { return (o << b.origin () << "|"
            << b.vX () << "|" << b.vY () << "|" << b.vZ () << "|"
            << b.width () << "|" << b.height () << "|" << b.depth ()); }
}

/** Creates a orthogonal cube containing all points passed as parameter */
static Box boxToInclude (const Positions& pos);
/** Same but with the list of rectangles (more complicated). */
static Box boxToInclude (const Rectangles& rects);

```

```

private:
void calculateInternalPoints ();

private:
Position _origin;
Vector _vX, _vY, _vZ;
bdata _wX, _wY, _wZ;

// Implementation
Rectangle _rTop, _rBase, _rFront, _rBack, _rRight, _rLeft;
};

}

#endif

// End of the file
/*}*/

```

41: Clase Box. Definición include

```

#include <Common/box.hpp>

QGAMES::Box QGAMES::Box::_noBox = QGAMES::Box ();

// ---
QGAMES::Box::Box (const QGAMES::Rectangle& bottom, QGAMES::bdata wz)
: _origin (Position::_noPoint),
_vX (Vector::_noPoint), _vY (Vector::_noPoint), _vZ (Vector::_noPoint),
_wX (_BD 0), _wY (_BD 0), _wZ (_BD 0),
_rBase (Rectangle::_noRectangle), _rTop (Rectangle::_noRectangle),
_rFront (Rectangle::_noRectangle), _rBack (Rectangle::_noRectangle),
_rRight (Rectangle::_noRectangle), _rLeft (Rectangle::_noRectangle)
{
    switch (bottom.plane ())
    {
        case QGAMES::Rectangle::_XY:
        {
            _origin = bottom.pos1 ();
            _vX = QGAMES::Vector (_BD 1, _BD 0, _BD 0);
            _vY = QGAMES::Vector (_BD 0, _BD 1, _BD 0);
            _vZ = QGAMES::Vector (_BD 0, _BD 0, _BD -1);
            _wX = bottom.pos2 ().posX () - bottom.pos1 ().posX ();
            _wY = bottom.pos2 ().posY () - bottom.pos1 ().posY ();
            _wZ = wz;
        }
        break;

        case QGAMES::Rectangle::_XZ:
        {
            _origin = bottom.pos1 ();
            _vX = QGAMES::Vector (_BD 1, _BD 0, _BD 0);
            _vY = QGAMES::Vector (_BD 0, _BD 0, _BD 1);
            _vZ = QGAMES::Vector (_BD 0, _BD -1, _BD 0);
            _wX = bottom.pos2 ().posX () - bottom.pos1 ().posX ();
            _wY = bottom.pos2 ().posZ () - bottom.pos1 ().posZ ();
            _wZ = wz;
        }
        break;

        case QGAMES::Rectangle::_YZ:
        {
            _origin = bottom.pos1 ();
            _vX = QGAMES::Vector (_BD 0, _BD 0, _BD 1);
            _vY = QGAMES::Vector (_BD 0, _BD 1, _BD 0);
            _vZ = QGAMES::Vector (_BD -1, _BD 0, _BD 0);
            _wX = bottom.pos2 ().posZ () - bottom.pos1 ().posZ ();
            _wY = bottom.pos2 ().posY () - bottom.pos1 ().posY ();
            _wZ = wz;
        }
        break;
    }
}

```

default:

```

        _origin = bottom.pos1 ();
        _vX = bottom.pos3 () - bottom.pos1 ();
        _vY = bottom.pos4 () - bottom.pos1 ();
        _vZ = -_vX.crossProduct (_vY);
        _wX = _vX.module (); _wY = _vY.module (); _wZ = wZ;
        _vX.normalize (); _vY.normalize (); _vZ.normalize ();
    }

    break;

}

calculateInternalPoints ();
}

// ---
QGAMES::Box QGAMES::Box::operator + (const QGAMES::Box& b) const
{
    // Only two boxes with the same orientation can be added
    if (_vX != b._vX || _vY != b._vY || _vZ != b._vZ)
        return (QGAMES::Box::_noBox);

    QGAMES::Position nO (std::min (_origin.posX (), b._origin.posX ()),
                         std::min (_origin.posY (), b._origin.posY ()),
                         std::min (_origin.posZ (), b._origin.posZ ()));

    QGAMES::bdata nWX = (nO.posX () == _origin.posX ())
        ? b._wX + _vX.dotProduct (b._origin - _origin)
        : _wX + _vX.dotProduct (_origin - b._origin);
    QGAMES::bdata nWY = (nO.posY () == _origin.posY ())
        ? b._wY + _vY.dotProduct (b._origin - _origin)
        : _wY + _vY.dotProduct (_origin - b._origin);
    QGAMES::bdata nWZ = (nO.posZ () == _origin.posZ ())
        ? b._wZ + _vZ.dotProduct (b._origin - _origin)
        : _wZ + _vZ.dotProduct (_origin - b._origin);

    return (QGAMES::Box (nO, _vX, _vY, nWX, nWY, nWZ));
}

// ---
QGAMES::Box QGAMES::Box::boxToInclude (const QGAMES::Positions& pos)
{
    if (pos.size () == 0)
        return (QGAMES::Box::_noBox);

    QGAMES::Rectangles result;

    QGAMES::bdata xMax, yMax, zMax;
    QGAMES::bdata xMin, yMin, zMin;
    xMax = yMax = zMax = __MINBDATA__;
    xMin = yMin = zMin = __MAXBDATA__;
    for (QGAMES::Positions::const_iterator i = pos.begin (); i != pos.end (); i++)
    {
        QGAMES::Position p = (*i);
        if (p.posX () > xMax) xMax = p.posX ();
        if (p.posY () > yMax) yMax = p.posY ();
        if (p.posY () > zMax) zMax = p.posZ ();
        if (p.posX () < xMin) xMin = p.posX ();
        if (p.posY () < yMin) yMin = p.posY ();
        if (p.posY () < zMin) zMin = p.posZ ();
    }

    QGAMES::Position org (xMin, yMin, zMin);
    QGAMES::bdata wx = xMax - xMin;
    QGAMES::bdata wy = yMax - yMin;
    QGAMES::bdata wz = zMax - zMin;
    QGAMES::Vector vx (wx, __BD 0, __BD 0);
    QGAMES::Vector vy (__BD 0, wy, __BD 0);

    return (QGAMES::Box (org, vx, vy, wx, wy, wz));
}

// ---
QGAMES::Box QGAMES::Box::boxToInclude (const QGAMES::Rectangles& rects)
{
    QGAMES::Positions pos;
    for (QGAMES::Rectangles::const_iterator i = rects.begin (); i != rects.end (); i++)

```

```

    {
        pos.push_back ((*i).pos1()); pos.push_back ((*i).pos2());
        pos.push_back ((*i).pos3()); pos.push_back ((*i).pos4());
    }

    return (QGAMES::Box::boxToInclude (pos));
}

// ---
void QGAMES::Box::calculateInternalPoints ()
{
    // It has to be called only when the basic variables are calculated
    QGAMES::Position pD1 = _origin;
    QGAMES::Position pD2 = pD1 + (_wX * _vX) + (_wY * _vY);
    QGAMES::Position pD3 = pD1 + (_wX * _vX);
    QGAMES::Position pD4 = pD1 + (_wY * _vY);
    QGAMES::Position pU1 = _origin + (_wZ * _vZ);
    QGAMES::Position pU2 = pU1 + (_wX * _vX) + (_wY * _vY);
    QGAMES::Position pU3 = pU1 + (_wX * _vX);
    QGAMES::Position pU4 = pU1 + (_wY * _vY);

    _rTop    = QGAMES::Rectangle (pU1, pU2, _vZ);
    _rBase   = QGAMES::Rectangle (pD1, pD2, _vZ);
    _rFront  = QGAMES::Rectangle (pU4, pD2, _vY);
    _rBack   = QGAMES::Rectangle (pU1, pD3, _vY);
    _rRight  = QGAMES::Rectangle (pU2, pD3, _vX);
    _rLeft   = QGAMES::Rectangle (pU1, pD4, _vX);
}

```

42: Clase Box. Definición src

Bloque 5: Ecuaciones de movimiento adaptadas a juegos

IGNACIO CEA FORNIÉS

Introducción

Para el desarrollo de las ecuaciones de movimiento se considerarán entidades físicas sujetas a la acción de fuerzas vectoriales de dirección, sentido y módulo constantes en función del tiempo.

Sea:

$$\sum \overrightarrow{F(t)} = \overrightarrow{F_0}$$

Constante en el tiempo, la resultante de las fuerzas que actúan sobre una entidad cualquiera del juego durante un periodo de tiempo determinado.

Si en un momento concreto ésta resultante cambiara su dirección, sentido o módulo, consideraríamos una nueva ecuación de movimiento, y todo el razonamiento que sigue continuaría siendo válido.

Las ecuaciones de movimiento

Es decir, vamos a suponer que esta resultante es constante en el intervalo $(t_0, t_1) \in T$.

La primera ley de Newton nos dice:

$$\sum \overrightarrow{F(t)} = m \overrightarrow{a(t)}$$

Entonces:

$$\overrightarrow{a(t)} = \frac{\sum \overrightarrow{F(t)}}{m}$$

Y si, $\overrightarrow{F(t)}$ es constante en el tiempo, entonces a también lo será:

$$\overrightarrow{a(t)} = \overrightarrow{a_0}$$

A partir de ahí, sabiendo que:

$$\frac{d\overrightarrow{v(t)}}{dt} = \overrightarrow{a(t)} = \overrightarrow{a_0}$$

Dónde $\overrightarrow{v(t)}$ es la velocidad de la entidad dependiendo del tiempo, entonces:

$$\overrightarrow{v(t)} = \overrightarrow{v_0} + \overrightarrow{a_0}t$$

donde $\overrightarrow{v_0}$ es la velocidad inicial de la entidad antes de que empezara a actuar la fuerza $\overrightarrow{F_0}$ sobre él.

y sabiendo que:

$$\frac{d\overrightarrow{p(t)}}{dt} = \overrightarrow{v(t)} = \overrightarrow{v_0} + \overrightarrow{a_0}t$$

Dónde $\overrightarrow{p(t)}$ es la posición de la entidad en el tiempo, se tiene:

$$\overrightarrow{p(t)} = \overrightarrow{p_0} + \overrightarrow{v_0}t + \frac{1}{2}\overrightarrow{a_0}t^2$$

Donde $\overrightarrow{p_0}$ es la posición inicial de la partícula en el espacio, antes de que empezara a actuar la fuerza $\overrightarrow{F_0}$ sobre él.

En resumen:

$$\overrightarrow{p(t)} = \overrightarrow{p_0} + \overrightarrow{v_0}t + \frac{1}{2}\overrightarrow{a_0}t^2$$

$$\overrightarrow{v(t)} = \overrightarrow{v_0} + \overrightarrow{a_0}t$$

$$\overrightarrow{a(t)} = \overrightarrow{a_0}$$

47: Ecuaciones generales de movimiento continuas (en función de t)

La forma básica de construir juegos es basándose en un bucle principal. En dicho bucle se realizan 2 tareas básicas: Calcular las posiciones de las entidades en función de la lógica del

juego, y dibujar luego el conjunto (fondos, entidades, etc.). Para que un juego se aprecie adecuadamente se deben alcanzar un mínimo de fotogramas por segundo (f.p.s) en el entorno o superior a los 60 (alienado habitualmente también con la frecuencia de refresco del monitor usado). Teniendo en cuenta esto, podemos intentar convertir la anterior expresión continua en una expresión discreta.

Sea:

n el número de bucle que está ejecutando el juego desde su inicio ($\in \mathbb{N}$), y fps constante el número de bucles por segundo ejecutados en el juego ($\in \mathbb{N}$). Entonces:

$$t = \frac{n}{fps}$$

Y podríamos poner las ecuaciones anteriores, de la siguiente forma:

$$\begin{aligned}\overrightarrow{p(n)} &= \overrightarrow{p_0} + \frac{n}{fps} \overrightarrow{v_0} + \frac{1}{2} \left(\frac{n}{fps} \right)^2 \overrightarrow{a_0} \\ \overrightarrow{v(n)} &= \overrightarrow{v_0} + \frac{n}{fps} \overrightarrow{a_0} \\ \overrightarrow{a(n)} &= \overrightarrow{a_0}\end{aligned}$$

48:Ecuaciones generales de movimiento discretas (en función de fracciones de fps)

Siendo éstas, funciones (simplificadas) de muestreo de las anteriores continuas.

Dado que, tal y como hemos comentado anteriormente, cualquier juego es una sucesión continua de bucles, sería útil que el cálculo en las variaciones de las variables velocidad y posición (la aceleración es constante), pudieran inferirse a partir del valor proporcionado por el anterior bucle.

La variación de la posición por bucle sería entonces:

$$\begin{aligned}\Delta \vec{p} &= \overrightarrow{p(n+1)} - \overrightarrow{p(n)} \\ \Delta \vec{p} &= \overrightarrow{p_0} + \frac{n+1}{fps} \overrightarrow{v_0} + \frac{1}{2} \left(\frac{n+1}{fps} \right)^2 \overrightarrow{a_0} - \overrightarrow{p_0} - \frac{n}{fps} \overrightarrow{v_0} - \frac{1}{2} \left(\frac{n}{fps} \right)^2 \overrightarrow{a_0} \\ \Delta \vec{p} &= \frac{1}{fps} \overrightarrow{v_0} + \frac{1}{2fps^2} \overrightarrow{a_0} + \frac{1}{fps^2} n \overrightarrow{a_0}\end{aligned}$$

Y la de la velocidad:

$$\begin{aligned}\Delta \vec{v} &= \overrightarrow{v(n+1)} - \overrightarrow{v(n)} \\ \Delta \vec{v} &= \overrightarrow{v_0} + \frac{n+1}{fps} \overrightarrow{a_0} - \overrightarrow{v_0} - \frac{n}{fps} \overrightarrow{a_0} \\ \Delta \vec{v} &= \frac{1}{fps} \overrightarrow{a_0}\end{aligned}$$

Y la de la aceleración:

$$\Delta \vec{a} = \overrightarrow{a(n+1)} - \overrightarrow{a(n)}$$

$$\Delta \vec{a} = \vec{a}_0 - \vec{a}_0$$

$$\Delta \vec{a} = 0$$

Como era de esperar, ya que hemos partido de que la aceleración es constante.

Poniéndolo los valores obtenidos en función del valor del anterior bucle:

$$\begin{aligned}\overrightarrow{a(n+1)} &= \overrightarrow{a_0(n)} = \overrightarrow{a_0} \\ \overrightarrow{v(n+1)} &= \overrightarrow{v(n)} + \frac{1}{fps} \overrightarrow{a_0} \\ \overrightarrow{p(n+1)} &= \overrightarrow{p(n)} + \frac{1}{fps} \overrightarrow{v_0} + \frac{1}{2fps^2} \overrightarrow{a_0} + \frac{1}{fps^2} n \overrightarrow{a_0}\end{aligned}$$

Y si llamamos:

$$\begin{aligned}\overrightarrow{K_1} &= \frac{1}{fps} \overrightarrow{a_0} \\ \overrightarrow{K_2} &= \frac{1}{fps} \overrightarrow{v_0} \\ \overrightarrow{K_3} &= \frac{1}{2fps^2} \overrightarrow{a_0} \\ \overrightarrow{K_4(n)} &= \frac{1}{fps^2} n \overrightarrow{a_0}\end{aligned}$$

Entonces:

$$\begin{aligned}\overrightarrow{a(n+1)} &= \overrightarrow{a_0} \\ \overrightarrow{v(n+1)} &= \overrightarrow{v(n)} + \overrightarrow{K_1} \\ \overrightarrow{p(n+1)} &= \overrightarrow{p(n)} + \overrightarrow{K_2} + \overrightarrow{K_3} + \overrightarrow{K_4(n)}\end{aligned}$$

Y calculando también $\overrightarrow{K_4(n)}$ también de forma incremental:

$$\begin{aligned}\overrightarrow{\Delta K_4} &= \overrightarrow{K_4(n+1)} - \overrightarrow{K_4(n)} \\ \overrightarrow{\Delta K_4} &= \frac{1}{fps^2} (n+1) \overrightarrow{a_0} - \frac{1}{fps^2} n \overrightarrow{a_0} \\ \overrightarrow{\Delta K_4} &= \frac{1}{fps^2} \overrightarrow{a_0} = 2 \overrightarrow{K_3}\end{aligned}$$

Por tanto:

$$\overrightarrow{K_4(n+1)} = \overrightarrow{K_4(n)} + 2 \overrightarrow{K_3}$$

Ya podemos inferir nuestro algoritmo de ecuaciones de movimiento.

En resumen, en el primer bucle de cualquier movimiento se han de calcular las siguientes variables:

$$\vec{K}_1 = \frac{1}{fps} \vec{a}_0$$

$$\vec{K}_2 = \frac{1}{fps} \vec{v}_0$$

$$\vec{K}_3 = \frac{1}{2fps^2} \vec{a}_0$$

$$\vec{K}_4 = \vec{0}$$

$$\vec{a} = \vec{a}_0$$

$$\vec{v} = \vec{v}_0$$

$$\vec{p} = \vec{p}_0$$

49: Valores que calcular antes de iniciar cualquier movimiento o al variar cualquiera las fuerzas que actúan

Y, a partir de ahí, en cada bucle:

$$\vec{K}_4 = \vec{K}_4 + 2\vec{K}_3$$

$$\vec{v} = \vec{v} + \vec{K}_1$$

$$\vec{p} = \vec{p} + \vec{K}_2 + \vec{K}_3 + \vec{K}_4$$

50: Valores que calcular en cada uno de los bucles de un juego asociado a los movimientos

Mientras la aceleración (y en el fondo las fuerzas que actúan sobre la entidad) se mantengan constantes.

Son fórmulas bastante sencillas de recordar y de programar en un ordenador, además de ser rápidas, imprescindibles en cada bucle. Recuerde que para el correcto funcionamiento de éstas los valores utilizados deben ser reales.

Los valores de posición, velocidad y aceleración están notados en magnitudes del sistema internacional. Esto es, metros, metros por segundo y metros por segundo al cuadrado, respectivamente.

Nótese adicionalmente que son todo magnitudes vectoriales y, por tanto, perfectamente válidas para juegos de 2 y 3 dimensiones.

La traslación de la posición \vec{p} a pixeles dependerá de las dimensiones que se quieran reflejar en el escenario del juego. Supongamos, por tanto, un factor de conversión entre pixeles y metros f , de tal manera que:

$$1 \text{ metro} = f \text{ pixeles}$$

Factor que habría que aplicar (multiplicando) a las constantes iniciales \vec{k}_1 , \vec{k}_2 y \vec{k}_3 identificadas anteriormente (no así a \vec{k}_4 que es función de \vec{k}_3), así como a los valores iniciales de \vec{a}_0 , \vec{v}_0 y \vec{p}_0 .

Cómo se decía hace un par de párrafos, todo el razonamiento llevado a cabo parte de la consideración de que el resultado de las fuerzas que actúan sobre la entidad se mantiene constante en el tiempo. Si éstas variaran, las fórmulas serían válidas en los períodos entre cambios.

Los algoritmos tendrían que, con cada cambio en su valor:

- Considerar como valores iniciales de \vec{v}_0 , \vec{p}_0 y \vec{k}_4 los que la entidad tuviera antes de empezar el nuevo bucle.
- Volver a calcular el valor de la aceleración en función de la nueva disposición de fuerzas y el valor de la masa considerada para la entidad.
- Volver a calcular las constantes \vec{k}_1 , \vec{k}_2 y \vec{k}_3 .
- \vec{k}_4 conserva su último valor.
- Continuar normalmente con la ejecución en el siguiente bucle.

Insistimos en que todas las ecuaciones anteriores suponen que fps se mantiene constante en el tiempo. Ante fuertes variaciones de éste en el transcurso del juego, las ecuaciones anteriores dejarían de ser válidas.

Si fps redujera su valor, los movimientos basados en las anteriores ecuaciones se ralentizarían, pues se estarían aplicando variables de valor menor que las que realmente debieran aplicarse. De igual manera se puede razonar en sentido contrario.

Sin embargo, el considerar que fps se mantiene constante (o aproximadamente constante) toda vez que el juego ha arrancado es una aproximación correcta.

Control del movimiento de las entidades de un juego

Imaginemos que los valores de las constantes $\vec{K}_1, \vec{K}_2, \vec{K}_3$ y \vec{K}_4 de las anteriores ecuaciones fueran lo suficientemente grandes como para que los incrementos de posición que se produjeran en cada bucle para una entidad determinada del juego, traducidas a píxeles equivalentes, fueran netamente superiores a la unidad.

Supongamos que la entidad a la que se refiere este supuesto se estuviera moviendo por una plataforma (digamos en un juego tipo Mario Bros(C)) y que en su camino hubiera una pared. Para dar por buena una nueva posición calculada debería antes verificarse que nuestra entidad no ha chocado ya con la pared.

Si antes de iniciar ese nuevo bucle, nuestra entidad estuviera por ejemplo 5 píxeles delante del límite del muro, y el nuevo cálculo le lleva a moverse 6 píxeles, la comprobación de viabilidad justo posterior impediría el movimiento, dejando a nuestra entidad parada en realidad 5 píxeles antes de su máximo posible.

Esta diferencia sería aún mayor cuanto mayor fuera la velocidad de la entidad (y en consecuencia más píxeles pudiera avanzar por bucle). Adicionalmente, es posible que no siempre antes de parar el movimiento la entidad estuviera a 5 píxeles de su límite, sino a 2, 3, 4, por lo que no siempre se pararía a la misma distancia de ese límite. Ambos efectos juegan en contra de la jugabilidad y realismo y, por tanto, debe buscarse una forma de corregirlos.

La solución aparentemente es sencilla: Calcular todos los píxeles intermedios entre el inicial (siguiendo con el ejemplo el píxel -5 respecto al límite) y el final del movimiento (el píxel +1 respecto al límite) y realizar la comprobación de si supera o no el límite en cada uno de ellos.

Fácil de decir y no tan fácil de implementar. Si el movimiento que está siguiendo la entidad es, por ejemplo, horizontal, sería fácil inferir todos esos puntos intermedios (en un juego 2D bastaría con dejar constante la coordenada vertical y variar de uno en uno la coordenada horizontal). Pero ¿Y si es, por ejemplo, circular? ¿Y si es lineal, pero con una pendiente diferente a la del movimiento horizontal? ¿Podemos encontrar una forma general de hacerlo?

Como ejemplo, limitémonos, de momento, a un mundo 2D. Imaginemos, como ejemplo, que nuestra entidad se moviera siguiendo la siguiente función:

$$f(x) = x^{\frac{5}{4}} + 10,$$

siendo x la posición en el eje \overrightarrow{OX} en píxeles. Si nuestra entidad pasara de estar en el píxel $x = 100$ al píxel $x = 104$, en el eje \overrightarrow{OY} se movería 16 píxeles, del 326 al 342.

$$f(100) = 326 \text{ y } f(104) = 342; f(104) - f(100) = 16$$

Si eso hubiera sucedido del 150 al 154, se movería 18 píxeles:

$$f(150) = 534 \text{ y } f(154) = 552; f(154) - f(150) = 18$$

Visualmente nuestra entidad, en bucles sucesivos del juego, ocuparía las posiciones (100, 326) y (101, 342) en el primer caso y la (150, 534) y la (151, 552) en el segundo, pero ¿en qué posiciones intermedias deberíamos comprobar si se ha superado el límite para cada uno de los casos?

Una aproximación sencilla sería comprobarlo en todos los puntos de la recta que uniera el punto inicial con el punto final. La ecuación de esa recta tendría la forma:

$$l(x) = mx + b$$

En donde:

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)}$$
$$b = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1} = y_1 - mx_1$$

Donde m es la pendiente de la recta y b el parámetro constante. Para el primer movimiento la ecuación, quedaría:

$$m = \frac{(342 - 326)}{(104 - 100)} = \frac{16}{4} = 4$$
$$b = \frac{(104 \times 326) - (100 \times 342)}{104 - 100} = -74$$

Entonces, en este caso:

$$l_1(x) = 4x - 74$$

Tomemos los puntos intermedios entre 100 y 104:

$$\begin{aligned}l_1(100) &= 326 \\l_1(101) &= 330 \\l_1(102) &= 334 \\l_1(103) &= 338 \\l_1(104) &= 342\end{aligned}$$

Es ahí, en cada uno de esos puntos intermedios, donde deberíamos verificar que se cumplen los límites del movimiento; es decir: que un algoritmo posible de movimiento podría ser:

- Calcular el nuevo punto final
- Calcular la pendiente de la recta y el parámetro independiente de la recta que los une
- Calcular para cada uno de los puntos intermedios si es o no posible ese punto movimiento.
 - Si es posible, moverse.
 - Si no lo es, parar el movimiento en ese punto y continuar con otros procesos.

Pero ¿Qué sucede si el límite estuviera, por ejemplo, en la coordenada $Y = 336$? Esa es una coordenada por la que nunca se pasa explícitamente. Luego, aunque ajustáramos a una recta los dos puntos consecutivos de una curva, estaríamos en la misma situación anterior a efectos prácticos.

El algoritmo que hemos definido, por tanto, el de simplemente verificar los puntos que da la fórmula de la recta que une los dos puntos inicial y final producidos por una curva, no produce un resultado continuo en puntos y por ende no es posible utilizarlo para determinar límites de forma precisa. Sin duda que es un buen añadido al algoritmo que simplemente comprobara los puntos original y destino proporcionados por la ecuación de la curva, pero es aún incompleto.

Existen, sin embargo, algoritmos que permiten trazar una recta de píxeles continua y entre dos puntos. Entre ellos está el algoritmo de Bresenham.

IGNACIO CEA FORNIÉS

Algoritmo de Bresenham

Este algoritmo fue descrito por Jack Elton Bresenham en el verano de 1962 (aunque se le atribuye la publicación en 1965) para dibujar en un plotter, y hoy es tremadamente utilizado para el dibujo en pantallas gráficas. Aquí lo vamos a utilizar para determinar píxeles sobre los que realizar comprobaciones en un juego 2D. Posteriormente lo extenderemos a 3D.

Bresenham 2D

Partamos de las funciones básicas de una recta entre dos puntos vistas en el punto anterior:

$$l(x) = mx + b$$

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)}$$

$$b = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1} = y_1 - mx_1$$

Aplicando estas fórmulas al incremental de movimiento, podemos decir que:

$$\Delta y = l(x + \Delta x) - l(x) = m(x + \Delta x) + b - mx - b = m\Delta x$$

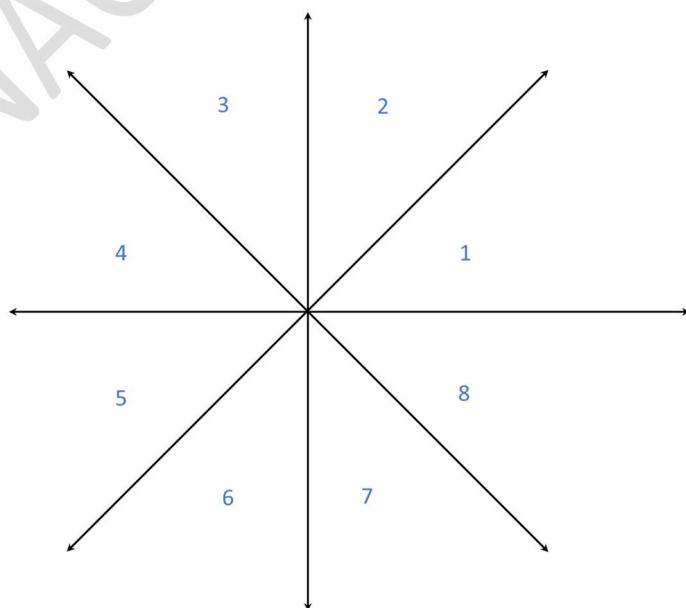
$$\Delta y = m\Delta x; \Delta x = \frac{\Delta y}{m}; m = \frac{\Delta y}{\Delta x}$$

Si $|m|$ es < 1 , la línea será más horizontal que vertical. Y cuanto más cercano a 0 sea m , más horizontal será la línea, llegando a que cuando es 0, la línea es completamente horizontal.

Si $|m|$ es > 1 , la línea será más vertical que horizontal. Y según m tienda a infinito la línea tenderá a ser más vertical.

Cuando $|m| = 1$, la línea es la diagonal de uno de los cuadrantes (y del situado en el lado opuesto también).

Teniendo en cuenta lo dicho, dividamos el espacio 2D en 8 secciones:



Consideremos ahora que $|m| < 1$ y que $m > 0$ (positivo). Es decir, que nuestra recta pasa por el cuadrante 1 y 5. Δy se puede determinar mediante variaciones unitarias de x . Llamemos (x_k, y_k) a cada punto entre los valores x_0 y x_1 de la recta. Ambos enteros, al estar hablando de píxeles.

Así, situada la entidad en el punto (x_k, y_k) el siguiente punto será, para que haya continuidad visual en el movimiento: $(x_k + 1, y_k)$ o $(x_k + 1, y_k + 1)$. Aquel que, en realidad, diste menos del que daría la curva original:

$$y = m(x_k + 1) + b$$

Las distancias a ese punto real de cada una de las dos posibilidades serían:

$$d_1 = y - y_k = m(x_k + 1) + b - y_k$$

$$d_2 = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$$

Creemos el parámetro:

$$\begin{aligned} p_k &= \Delta x(d_1 - d_2) = \Delta x\left(\frac{\Delta y}{\Delta x}(x_k + 1) + b - y_k - \left(y_k + 1 - \frac{\Delta y}{\Delta x}(x_k + 1) - b\right)\right) \\ &= \Delta x\left(2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1\right) = 2\Delta y x_k - 2\Delta x y_k + c \end{aligned}$$

Dónde:

- El signo de p_k es el mismo que el de la diferencia entre d_1 y d_2 , dado que Δx es positivo puesto que, en nuestro modelo de inicio la recta pasa por los cuadrantes 1 y 5.
- $c = 2\Delta y + 2b\Delta x - \Delta x$ una constante.
- Si d_1 es menor que d_2 , entonces y_k está más cerca de la curva real que $y_k + 1$, y además p_k es negativo (esto nos irá sirviendo luego para la lógica del algoritmo)

Los cambios de coordenadas suceden de 1 en 1, pues estamos moviéndonos de píxel en píxel.

Por tanto, si en cada punto podemos calcular p_k y atendiendo a su signo, podemos concluir que:

- Si es negativo, hay que elegir y_k .
- Si es positivo, hay que elegir $y_k + 1$.

Este proceso habrá de repetirse desde x_0 hasta x_1 . Y además en el cálculo de p_k está involucrada y_k que es justo una de las variables que tenemos que decidir. Dado que es necesario calcular p_k en cada bucle, veamos si su valor se puede calcular a partir del obtenido en el ciclo anterior y eso simplifica las cosas:

$$p_{k+1} = 2\Delta y(x_{k+1}) - 2\Delta x y_{k+1} + c$$

$$\Delta p_k = p_{k+1} - p_k = 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

$(y_{k+1} - y_k)$ puede ser 0 o 1 (nos movemos por píxel), según sea el signo de p_k (igual que el de $d_1 - d_2$, dado que Δx es positivo).

El valor de p_0 sería:

$$p_0 = 2\Delta y - \Delta x$$

Por tanto, el algoritmo de control movimiento basado en Bresenham consistiría en:

- Se calcula el nuevo punto en la curva de movimiento $f(p_2) = (x_2, y_2)$ que sigue la entidad.
- Se calculan los parámetros: Δx , $2\Delta x$, Δy y $2\Delta y$ a partir de:
 - $\Delta x = x_2 - x_1$; $\Delta y = y_2 - y_1$
- Se calcula el parámetro p como: $2\Delta y - \Delta x$.
- Se calcula $y_k = y_1$.
- Se inicia un bucle hasta que $x_k = x_1$ sea igual a x_2 :
 - $x_k = x_k + 1$.
 - Se verifica el valor de p :
 - Si es positivo:
 - $p = p + 2\Delta y - 2\Delta x$.
 - Si es negativo:
 - $y_k = y_k + 1$; $p = p + 2\Delta y$.
 - ¿Es posible movernos a la posición calculada: (x_k, y_k) ?:
 - Si: La entidad se mueve a la nueva posición.
 - No: La entidad se para, se abandona el bucle y se notifica la situación de parada.

43: Primera versión del algoritmo de movimiento basado en Bresenham

Si la recta pasara por el cuadrante 2 – 6, sería más vertical que horizontal. En este caso, se demuestra siguiendo razonamientos similares a los realizados anteriormente que:

- El bucle sería en torno a la variable y , y no a la x .
- Habría que cambiar los Δx por Δy y viceversa.
- La elección de punto sería entre $(x_k, y_k + 1)$ cuando p es negativo y $(x_k + 1, y_k + 1)$ cuando es positivo.

Y si m fuera negativo, es decir que identificara rectas que pasan por los cuadrantes 2 y 8, o 4 y 7, bastaría con entender que las coordenadas e incrementales posibles serían no $x_k + 1$ o $y_k + 1$, sino $x_k - 1$ o $y_k - 1$ respectivamente-

El algoritmo, al completo, quedaría contemplando estas dos circunstancias en algo similar a:

- Se calcula el nuevo punto en la curva de movimiento $f(p_2) = (x_2, y_2)$ que sigue la entidad.
- Se calculan los parámetros: Δx , $2\Delta x$, Δy y $2\Delta y$ a partir de:
 - $\Delta x = |x_2 - x_1|$; $\Delta y = |y_2 - y_1|$
- Se calculan dos variables intermedias:
 - $incx = (x_2 - x_1) \geq 0 \Rightarrow 1$; $(x_2 - x_1) < 0 \Rightarrow -1$
 - $incy = (y_2 - y_1) \geq 0 \Rightarrow 1$; $(y_2 - y_1) < 0 \Rightarrow -1$
- Si $\Delta x \geq \Delta y$, la recta es más horizontal que vertical:
 - Se calcula el parámetro p como: $2\Delta y - \Delta x$.
 - Se calcula $y_k = y_1$.
 - Se inicia un bucle hasta que $x_k = x_1$ sea igual a x_2 :
 - $x_k = x_k + incx$.

- Se verifica el valor de p :
 - Si es positivo:
 - $p = p + 2\Delta y - 2\Delta x$.
 - Si es negativo:
 - $y_k = y_k + incy; p = p + 2\Delta y$.
- ¿Es posible movernos a la posición calculada?
 - Si: La entidad se mueve.
 - No: La entidad se para, se abandona el bucle y se notifica la situación.
- Si $\Delta x < \Delta y$, la recta es vertical más vertical que horizontal:
 - Se calcula el parámetro p como: $2\Delta x - \Delta y$.
 - Se calcula $x_k = x_1$.
 - Se inicia un bucle hasta que $y_k = y_1$ sea igual a y_2 :
 - $y_k = y_k + incy$.
 - Se verifica el valor de p :
 - Si es positivo:
 - $p = p + 2\Delta x - 2\Delta y$.
 - Si es negativo:
 - $x_k = x_k + incx; p = p + 2\Delta x$.
 - ¿Es posible movernos a la posición calculada?:
 - Si: La entidad se mueve.
 - No: La entidad se para, se abandona el bucle y se notifica la situación.

44: Ecuaciones de control de movimiento 2D basadas en Bresenham

Besenham 3D

Pero ¿Y si el movimiento fuera en 3D? El algoritmo se complejiza un tanto, pero manteniendo siempre la misma racionalidad anterior.

En 2D hay dos situaciones posibles: Que prime el eje \overrightarrow{OX} sobre el eje \overrightarrow{OY} o al revés. En el mundo 3D habrá 3 posibilidades: Que primer el \overrightarrow{OX} sobre los otros dos, o lo haga el eje \overrightarrow{OY} , o lo haga el \overrightarrow{OZ} .

Las variaciones en la variable p , que es la responsable de la decisión de tomar un punto u otro como siguiente, tienen la siguiente estructura: p es un dato que involucra siempre valores de la recta en los dos ejes involucrados. Cuando el eje dominante es el \overrightarrow{OX} , se incrementa en 2 veces Δy (el valor del eje no dominante) y se resta un término Δx (el valor del eje dominante) cuando p es negativo.

Para el caso de 3D parece lógico pensar que vamos a necesitar dos p diferentes: p_1 y p_2 , y que cada uno de ellos relacionaría datos de cada uno de los ejes no dominantes con el valor del eje dominante. Por ejemplo, si el eje dominante fuera nuevamente el \overrightarrow{OX} :

- p_1 : sumaría siempre $2\Delta y$ y restaría Δx cuando el valor de p_1 fuera positivo.
- p_2 : sumaría $2\Delta z$ y restaría Δx cuando el valor de p_2 fuera positivo.

Además, sus valores iniciales habría que calcularlos:

- $p_{10} = 2\Delta y - \Delta x$ y $p_{20} = 2\Delta z - \Delta x$.

Ya podemos entonces vislumbrar el algoritmo de Bresenham para un mundo 3D (je incluso mentalmente podríamos extender a cualquier dimensión, aunque no nos hace falta!):

- Se calcula el nuevo punto en la curva de movimiento $f(\mathbf{p}_2) = (x_2, y_2, z_2)$ que sigue la entidad.
- Se calculan los parámetros: $\Delta x, 2\Delta x, \Delta y$ y $2\Delta y$ a partir de:
 - $\Delta x = |x_2 - x_1|$; $\Delta y = |y_2 - y_1|$; $\Delta z = |z_2 - z_1|$
- Se calculan tres variables intermedias:
 - $incx = (x_2 - x_1) \geq 0 \Rightarrow 1$ o $(x_2 - x_1) < 0 \Rightarrow -1$
 - $incy = (y_2 - y_1) \geq 0 \Rightarrow 1$ o $(y_2 - y_1) < 0 \Rightarrow -1$
 - $incz = (z_2 - z_1) \geq 0 \Rightarrow 1$ o $(z_2 - z_1) < 0 \Rightarrow -1$
- Si $\Delta x \geq \Delta y$ y además $\Delta x \geq \Delta z$, la recta es más horizontal que vertical o profunda:
 - Se calculan los parámetros $p_1 = 2\Delta y - \Delta x$, $p_2 = 2\Delta z - \Delta x$
 - Se calcula $y_k = y_1$ y $z_k = z_1$
 - Se inicia un bucle hasta que $x_k = x_1$ sea igual a x_2 :
 - $x_k = x_k + incx$.
 - Se verifica el valor de p_1 :
 - Si es positivo:
 - $p_1 = p_1 + 2\Delta y - 2\Delta x$.
 - Si es negativo:
 - $y_k = y_k + incy$; $p_1 = p_1 + 2\Delta y$.
 - Se verifica el valor de p_2 :
 - Si es positivo:
 - $p_2 = p_2 + 2\Delta z - 2\Delta x$.
 - Si es negativo:
 - $z_k = z_k + incz$; $p_2 = p_2 + 2\Delta z$.
 - ¿Es posible movernos a la posición calculada?
 - Si: La entidad se mueve.
 - No: La entidad se para, se abandona el bucle y se notifica la situación.
 - Si $\Delta y \geq \Delta x$ y además $\Delta y \geq \Delta z$, la recta es más vertical que horizontal o profunda:
 - Se calculan los parámetros $p_1 = 2\Delta x - \Delta y$, $p_2 = 2\Delta z - \Delta y$
 - Se calcula $x_k = x_1$ y $z_k = z_1$
 - Se inicia un bucle hasta que $y_k = y_1$ sea igual a y_2 :
 - $y_k = y_k + incy$.
 - Se verifica el valor de p_1 :
 - Si es positivo:
 - $p_1 = p_1 + 2\Delta x - 2\Delta y$.
 - Si es negativo:
 - $x_k = x_k + incx$; $p_1 = p_1 + 2\Delta x$.
 - Se verifica el valor de p_2 :
 - Si es positivo:
 - $p_2 = p_2 + 2\Delta z - 2\Delta y$.
 - Si es negativo:
 - $z_k = z_k + incz$; $p_2 = p_2 + 2\Delta z$.
 - ¿Es posible movernos a la posición calculada?
 - Si: La entidad se mueve.

- No: La entidad se para, se abandona el bucle y se notifica la situación.
- Si $\Delta z \geq \Delta x$ y además $\Delta z \geq \Delta y$, la recta es más profunda que horizontal o vertical:
 - Se calculan los parámetros $p_1 = 2\Delta x - \Delta z$, $p_2 = 2\Delta y - \Delta z$
 - Se calcula $x_k = x_1$ y $y_k = y_1$
 - Se inicia un bucle hasta que $z_k = z_1$ sea igual a z_2 :
 - $z_k = z_k + incz$.
 - Se verifica el valor de p_1 :
 - Si es positivo:
 - $p_1 = p_1 + 2\Delta x - 2\Delta z$.
 - Si es negativo:
 - $x_k = x_k + incx$; $p_1 = p_1 + 2\Delta x$.
 - Se verifica el valor de p_2 :
 - Si es positivo:
 - $p_2 = p_2 + 2\Delta y - 2\Delta z$.
 - Si es negativo:
 - $y_k = y_k + incy$; $p_2 = p_2 + 2\Delta y$.
 - ¿Es posible movernos a la posición calculada?
 - Si: La entidad se mueve.
 - No: La entidad se para, se abandona el bucle y se notifica la situación.

1: Algoritmo de verificación de movimiento en una curva punto a punto 3D basado en Bresenham

Algoritmo que es de fácil implementación y que además sólo maneja números enteros.

Un ejemplo de implementación se ve en el siguiente código de QGAMES:

```
// ---
void QGAMES::IncrementalMovement::move (const QGAMES::Vector& d, const QGAMES::Vector&
a,
QGAMES::Entity* e)
{
    assert (e);

    // The number of times to repaet this part of the code
    // will depend on the speed!
    for (int s = 0; s < _speed; s++)
    {
        // If it the first time passing through here...
        // it is only needed just to prepare the movement!
        if (_firstMovement)
        {
            if (_data == NULL) setData ();
            // Internal Data to track the movement is set, if there is nothing before
            // Remember that the internal data variable can be set, i.e initializing the
movement
            _data -> _initialPosition = e -> position ();
            _data -> _nextDirection = _data -> _nextAcceleration =
                QGAMES::Vector::_cero; // Starts from the beggining...

            prepareMovement (_data, d, a, e); // Additional data preparation...

            // It is not longer the first movement...
            _firstMovement = false;
        }
        // Otherwise the entity is moved... using Bresemham's algorithm
        else
        {
            // What is the increment to execute?
            int l = abs (_data -> _incX); int dx2 = l << 1;
            int m = abs (_data -> _incY); int dy2 = m << 1;
            int n = abs (_data -> _incZ); int dz2 = n << 1;
            // Vectors with the little steps (one by one) per direction
```

```

QGAMES::Vector vX (_BD ((_data -> _incX < 0) ? -1 : 1), _BD 0, _BD 0);
QGAMES::Vector vY (_BD 0, _BD ((_data -> _incY < 0) ? -1 : 1), _BD 0);
QGAMES::Vector vZ (_BD 0, _BD 0, _BD ((_data -> _incZ < 0) ? -1 : 1));

int err_1, err_2;

// The X axis is the predominat!
if ((l >= m) && (l >= n))
{
    err_1 = dy2 - l; err_2 = dz2 - l;
    bool mX = true; bool mY = true; bool mZ = true;
    bool nMX = false; bool nMY = false; bool nMZ = false;
    for (int i = 0; i < l && mX; i++)
    {
        if (err_1 > 0)
        {
            err_1 -= dx2;
            if (mY && (mY = e -> canMove (vY, _data -> _nextAcceleration)))
                e -> setPosition (e -> position () + vY);
            else if (!nMY)
            {
                nMY = true; // Just one notification...
                notify (QGAMES::Event (_QGAMES_MOVEMENTNOTPOSSIBLE_, this));
                stopDuringMovement (vY, _data -> _nextAcceleration, e);
            }
        }

        if (err_2 > 0)
        {
            err_2 -= dx2;
            if (mZ && (mZ = e -> canMove (vZ, _data -> _nextAcceleration)))
                e -> setPosition (e -> position () + vZ);
            else if (!nMZ)
            {
                nMZ = true;
                notify (QGAMES::Event (_QGAMES_MOVEMENTNOTPOSSIBLE_, this));
                stopDuringMovement (vZ, _data -> _nextAcceleration, e);
            }
        }

        err_1 += dy2; err_2 += dz2;
        if (mX && (mX = e -> canMove (vX, _data -> _nextAcceleration)))
            e -> setPosition (e -> position () + vX);
        else if (!nMX)
        {
            nMX = true;
            notify (QGAMES::Event (_QGAMES_MOVEMENTNOTPOSSIBLE_, this));
            stopDuringMovement (vX, _data -> _nextAcceleration, e);
        }
    }
}

// The Y Axis is the predominat...
else if ((m >= l) && (m >= n))
{
    err_1 = dx2 - m; err_2 = dz2 - m;
    bool mX = true; bool mY = true; bool mZ = true;
    bool nMX = false; bool nMY = false; bool nMZ = false;
    for (int i = 0; i < m && mY; i++)
    {
        if (err_1 > 0)
        {
            err_1 -= dy2;
            if (mX && (mX = e -> canMove (vX, _data -> _nextAcceleration)))
                e -> setPosition (e -> position () + vX);
            else if (!nMX)
            {
                nMX = true; // Just one notification...
                notify (QGAMES::Event (_QGAMES_MOVEMENTNOTPOSSIBLE_, this));
                stopDuringMovement (vX, _data -> _nextAcceleration, e);
            }
        }

        if (err_2 > 0)
        {
            err_2 -= dy2;
            if (mZ && (mZ = e -> canMove (vZ, _data -> _nextAcceleration)))
                e -> setPosition (e -> position () + vZ);
        }
    }
}

```

```

        else if (!nMZ)
        {
            nMZ = true;
            notify (QGAMES::Event (__QGAMES_MOVEMENTNOTPOSSIBLE__, this));
            stopDuringMovement (vZ, _data -> _nextAcceleration, e);
        }
    }

    err_1 += dx2; err_2 += dz2;
    if (mY && (mY = e -> canMove (vY, _data -> _nextAcceleration)))
        e -> setPosition (e -> position () + vY);
    else if (!nMY)
    {
        nMY = true;
        notify (QGAMES::Event (__QGAMES_MOVEMENTNOTPOSSIBLE__, this));
        stopDuringMovement (vY, _data -> _nextAcceleration, e);
    }
}
// The Z Axis is the predominat...
else
{
    err_1 = dy2 - n; err_2 = dx2 - n;
    bool mX = true; bool mY = true; bool mZ = true;
    bool nMX = false; bool nMY = false; bool nMZ = false;
    for (int i = 0; i < n && mZ; i++)
    {
        if (err_1 > 0)
        {
            err_1 -= dz2;
            if (mY && (mY = e -> canMove (vY, _data -> _nextAcceleration)))
                e -> setPosition (e -> position () + vY);
            else if (!nMY)
            {
                nMY = true; // Just one notification...
                notify (QGAMES::Event (__QGAMES_MOVEMENTNOTPOSSIBLE__, this));
                stopDuringMovement (vY, _data -> _nextAcceleration, e);
            }
        }

        if (err_2 > 0)
        {
            err_2 -= dz2;
            if (mX && (mX = e -> canMove (vX, _data -> _nextAcceleration)))
                e -> setPosition (e -> position () + vX);
            else if (!nMX)
            {
                nMX = true;
                notify (QGAMES::Event (__QGAMES_MOVEMENTNOTPOSSIBLE__, this));
                stopDuringMovement (vX, _data -> _nextAcceleration, e);
            }
        }

        err_1 += dy2; err_2 += dx2;
        if (mZ && (mZ = e -> canMove (vZ, _data -> _nextAcceleration)))
            e -> setPosition (e -> position () + vZ);
        else if (!nMZ)
        {
            nMZ = true;
            notify (QGAMES::Event (__QGAMES_MOVEMENTNOTPOSSIBLE__, this));
            stopDuringMovement (vZ, _data -> _nextAcceleration, e);
        }
    }
}

calculateNextMovement (_data, d, a, e); // ...calculate next movement
somethingElseToDoAfterMoving (d, a, e); // ...and exit to add something!
}

// Nothing else to do except prepare the entity point to the next movement...
e -> setOrientation (_data -> _nextOrientation);
}
}

```

45: Ejemplo de implementación de Bresenham en QGAMES

Y con este ejemplo, damos por finalizado esta primera versión del ensayo sobre algoritmos y matemática utilizado en juegos. Esperamos que hay sido de su interés.

IGNACIO CEA FORNIÉS

Anexos

IGNACIO CEA FORNIÉS

Ilustraciones

1: EL GLOBO. EL PRINCIPIO DE TODO	3
2: LOS MANUALES QUE LEÍ	4
3: UN LINRO MUY INTERSANTE SOBRE LIBRERÍAS GRÁFICAS	5
4: CÓDIGO POSITION. MÓDULO “INCLUDE”	16
5: CÓDIGO POSITION. MÓDULO “SRC”	18
6: CÓDIGO MATRIX	26
7: EL PLANO Y SUS VECTORES REPRESENTATIVOS.....	28
8: LA CAJA DE PROYECCIÓN.....	35
9: PROYECCIÓN CÓNICA.....	37
10: PROYECCIÓN CILÍNDRICA	38
11: PROYECCIÓN AXONOMÉTRICA	39
12: PROYECCIÓN ISOMÉTRICA.....	40
13: PROYECCIÓN CABALLERA	40
14: FORNITE®. PERSPECTIVA CÓNICA	41
15: MARIO BROSS®. PERSPECTIVA ORTOGONAL.....	41
16: CLASH OF CLANS®. PERSPECTIVA ISOMÉTRICA	42
17: ENTOMBED®. PERSPECTIVA CABALLERA	42
18: PROYECCIÓN CÓNICA. CÁLCULO EN LA PANTALLA DE PROYECCIÓN	43
19: PROYECCIÓN DE UNA RECTA EN CÓNICA CON PRINCIPIO POR DETRÁS DEL PUNTO DE VISTA.....	47
20: FUNCIÓN LAMBDA PARA RECORTAR LOS PUNTOS DE UNA LÍNEA EN PROYECCIÓN CÓNICA.....	53
21: TIPOS DE RECTAS QUE HACEN T1 O T3 NULO	54
22: FIGURA GEOMÉTRICA CON PUNTOS MÁS ALLÁ DEL PUNTO DE VISTA.....	56
23: POSIBLES PROYECCIONES DE UN TRIÁNGULO CON ALGÚN VÉRTICE MÁS ALLÁ DEL PUNTO DE VISTA.....	57
24: PROYECCIÓN ISOMÉTRICA. CALCULO EN LA PANTALLA DE PROYECCIÓN	60
25: PROYECCIÓN CABALLERA. CÁLCULO EN LA PANTALLA DE PROYECCIÓN.....	62
26: CAMBIO DE SISTEMA DE REFERENCIA	66
27: GIRO SOBRE EL EJE X (YAW)	68
28: REFERENCIA DE PROYECCIÓN EN JUEGOS.....	69
29: GIRO SOBRE EL EJE Y (PITCH).....	70
30: GIRO SOBRE EL EJE Z (ROLL)	71
31: HOJA DE FOTOGRAMAS PARA UN JUEGO ORTOGONAL	76
32: HOJA DE FOTOGRAMAS PARA UN JUEGO EN PERSPECTIVA ISOMÉTRICA.....	77
33: PARALELIPÍPEDO EN PERSPECTIVA CABALLERA (45°/35°)	78
34: COORDENADAS EN CABALLERA DEL OFFSET.....	83
35: PARALELIPÍPEDO EN PERSPECTIVA ISOMÉTRICA.....	90
36: REPRESENTACIÓN ISOMÉTRICA DE LOS PUNTOS OFFSET.....	93
37: CUADRADOS ALINEADOS, ORIENTADOS EN UN PLANO XY Y CON LOS LADOS ALINEADOS CON LOS EJES	105
38: CUADRADOS ORIENTADOS EN UN PLANO XY Y CON LOS LADOS GIRADOS CON RESPECTO A LOS EJES	107
39: CLASE RECTANGLE. DEFINICIÓN .HPP.....	111
40: CLASE RECTANGLE. DEFINICIÓN SRC	122
41: CLASE Box. DEFINICIÓN INCLUDE	125
42: CLASE Box. DEFINICIÓN SRC	127
43: PRIMERA VERSIÓN DEL ALGORITMO DE MOVIMIENTO BASADO EN BRESENHAM	140
44: ECUACIONES DE CONTROL DE MOVIMIENTO 2D BASADAS EN BRESENHAM	141
45: EJEMPLO DE IMPLEMENTACIÓN DE BRESENHAM EN QGAMES.....	145

Ecuaciones

1: COORDENADAS DEL VECTOR QUE UNE DOS PUNTOS	10
2: ECUACIÓN BÁSICA DE MOVIMIENTO DE UN PUNTO EN EL ESPACIO	11
3: PRODUCTO ESCALAR ENTRE DOS VECTORES. DEFINICIÓN BÁSICA.....	12
4: PRODUCTO ESCALAR ENTRE DOS VECTORES. DEFINICIÓN ALTERNATIVA	12
5: CÁLCULO RÁPIDO DEL MÓDULO AL CUADRADO DE UN VECTOR USANDO EL PRODUCTO ESCALAR.....	12
6: PRODUCTO VECTORIAL ENTRE DOS VECTORES. DEFINICIÓN BÁSICA.....	13
7: PRODUCTO VECTORIAL ENTRE DOS VECTORES. DEFINICIÓN ALTERNATIVA	13
8: MÓDULO DE LA PROYECCIÓN DE UN VECTOR SOBRE OTRO, USANDO EL PRODUCTO ESCALAR	14
9: ECUACIÓN GENERAL DEL PLANO	27
10: ECUACIÓN GENERAL DE LA RECTA.....	28
11: PUNTO DE INTERSECCIÓN DE PLANO CON UNA RECTA PERPENDICULAR A ÉL.....	29
12: ROTACIÓN DE UN PUNTO B ALREDEDOR DE UN EJE DETERMINADO POR EL VECTOR N Y EL PUNTO A	30
13: PROYECCIÓN DE UN PUNTO EN UN PLANO.....	30
14: DISTANCIA DE UN PUNTO A UNA RECTA	31
15: COORDENADAS DEL PUNTO DE CORTE ENTRE UNA RECTA Y UN PLANO	32
16: COORDENADAS DEL PUNTO PROYECTADO EN PERSPECTIVA CÓNICA SOBRE UNA PANTALLA CUADRADA	44
17: COORDENADAS DEL PUNTO PROYECTADO EN PERSPECTIVA CÓNICA SOBRE UNA PANTALLA PANORÁMICA	45
18: COORDENADAS DEL PUNTO PROYECTADO EN PERSPECTIVA CÓNICA SOBRE UNA PANTALLA DE CUALQUIER RELACIÓN DE ASPECTO	45
19: ALGORITMO PARA LIMITAR LOS PUNTOS FUERA DE LA ZONA DE VISIÓN EN UNA PROYECCIÓN CÓNICA.....	52
20: VALORES DE T PARA LIMITAR LOS PUNTOS DE UNA RECTA MÁS ALLÁ DEL PUNTO DE VISTA EN UNA PROYECCIÓN CÓNICA..	55
21: COORDENADAS DEL PUNTO PROYECTADO EN PERSPECTIVA AXONOMÉTRICA ISOMÉTRICA.....	61
22: COORDENADAS PROYECTADAS DE UN PUNTO EN PERSPECTIVA AXONOMÉTRICA CABALLEA	62
23: COORDENADAS UNIVERSALES DE UN PUNTO A PARTIR DE LAS MISMAS RESPECTO A OTRO SISTEMA DE REFERENCIA	67
24: COORDENADAS DE UN PUNTO EN UN SISTEMA DE REFERENCIA A PARTIR DE SUS EQUIVALENTES EN UN SISTEMA UNIVERSAL	67
25: ECUACIÓN DE CÁLCULO DE UN PUNTO CUALQUIERA REFERIDO AL SISTEMA DE PROYECCIÓN V	72
26: MATRICES DE GIRO DE UN PUNTO CUALQUIERA REFERIDO AL SISTEMA DE PROYECCIÓN	72
27: DIMENSIONES VISUALES APROXIMADAS GENERALES (k) DE UN FOTOGRAMA EN PERSPECTIVA CABALLERA 30 GRADOS....	81
28: VERIFICACIÓN A REALIZAR PARA VALORES GENÉRICOS DE K ENTRE LAS DIMENSIONES DEL FOTOGRAMA EN CABALLERA....	81
29: DIMENSIONES VISUALES APROXIMADAS (p) GENERALES DE UN FOTOGRAMA EN PERSPECTIVA CABALLERA 30 GRADOS....	81
30: CUADRO DE DIMENSIONES PARA DIFERENTES VALORES DE P EN PERSPECTIVA CABALLERA.....	82
31: VALORES MÍNIMOS DE LA DIMENSIÓN VERTICAL SEGÚN DIFERENTES VALORES DE P EN PERSPECTIVA CABALLERA.....	82
32: PUNTOS CARACTERÍSTICOS DEL PARALELEPÍPEDO QUE ENVUELVE A UN FOTOGRAMA EN CABALLERA EN REPRESENTACIÓN 3D.....	84
33: PUNTOS CARACTERÍSTICOS 2D DE UN FOTOGRAMA EN PERSPECTIVA CABALLERA	85
34: ECUACIONES DE LAS DIMENSIONES DE UN FOTOGRAMA EN PERSPECTIVA CABALLERA 30 GRADOS RÁPIDA	86
35: ECUACIONES DE LAS DIMENSIONES VISUALES DE UN FOTOGRAMA EN PERSPECTIVA CABALLERA 30 GRADOS RAPIDA EN BASE A P	86
36: ECUACIONES DE LAS DIMENSIONES DE UN FOTOGRAMA EN PERSPECTIVA ISOMÉTRICA APROXIMADA.....	92
37: PUNTOS CARACTERÍSTICOS DEL PARALELEPÍPEDO QUE ENVUELVE A UN FOTOGRAMA EN ISOMÉTRICA EN REPRESENTACIÓN 3D.....	95
38: PUNTOS CARACTERÍSTICOS 2D DE UN FOTOGRAMA EN PERSPECTIVA ISOMÉTRICA	95
39: ECUACIONES DE LAS DIMENSIONES VISUALES DE UN FOTOGRAMA EN PERSPECTIVA ISOMÉTRICA RÁPIDA	96
40: ECUACIONES DE LAS DIMENSIONES VISUALES DE UN FOTOGRAMA EN PERSPECTIVA ISOMÉTRICA RÁPIDA EN BASE A P	96
41: ECUACIÓN PARA DECIDIR QUÉ ELEMENTO ESTÁ DETRÁS DE OTRO EN UNA PROYECCIÓN ISOMÉTRICA	99
42: PUNTO MEDIO DE UN RECTÁNGULO.....	104
43: VERIFICAR SI UN PUNTO ESTÁ DENTRO DE UN RECTÁNGULO ORIENTADO, PARALELO AL PLANO XY Y ALINEADO CON SUS EJES	104
44: VERIFICAR SI DOS RECTÁNGULOS ORIENTADOS, PARALELOS AL PLANO XY Y ALINEADOS CON SUS EJES SE CORTAN	105

45: INTERSECCIÓN DE 2 RECTÁNGULOS EN EL MISMO PLANO XY, ORIENTADOS Y CON LOS LADOS ALINEADOS CON LOS EJES	105
46: ECUACIÓN PARA VERIFICAR SI UN PUNTO P PERTENECE A UN RECTÁNGULO	106
47: ECUACIONES GENERALES DE MOVIMIENTO CONTINUAS (EN FUNCIÓN DE T).....	130
48: ECUACIONES GENERALES DE MOVIMIENTO DISCRETAS (EN FUNCIÓN DE FRACCIONES DE FPS)	131
49: VALORES QUE CALCULAR ANTES DE INICIAR CUALQUIER MOVIMIENTO O AL VARIAR CUALQUIERA LAS FUERZAS QUE ACTÚAN	133
50: VALORES QUE CALCULAR EN CADA UNO DE LOS BUCLES DE UN JUEGO ASOCIADO A LOS MOVIMIENTOS	133

IGNACIO CEA FORNÍES