CUBOS & SPHERES

IMPLEMENTACION:

En cuanto a los requisitos, el juego es completo, y ha sido implementad con el motor visto en clase. Incluye una pantalla de inicio y posee la funcionalidad de guardar y cargar partidas. Es posible (aunque a veces no recomendable) cambiar la orientación de la cámara y por tanto las escena. Emplea OpenCV para introducir texturas.

A parte de estos requisitos este juego posee otros aspectos como:

-Un control de movimiento fluido

-Una completa implementación de quaterniones y rotaciones con estos.

-Un sistema de colisiones entre cubos y esferas físicamente realista y exacto.

CONTROL FLUIDO:

Usando un array de booleanos de 256 posiciones, cada vez que se presiona una tecla con la función glutKeyPressedFunc(), la posición correspondiente al código ASCI de la letra se cambia a true. Al soltar dicha tecla con la función glutKeyUpFunc() el valor es devuelto a false. En el idle se comprueba las posiciones ASCI adecuadas (por ejemplo las teclas WASD de movimiento) y se establece el movimiento/ velocidad deseado.

Usando la función `glutIgnoreKeyRepeat(1)` se evita la repetición de comando asegurando una fluidez de movimiento aun mejor.

QUATERNIONES:

Empleando una clase nueva (Quaternion.h) semejante a Vector3D se sustituyen todas las rotaciones tanto de primitivas como de vectores o puntos por rotaciones de quaterniones.

La clase puede transformarse libremente de vector3D a quaternion asegurando que se pueda usar en cualquier situación.

Para rotar primitivas (cubos, esferas…) primero, el atributo de rotación de sólido es cambiado de un Vector3D a un Quaternion. Con un método que lo transforma en matriz este se multiplica a las matrices de transformación de glut y opengl rotando la primitiva cada update.

Formula obtenida de: https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation

Adicionalmente Solido dispone de otro quaternion atributo, velocidad angular. Esta lo rota cada update, es decir cambia su rotación multiplicándola cada vez por el quaternion de velocidad angular.

Para rotar vectores, como ya he mencionado antes, se transforma a quaternion (si no lo és) y se multiplica por otro quaternion de rotación de manera peculiar: quatRot * quatVect * quatRot.inv(). Es necesario aplicar la inversa también para evitar errores “extraños”.

Formulas multiplicación: https://en.wikipedia.org/wiki/Quaternion

COLISIONES:

Empleando la siguiente fórmula:

$$|J| = (e+1) * (v_{ai} - v_{bi}) / (1/Ma + n \bullet ([Ia]^{-1}(n \times ra)) \times ra + 1/Mb + n \bullet ([Ib]^{-1}(n \times rb)) \times rb)$$

Se averigua el impulso generado por una colisión, el cual luego se usa para averiguar las velocidades resultantes.

-Vai,vbi son velocidades de ambos cuerpos.

-Ia, Ib se refieren al matrix tensor de cada objeto. Una matriz que depende de la forma de la primitiva en cuestión y afecta la velocidad y la dirección del rebote

-n es la normal del punto de choque (como regla general n apunta siempre hacia el segundo cuerpo). En el caso de esferas, la normal es simplemente un vector que sale de la posición de la primera y acaba en la posición de la segunda

-ra y rb son vectores que unen en punto de colisión con el centro de su respectivo cuerpo.

-e es el coeficiente de elasticidad usad en el cálculo del impulso.

Para la formula se tienen en cuenta 3 parámetros de Solido adicionales:

> -elasR: en este case “e”, es el coeficiente de elasticidad. Para calcular la colisión es posible coger la media de ambos “e” de cada objeto o usar el mínimo de los dos.
>
> -IMT: Inverse Matrix Tensor: una matriz 3x3 correspondiente a Ia,Ib. Se invierte ya que solo se pide la inversa en la formula.
>
> -colMask: un array de información de la mascara: la primera posición determina si el objeto participa en las colisiones, la segunda que tipo es (esfera, cubo, prisma…), la tercera determina su tamaño, su radio, su S (si se amplia en el futuro el motor a otros objetos habrá que expandirlo). Por ultimo su cuarta posición determina aspectos de la puntuación del juego (si se ha chocado un cubo rojo, una esfera amarilla…)

Una vez aplicada la formula se calcula cada velocidad de las siguiente manera:

$$v_{af} = v_{ai} - J/Ma$$
$$v_{bf} = v_{bi} + J/Mb$$

Adicionalmente la velocidad angular resultante se puede averiguar calculando el producto vectorial de n (la normal de la colision) y ra ò rb (vectores que van del centro de cada objeto al punto de colisión)

Formulas: http://www.euclideanspace.com/physics/dynamics/collision/threed/index.htm

http://www.euclideanspace.com/physics/dynamics/collision/index.htm

Por último se tiene unas consideraciones especiales al calcular el choque entre cubo y esfera:

Usando la función clamp, se saca el punto mas cercano de cubo respecto al centro de la esfera (Ejemplo para valor X del punto: clamp(-cubo.s , esfera.x, cubo.s), dependiendo de la posición de la esfera se obtendrá un valor u otro, juntando los valores X Y Z se obtiene dicho punto). Una vez obtenido se va comprobando su distancia a la esfera.

Método sacado de Libro: "Collision Detection in Interactive 3D Environments"

3.2.2 Sphere-Box Test

We conclude this section with a discussion of how a sphere and an axis-aligned box are tested for intersection. This is done by computing the point in the box closest to the sphere's center, and testing whether this point is contained in the sphere. Let the box be centered at the origin with extent $\mathbf{h} = (\eta_1, \eta_2, \eta_3)$, and let the sphere's center be given by $\mathbf{c} = (\gamma_1, \gamma_2, \gamma_3)$. Then, the point in the box closest to $\mathbf{c}$ is the point $\mathbf{x} = (\text{clamp}(\gamma_1, -\eta_1, \eta_1), \text{clamp}(\gamma_2, -\eta_2, \eta_2), \text{clamp}(\gamma_3, -\eta_3, \eta_3))$, where

$$\text{clamp}(\gamma, \alpha, \beta) = \begin{cases} \alpha \text{ if } \gamma < \alpha \\ \beta \text{ if } \gamma > \beta \\ \gamma \text{ otherwise.} \end{cases}$$

The sphere intersects the box iff the point $\mathbf{x}$ is contained in the sphere; that is, the squared distance $\|\mathbf{x} - \mathbf{c}\|^2$ is at most the squared radius ρ^2.

The witness points for a nonzero distance are found in the following way. We take the point $\mathbf{x}$ closest to the sphere's center as the witness point on the boundary of the box. The witness point on the sphere's boundary is the point

$$\mathbf{r} = \mathbf{c} + \rho \frac{\mathbf{v}}{\|\mathbf{v}\|},$$

where $\mathbf{v} = \mathbf{x} - \mathbf{c}$. The distance between the sphere and the box is $\|\mathbf{x} - \mathbf{r}\| = \|\mathbf{v}\| - \rho$—that is, if the sphere and the box do not intersect ($\|\mathbf{v}\| > \rho$), since otherwise the distance is zero.

If the sphere and the box intersect and the center of the sphere is not contained by the box, then the points $\mathbf{r}$ and $\mathbf{x}$ are witness points of a

Otra consideración a tener en cuenta es que la normal del choque entre cubo y esfera, corresponde con la normal de la cara donde se da el punto de colisión, resultando en un rebote muy distinto.

Esto no se cumple siempre, a veces se da una colisión esfera-esquina(del cubo), la normal de dicha colisión se averigua de la misma manera que entre esfera-esfera (un vector que une los centros de los dos cuerpos)

*En cuanto a las colisiones entre cubo y cubo, no son exactas. Se asume que los cubos son esferas de R=S (S = lado del cubo).

OTROS:

-La pantalla de inicio del juego es un rectángulo con una textura preestablecida situado frente a la cámara, este deja de renderizarse al presionar espacio.

-La orientación de la vista hace uso de las funciones de ratón de opengl para calcular un vector de rotación adecuado, aplicándolo a la cámara TPS.

-Adicionalmente usando un arrray de vistas, el juego tiene dos cámaras en las misma pantalla una TPS y otra Fly.

-Por último es posible guardar la puntuación usando un fichero de texto y cargarla mediante el mismo fichero, fijándose en que línea se ha guardado cada cacho de info.