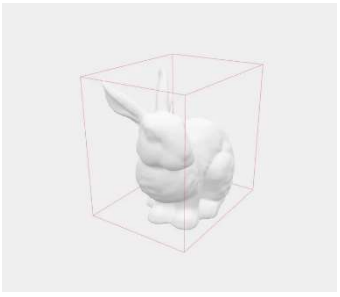
Solid Objects Physics

- **Definición de objetos sólidos:**

```
// Componentes
struct TransformComponent {
    float x, y;

    TransformComponent(float x, float y) : x(x), y(y) {}
};

struct VelocityComponent {
    float vx, vy, vz;

    VelocityComponent(float vx, float vy, float vz) : vx(vx), vy(vy), vz(vz) {}
};
struct MassComponent {
    float mass;

    MassComponent(float mass) : mass(mass) {}
};

struct CollisionComponent {
    float width, height, depth;

    CollisionComponent(float width, float height, float depth) : width(width), height(height),
depth(depth) {}
};
// Entity
class SolidObject {
public:
    TransformComponent transform;
    VelocityComponent velocity;
    MassComponent mass;
    CollisionComponent collision;

    SolidObject(float x, float y, float mass) : transform(x, y), velocity(0.0f, 0.0f), mass(mass) {}

    void applyGravity(float gravity) {
        velocity.vy -= gravity;
    }

    // Método Euler para actualizar la posición y velocidad
    void update(float deltaTime) {
        transform.x += velocity.vx * deltaTime;
        transform.y += velocity.vy * deltaTime;
    }

    void applyForce(float forceX, float forceY) {
        float ax = forceX / mass.mass;
```

```
        float ay = forceY / mass.mass;
        velocity.vx += ax;
        velocity.vy += ay;
    }

    void applyFriction(float frictionCoefficient) {
        float frictionX = -frictionCoefficient * velocity.vx;
        float frictionY = -frictionCoefficient * velocity.vy;
        applyForce(frictionX, frictionY);
    }

bool collisionRectToRect(const SolidObject& rectA, const SolidObject& rectB) {
    float thisLeft = rectA.transform.x - rectA.collision.width / 2;
    float thisRight = rectA.transform.x + rectA.collision.width / 2;
    float thisTop = rectA.transform.y + rectA.collision.height / 2;
    float thisBottom = rectA.transform.y - rectA.collision.height / 2;
    float thisFront = rectA.transform.z + rectA.collision.depth / 2;  // Nueva dimensión

    float otherLeft = rectB.transform.x - rectB.collision.width / 2;
    float otherRight = rectB.transform.x + rectB.collision.width / 2;
    float otherTop = rectB.transform.y + rectB.collision.height / 2;
    float otherBottom = rectB.transform.y - rectB.collision.height / 2;
    float otherFront = rectB.transform.z + rectB.collision.depth / 2;  // Nueva dimensión

    // Verificar la colisión en tres dimensiones
    return (thisLeft < otherRight && thisRight > otherLeft &&
            thisTop > otherBottom && thisBottom < otherTop &&
            thisFront > otherBottom && thisBottom < otherFront);
}

bool isColliding(const SolidObject& other) {

        if (collisionRectToRect(*this, other) || collisionSphereToRect(*this, other) {
            return true;
        }
};
```



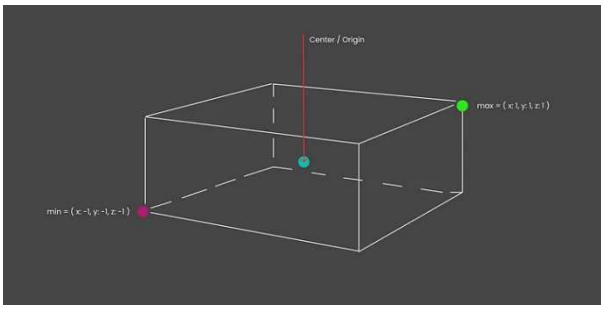
Un collision box es un cuboide que contiene el objeto geométrico y que define la extensión espacial de éste.

La caja actúa como un contenedor invisible que envuelve toda la geometría, proporcionando una forma rápida y eficaz de determinar su tamaño, posición y detección de colisiones.

La geometría es una colección de vértices, aristas y caras que representan un objeto tridimensional. Estas geometrías pueden ser tan simples como un cubo o tan complejas como un modelo 3D detallado de un personaje o un entorno. Independientemente de su complejidad, cada geometría puede describirse como un conjunto de puntos en el espacio 3D y, a partir de estos datos, podemos calcular su collision box.

1. AABB (Cuadro Delimitador Alineado con los Ejes):
    - *Características:*
        - Se alinea con los ejes del sistema de coordenadas.
        - Fácil de calcular e implementar.
        - Más eficiente en términos de rendimiento para cálculos de intersección.
    - *Ventajas:*
        - Simplifica las operaciones de colisión y detección de intersecciones.
        - Cálculos más rápidos debido a la alineación simple.
2. OBB (Cuadro Delimitador Orientado):
    - *Características:*
        - No necesariamente alineado con los ejes; puede tener cualquier orientación.
        - Más complejo de calcular y gestionar que un AABB.
        - Proporciona una representación más precisa de la geometría del objeto.
    - *Ventajas:*
        - Permite representar mejor la forma real de objetos con orientaciones no alineadas.
        - Útil en situaciones donde la rotación de objetos es fundamental.

Nosotros implementaremos sistema de cuadro delimitador alineado con los ejes (AABB) para aligerar la complejidad tanto computacional como al cálculo de colisiones.



el punto mínimo → mínimo_x, mínimo_y, mínimo_z

el punto máximo → máximo_x, máximo_y, máximo_z

Para calcular la caja delimitadora de una geometría dada, necesitamos procesar sus vértices y encontrar las coordenadas mínimas y máximas a lo largo de cada eje (x, y, z). Para ello necesitamos

1. Analizar los datos de la geometría: Trabajando con modelos 3D desde formatos de archivo como OBJ, necesitaremos extraer los vértices.
2. Encontrar las coordenadas mínimas y máximas: Recorreremos los vértices y encontramos las coordenadas mínimas y máximas a lo largo de los ejes x, y, y z.
3. Construir el collision box: Con las coordenadas mínimas y máximas, construimos la caja delimitadora definiendo las dos esquinas.

Componentes:

TransformComponent: Representa la posición en el plano 2D de un objeto. Toma dos valores x y y para definir la posición del objeto en el espacio. Esto se utiliza para rastrear la ubicación de un objeto en el mundo.

VelocityComponent: Representa la velocidad de un objeto en el plano 2D. Almacena las velocidades en las direcciones vx (horizontal) y vy (vertical). Este componente se utiliza para controlar el movimiento de un objeto.

MassComponent: Almacena la masa del objeto. La masa es un valor escalar que se utiliza en cálculos de física para simular el movimiento y las fuerzas que actúan sobre el objeto.

CollisionComponent: Define la forma de colisión del objeto y su tamaño. En este caso, asumimos que los objetos tienen una forma rectangular y se almacena el ancho (width) y alto (height) del rectángulo.

Entity SolidObject: Esta es la entidad principal que representa un objeto sólido en el juego y agrega componentes que describen su estado físico y colisiones. Está por definir si la pelota entra dentro de solidObject por sus características propias como el colisión box circular.

Constructor: la posición inicial (x y y) y la masa del objeto. La velocidad se inicia en cero por defecto.

applyGravity: aplicar una fuerza de gravedad al objeto. Reduce la velocidad vertical (vy) del objeto para simular la aceleración hacia abajo debido a la gravedad.

update: Implementa el método de Euler para actualizar la posición y velocidad del objeto en función del tiempo (deltaTime). Calcula la nueva posición en función de la velocidad actual.

applyForce: Aplica una fuerza horizontal (forceX) y vertical (forceY) al objeto, lo que provoca un cambio en la velocidad del objeto en función de su masa.

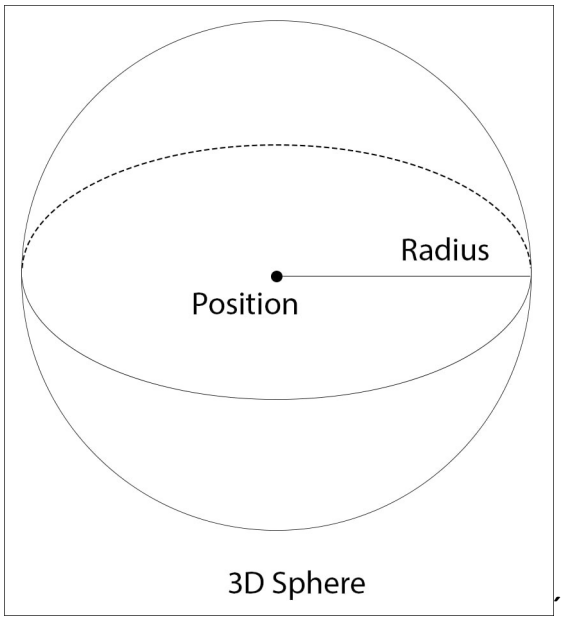
applyFriction: Simula la fricción al aplicar una fuerza de fricción opuesta a la dirección de movimiento. Esto reduce la velocidad del objeto con el tiempo.

isColliding: Detectar si el objeto actual está colisionando con otro objeto (other). Se calculan las coordenadas de los límites de colisión de ambos objetos y se verifica si se superponen. Si los límites se superponen en ambas dimensiones (horizontal y vertical), se considera que hay una colisión.

SphereCollision:

```
struct SphereCollisionComponent {
    float radius;
    float centerX;
    float centerY;
    float centerZ;

    SphereCollisionComponent(float radius, float centerX, float centerY, float centerZ)
        : radius(radius), centerX(centerX), centerY(centerY), centerZ(centerZ) {}
};

class SolidObject {
public:
    // ... ...

    SphereCollisionComponent sphereCollision;

    SolidObject(float x, float y, float z, float mass, float radius)
        : transform(x, y, z), velocity(0.0f, 0.0f, 0.0f), mass(mass), collision(2 * radius, 2 * radius),
sphereCollision(radius, x, y, z) {}

  // ...

private:
    // ...  ...

    bool collisionSphereToRect(const SolidObject& sphere, const SolidObject& rect) {
        float circleX = sphere.transform.x;
        float circleY = sphere.transform.y;
        float circleZ = sphere.transform.z;
        float rectX = rect.transform.x;
        float rectY = rect.transform.y;
        float rectZ = rect.transform.z;

        float closestX = clamp(circleX, rectX - rect.collision.width / 2, rectX + rect.collision.width /
2);
        float closestY = clamp(circleY, rectY - rect.collision.height / 2, rectY + rect.collision.height
/ 2);
        float closestZ = clamp(circleZ, rectZ - rect.collision.depth / 2, rectZ + rect.collision.depth /
2);

        float distanceX = circleX - closestX;
        float distanceY = circleY - closestY;
        float distanceZ = circleZ - closestZ;
```

```
        float distanceSquared = distanceX * distanceX + distanceY * distanceY + distanceZ *
distanceZ;

        float combinedRadius = sphere.sphereCollision.radius + rect.collision.width / 2;
        return distanceSquared < (combinedRadius * combinedRadius);
    }

    float clamp(float value, float min, float max) {
        return std::max(min, std::min(value, max));
    }
};
```



¿Qué es una esfera? punto + radio

El hitbox consiste en un punto en el centro de la esfera y un radio que define su tamaño.

Almacenamos el centro (x, y, z) de la esfera y su radio. Luego, para detectar colisiones, verifica si la distancia entre el centro de la esfera y el centro de otro objeto es menor que la suma de sus radios.

¿Para qué vamos a implementar colisiones de esfera? Para la pelota de fútbol.

componente SphereCollisionComponent:

- Se creó una nueva estructura llamada **SphereCollisionComponent** para representar la colisión de una esfera.
- Este componente tiene un único atributo **radius** que almacena el radio de la esfera.

método collisionSphere:

- Se agregó un nuevo método llamado collisionSphere para verificar la colisión entre una esfera y un rectángulo.
- Este método utiliza la lógica de distancia entre el centro de la esfera y el punto más cercano en el rectángulo para determinar si hay colisión.

Bibliografía: https://medium.com/@egimata/understanding-and-creating-the-bounding-box-of-a-geometry-d6358a9f7121

- Integración numérica:

El método de Euler es un método numérico para aproximar el movimiento de un objeto en un campo de fuerzas. Se utiliza para predecir el comportamiento de objetos en movimiento bajo la influencia de fuerzas en incrementos de tiempo discretos.

- **Elección del intervalo de tiempo (deltaTime)**
- **Aplicación de la gravedad**
- **Actualización de la posición y la velocidad**

Nueva Velocidad = Velocidad anterior + (Aceleración * deltaTime)

Nueva Posición = Posición anterior + (Velocidad * deltaTime)

- Fuerzas y colisiones:

Gravedad: $F_{gravedad} = m \cdot g$ (m = masa, g=gravedad) (Segunda Ley Newton F=ma)

Velocidad: $v = u + at$ (v=velocidad final, u=velocidad inicial, a=aceleración, t=tiempo.)

- Interacción con el entorno:

- Actualización del juego:

- Optimización:

- Depuración y ajuste:

- Documentación y comentarios:

- Pruebas y ajustes: