Computación Gráfica

Clase 2 - Álgebra, Primitivas, Colisiones

Vectores

- Representación en código
- Magnitud
- Normalización
- Producto escalar
- Producto vectorial

Vectores - Representación

3 floats / doubles, uno por eje.

Implementación de referencia en

`javax.vecmath.Vector3d`

`javax.vecmath.Vector3f`

Vectores - Magnitud

$$|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

Vector normalizado / unitario:

$$|\vec{v}| = 1$$

Notación: $\hat{v} \iff |\vec{v}| = 1$

Normalización:

$$\vec{u} = \frac{\vec{v}}{|\vec{v}|}$$

Tip: mantener la inversa de la normal precomputada ayuda, ya que a veces querremos un vector unitario y otras no

Vectores - Producto Escalar

Se define como:

$$\vec{v} \cdot \vec{u} = |\vec{v}||\vec{u}| \cos\theta$$

Si son perpendiculares, siempre es 0

También computable como:

$$\vec{v} \cdot \vec{u} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix} \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = v_x u_x + v_y u_y + v_z u_z$$

Vectores - Producto Escalar

Corolario:

$$|\vec{v}| = \sqrt{\vec{v} \cdot \vec{v}}$$

Nota: si v y u estan normalizados, el producto escalar es el coseno del ángulo

Vectores - Producto Escalar - SIMD

SIMD : Single Instruction Multiple Data

Busca paralelizar el cómputo entre varias ALUs.

Intel incluye instrucciones SIMD.

Por ejemplo: MULPS toma 2 vectores de 4 elementos, y multiplica los pares en cada posición

Vectores - Producto Escalar - SIMD

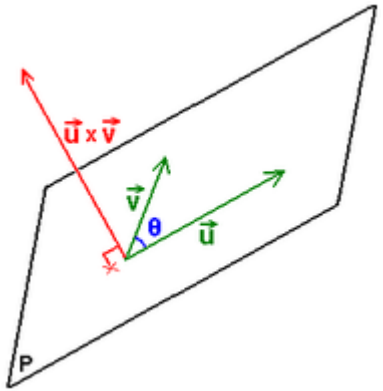
Producto escalar en SIMD:

```
000000013F8FD0E8  mulps     xmm0,xmmword ptr [13F940320h]
000000013F8FD0EF  movaps    xmm1,xmm0
000000013F8FD0F2  shufps    xmm1,xmm0,1Bh
000000013F8FD0F6  addps     xmm1,xmm0
000000013F8FD0F9  movaps    xmm0,xmm1
000000013F8FD0FC  shufps    xmm0,xmm1,4Eh
000000013F8FD100  addps     xmm0,xmm1
000000013F8FD103  movaps    xmmword ptr [rsp+50h],xmm0
```

Tenemos que definir nuestros vectores con 4 elementos, no 3.

Vectores - Producto Vectorial

$$\vec{v} \times \vec{u} = (|\vec{v}||\vec{u}| \sin\theta)\,\hat{n}$$



$$\vec{v} \times \vec{u} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ v_x & v_y & v_z \\ u_x & u_y & u_z \end{vmatrix} = (v_y u_z - v_z u_y)\mathbf{i} - (v_x u_z - v_z u_x)\mathbf{j} + (v_x u_y - v_y u_x)\mathbf{k}$$

Vectores - Producto Vectorial - SIMD

Producto vectorial en SIMD:

```
000000013F34EDAA  movaps    xmm6,xmm2
000000013F34EDAD  shufps    xmm6,xmm2,12h
000000013F34EDB1  movaps    xmm0,xmm1
000000013F34EDB4  shufps    xmm0,xmm1,9
000000013F34EDB8  mulps     xmm6,xmm0
000000013F34EDBB  shufps    xmm1,xmm1,12h
000000013F34EDBF  shufps    xmm2,xmm2,9
000000013F34EDC3  mulps     xmm2,xmm1
000000013F34EDC6  subps     xmm6,xmm2
```

Matrices

- Representación en código
- Multiplicaciones
- Inversión y transposición
- Operaciones con vectores

Matrices - Representación

Lo intuitivo es usar jagged arrays:

```
final float matrix[3][3];
```

No hacerlo! dereferenciar es costoso!

Implementación de referencia en

`javax.vecmath.Matrix3d`

`javax.vecmath.Matrix3f`

Matrices - Multiplicación

$$A \cdot B = C \iff c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}$$

Es un producto escalar entre un vector fila y un vector columna

$$A \cdot \vec{v} = \vec{u} \iff u_i = \sum_{k=1}^{m} a_{ik} v_k$$

Matrices - Identidad

Matriz diagonal, con todos sus valores en 1.

La matriz identidad es el elemento neutro de la multiplicación matricial.

Matrices - Inversión & Transposición

Transpuesta: $B = A^T \iff b_{ij} = a_{ji}$

Inversa: $A \cdot A^{-1} = I$

Cómputo de inversa: determinantes o método de Gauss

Matrices - Inversión & Transposición

Para matrices de tamaño fijo, es mejor hardcodear las operaciones y evitar JMPs!

```
if (determinant != 0) {
    double mult = 1.0 / determinant;

    // First row
    out.values[0][0] = mult
            * (values[1][1] * values[2][2] * values[3][3]
                    + values[1][2] * values[2][3] * values[3][1]
                    + values[1][3] * values[2][1] * values[3][2]

                    - values[1][1] * values[2][3] * values[3][2]
                    - values[1][2] * values[2][1] * values[3][3] - values[1][3]
                    * values[2][2] * values[3][1]);

    out.values[0][1] = mult
            * (values[0][1] * values[2][3] * values[3][2]
                    + values[0][2] * values[2][1] * values[3][3]
                    + values[0][3] * values[2][2] * values[3][1]

                    - values[0][1] * values[2][2] * values[3][3]
                    - values[0][2] * values[2][3] * values[3][1] - values[0][3]
                    * values[2][1] * values[3][2]);

    out.values[0][2] = mult
            * (values[0][1] * values[1][2] * values[3][3]
                    + values[0][2] * values[1][3] * values[3][1]
                    + values[0][3] * values[1][1] * values[3][2]
```

```
        * m.values[1][0] + values[0][2] * m.values[2][0] + values[0][3]
        * m.values[3][0];
out.values[0][1] = values[0][0] * m.values[0][1] + values[0][1]
        * m.values[1][1] + values[0][2] * m.values[2][1] + values[0][3]
        * m.values[3][1];
out.values[0][2] = values[0][0] * m.values[0][2] + values[0][1]
        * m.values[1][2] + values[0][2] * m.values[2][2] + values[0][3]
        * m.values[3][2];
out.values[0][3] = values[0][0] * m.values[0][3] + values[0][1]
        * m.values[1][3] + values[0][2] * m.values[2][3] + values[0][3]
        * m.values[3][3];
out.values[1][0] = values[1][0] * m.values[0][0] + values[1][1]
        * m.values[1][0] + values[1][2] * m.values[2][0] + values[1][3]
        * m.values[3][0];
out.values[1][1] = values[1][0] * m.values[0][1] + values[1][1]
        * m.values[1][1] + values[1][2] * m.values[2][1] + values[1][3]
        * m.values[3][1];
out.values[1][2] = values[1][0] * m.values[0][2] + values[1][1]
        * m.values[1][2] + values[1][2] * m.values[2][2] + values[1][3]
        * m.values[3][2];
out.values[1][3] = values[1][0] * m.values[0][3] + values[1][1]
        * m.values[1][3] + values[1][2] * m.values[2][3] + values[1][3]
        * m.values[3][3];
out.values[2][0] = values[2][0] * m.values[0][0] + values[2][1]
        * m.values[1][0] + values[2][2] * m.values[2][0] + values[2][3]
        * m.values[3][0];
out.values[2][1] = values[2][0] * m.values[0][1] + values[2][1]
        * m.values[1][1] + values[2][2] * m.values[2][1] + values[2][3]
        * m.values[3][1];
out.values[2][2] = values[2][0] * m.values[0][2] + values[2][1]
        * m.values[1][2] + values[2][2] * m.values[2][2] + values[2][3]
        * m.values[3][2];
out.values[2][3] = values[2][0] * m.values[0][3] + values[2][1]
        * m.values[1][3] + values[2][2] * m.values[2][3] + values[2][3]
        * m.values[3][3];
out.values[3][0] = values[3][0] * m.values[0][0] + values[3][1]
        * m.values[1][0] + values[3][2] * m.values[2][0] + values[3][3]
        * m.values[3][0];
out.values[3][1] = values[3][0] * m.values[0][1] + values[3][1]
        * m.values[1][1] + values[3][2] * m.values[2][1] + values[3][3]
        * m.values[3][1];
out.values[3][2] = values[3][0] * m.values[0][2] + values[3][1]
        * m.values[1][2] + values[3][2] * m.values[2][2] + values[3][3]
        * m.values[3][2];
```

Transformaciones

- Rotar
- Escalar
- Trasladar
- Orden de transformaciones
- Espacios de transformaciones

Rotar

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Escalar

$$R_s = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

Trasladar

$$A \cdot \vec{v} = \vec{u} \iff u_i = \sum_{k=1}^{m} a_{ik} v_k$$

¿?

$$A \cdot \vec{v} = \vec{u} \iff u_i = \sum_{k=1}^{m} a_{ik} v_k + t_i$$

Trasladar

Agregar una dimensión a matrices y vectores!

$$R_t = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \vec{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix} \quad R_t \cdot \vec{v} = \begin{bmatrix} v_x + t_x \\ v_y + t_y \\ v_z + t_z \\ 1 \end{bmatrix}$$

Que SIMD use vectores de 4 elementos no es casual!

Implementaciones de referencia

```
javax.vecmath.Matrix4d
javax.vecmath.Matrix4f
javax.vecmath.Vector4d
javax.vecmath.Vector4f
```

Composición de transformaciones

Se multiplican matrices para componerlas.

Se evalúan de derecha a izquierda.

Es importante el orden para tener los resultados esperados:

$$T \cdot R \cdot S \cdot \vec{v}$$

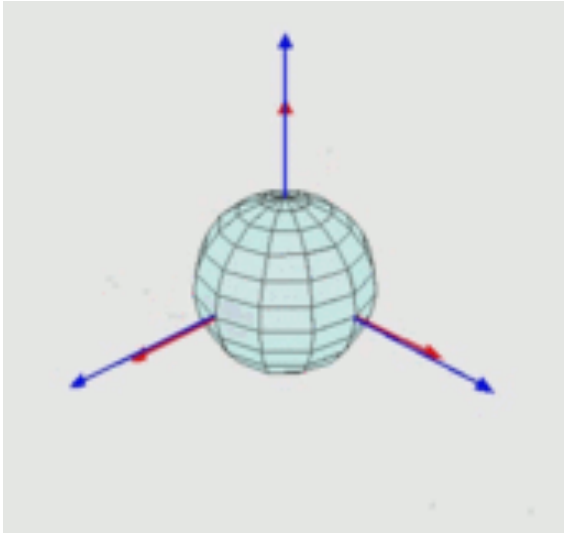
Rotaciones

- Representación Euleriana
- Gimbal lock
- Cuaterniones

Representación Euleriana

Se usan 3 ángulos, uno por eje: yaw, pitch y roll. Un eje rota y los otros quedan fijos.

$$R_z(\psi) \cdot R_y(\theta) \cdot R_x(\phi)$$



Gimbal Lock

Puede terminar en la pérdida de un grado de libertad...

Paso en Apollo 11…

"How about sending me a fourth gimbal for Christmas?"

- *Mike Collins, Lunar Module Pilot*

Gimbal Lock - Soluciones

1. sumar un cuarto gimbal teórico
2. resetear los gimbals al detectar un lock
3. reordenar las rotaciones en los 3 ejes para dejar en el medio la que sea menos común llegue a 90° (patch)
4. Cuaterniones aka versores

Cuaterniones

Se define la rotación en torno de un vector unitario

$$\hat{u} = u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k}$$

$$q = (\mathbf{s}, \vec{v}) = \cos\frac{\theta}{2} + \hat{u}\sin\frac{\theta}{2}$$

$$p' = qpq^{-1}$$

Producto Hamiltoniano….

Cuaterniones

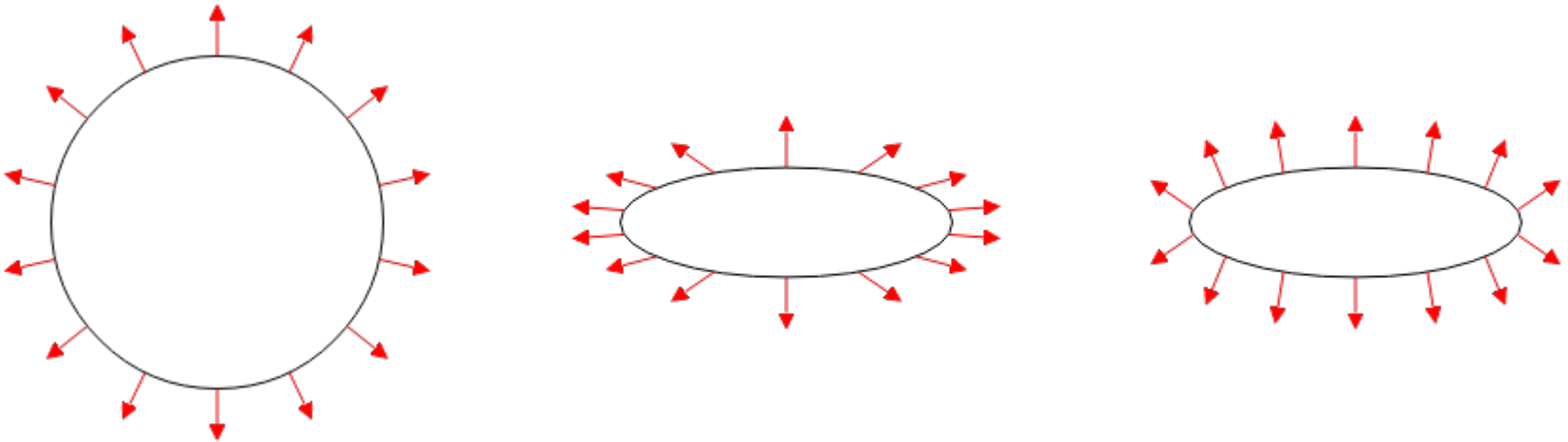
- Requieren un cuarto de la memoria
- Computacionalmente más eficientes
- Pero más complejos de entender y visualizar

Espacios de transformación

- Cada objeto tiene su transformación TRS (Translate, Rotate, Scale)
- En caso de que los objetos tengan jerarquía, estas transformaciones se componen.
- Al calcular las intersecciones, se transforma el rayo a espacio local de cada objeto.
- Luego, se calculan las intersecciones en espacio local, en el origen, sin rotación.
- Conociendo la distancia de la intersección al origen del rayo, puede calcularse el punto de intersección en espacio global.
- En caso de las normales y tangentes, debe multiplicarse por la matriz global inversa transpuesta.
- Por esta razón, conviene precalcular la inversa de cada transformación

Espacios de transformación

Transformar normales (o cualquier vector dirección) requiere un caso especial



Operaciones matemáticas

- Clamp
- Lerp
- Step
- Smoothstep

Clamp

Restringe el valor sobre un rango aceptado

$$clamp(x) = \begin{cases} max & \text{if } x \geq max \\ min & \text{if } x \leq min \\ x & otro \end{cases}$$

Lerp

Interpola linealmente entre dos valores

$$lerp(a, b, t) = (1 - t) \cdot a + t \cdot b$$

Step

Umbral sencillo

$$step(x, \mu) = \begin{cases} 0 & \text{if } x < \mu \\ 1 & otro \end{cases}$$

(a) 64x64 texture, alpha-blended (b) 64x64 texture, alpha tested (c) 64x64 texture using our technique

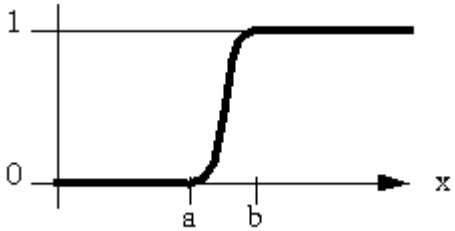
SmoothStep

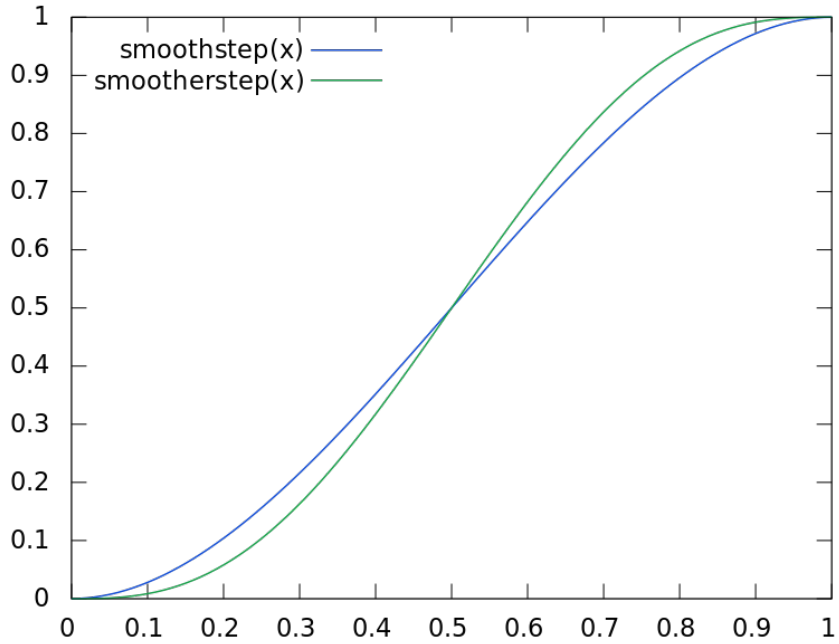
Umbral suavizado

$$smoothstep(t) = 3t^2 - 2t^3$$

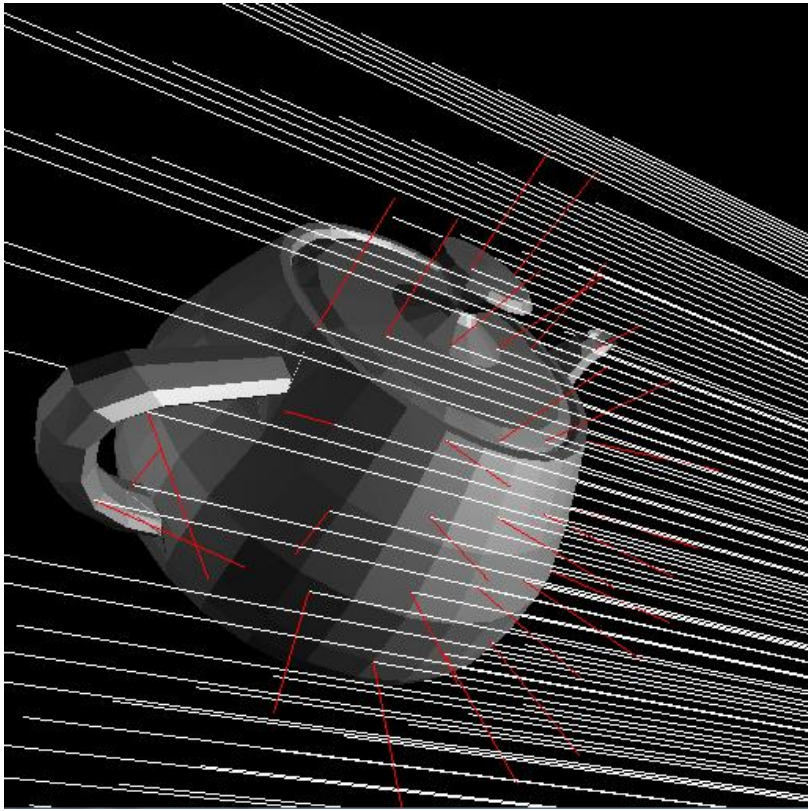
$$smootherstep(t) = 6t^5 - 15t^4 + 10t^3$$

Con rango:



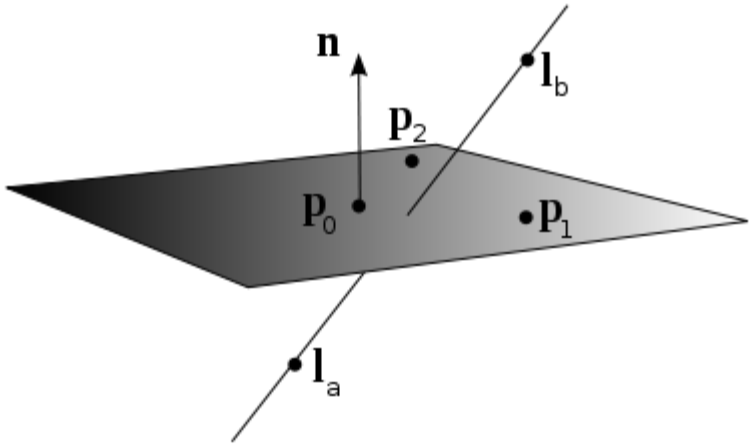


Intersecciones



Rayo

- Origen
- Dirección
- Distancia máxima



$$P = P_0 + t \cdot V$$

También...

$$R = o + t \cdot \mathbf{d}$$

Primitivas e Intersecciones

- Plano
- Esfera
- Disco
- Cilindro
- Triángulo
- Caja
- AABB

Primitivas e Intersecciones

Datos de salida necesarios de cada intersección:

- Punto
- Normal
- Tangente (más adelante…)
- Coordenadas UV (más adelante…)

Nota: para el punto, solo es necesario saber T

Primitivas e Intersecciones

Para simplificar los cálculos, es **ideal** transformar el rayo a espacio local!

De esta forma, no hay que considerar posicion, traslación ni escala de cada objeto.

Primitivas - Plano

Definido por un punto y una normal

$$(\mathbf{p} - \mathbf{p_0}) \cdot \mathbf{n} = 0$$

Sustituyendo el rayo en P, obtenemos

$$(\mathbf{P_0} + \mathbf{t} \cdot \mathbf{V} - p_0) \cdot n = 0$$

Primitivas - Plano

Tomando,

$$D = p_0 \cdot n$$

la solución es

$$t = -\frac{(P_0 \cdot n + D)}{V \cdot n}$$

Primitivas - Plano

En nuestro caso D = 0, por lo que

$$t = -\frac{P_0 \cdot n}{V \cdot n}$$

Ademas, obtener la normal es fácil, ya que es parte de la definición del plano.

Primitivas - Plano

Puede no colisionar nunca!

```
protected CollisionData localCheckCollision(Ray ray) {
        Vector3 rDir = ray.getDirection();
        Vector3 rPos = ray.getPosition();
        double collide = rDir.dotProduct(getNormal());

        if (Math.abs(collide) > 0) {
                double t = -(rPos.dotProduct(getNormal()) / rDir.dotProduct(getNormal()));
                if (t > 0)
                        return new CollisionData(this, t, getNormal());
        }
        return CollisionData.noCollision();
}
```

Primitivas - Esfera

Definida implícitamente por un punto y un radio

$$x^2 + y^2 + z^2 - r^2 = 0.$$

Sustituyendo el rayo,

$$(\mathrm{o}_x + t\mathbf{d}_x)^2 + (\mathrm{o}_y + t\mathbf{d}_y)^2 + (\mathrm{o}_z + t\mathbf{d}_z)^2 = r^2.$$

Primitivas - Esfera

Reordenando la ecuación,

$$At^2 + Bt + C = 0,$$

$$A = \mathbf{d}_x^2 + \mathbf{d}_y^2 + \mathbf{d}_z^2$$

$$B = 2(\mathbf{d}_x \mathrm{o}_x + \mathbf{d}_y \mathrm{o}_y + \mathbf{d}_z \mathrm{o}_z)$$

$$C = \mathrm{o}_x^2 + \mathrm{o}_y^2 + \mathrm{o}_z^2 - r^2.$$

Primitivas - Esfera

Resolviendo la ecuación cuadrática,

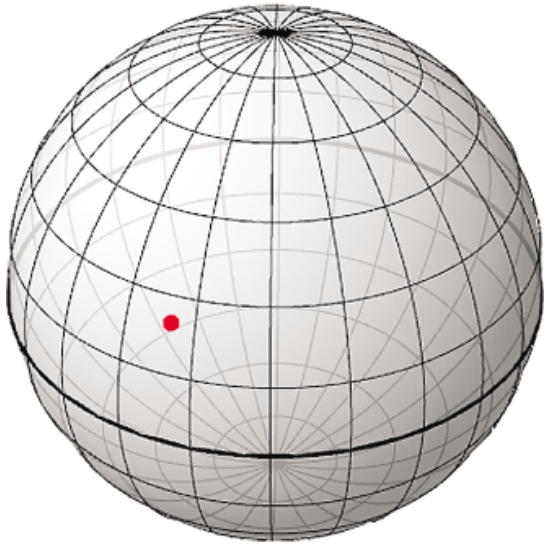
$$t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

$$t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}.$$

Se puede aprovechar el valor del discriminante para saber si hay raíces reales!

Primitivas - Esfera

La normal es el punto normalizado:



Primitivas - Discos y planos finitos

Utilizando la ecuación del plano, se pueden agregar limitaciones al resultado dependiendo del punto de colisión con respecto al origen.

Discos: si la distancia al origen es mayor que r, no hay intersección

Primitivas - Discos y planos finitos

Utilizando la ecuación del plano, se pueden agregar limitaciones al resultado dependiendo del punto de colisión con respecto al origen.

Plano finito: si el punto no cae dentro de un rectángulo definido del plano en el origen, no hay intersección

Primitivas - Cilindros

Definido implícitamente por

$$x^2 + y^2 - r^2 = 0.$$

Por lo que, expandiendo con la ecuación del rayo…

$$\left(\mathrm{o}_x + t\mathbf{d}_x\right)^2 + \left(\mathrm{o}_y + t\mathbf{d}_y\right)^2 = r^2.$$

Resolver cuadrática!

Primitivas - Triángulo

- Definido por 3 puntos
- La normal es implícita (pero puede no ser lo que queremos)

Primitivas - Triángulo

- Primero se busca una intersección con el plano
- Luego se verifica que caiga dentro del triángulo
- Se utilizan coordenadas baricéntricas

$$\mathrm{p}(b_1, b_2) = (1 - b_1 - b_2)\mathrm{p}_0 + b_1\mathrm{p}_1 + b_2\mathrm{p}_2.$$

$$b_1 \geq 0,\ b_2 \geq 0,\ \text{and}\ b_1 + b_2 \leq 1.$$

Perfecto para interpolar luego! $(1 - b_1 - b_2)a_0 + b_1a_1 + b_2a_2.$

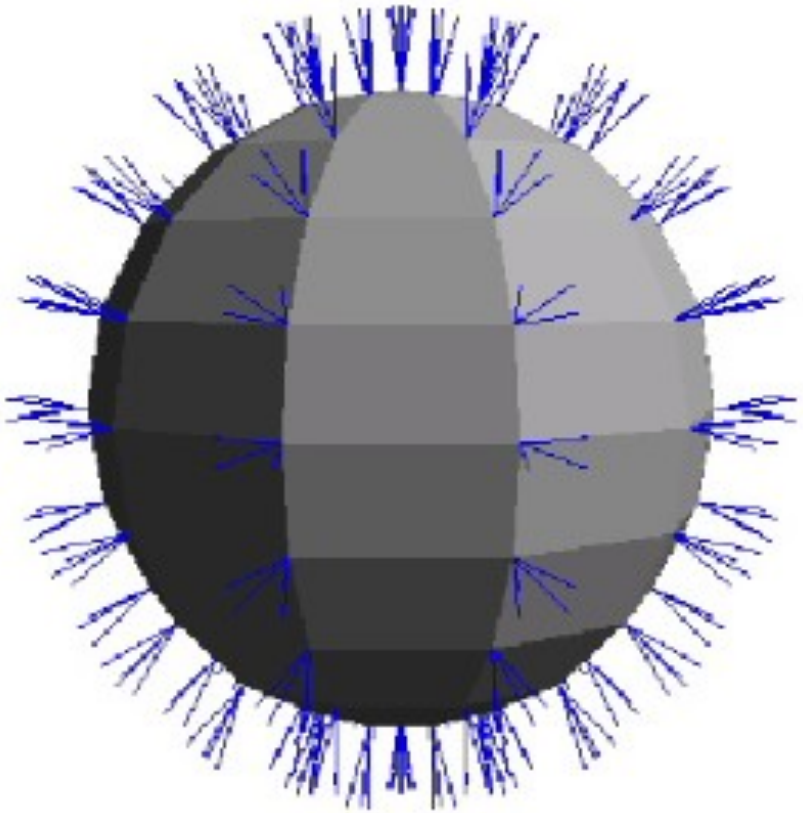
Referencia: "Fast, Minimum Storage Ray/Triangle Intersection" [Möller, Trumbore]

Primitivas - Triángulo

Cómo calcular la normal?

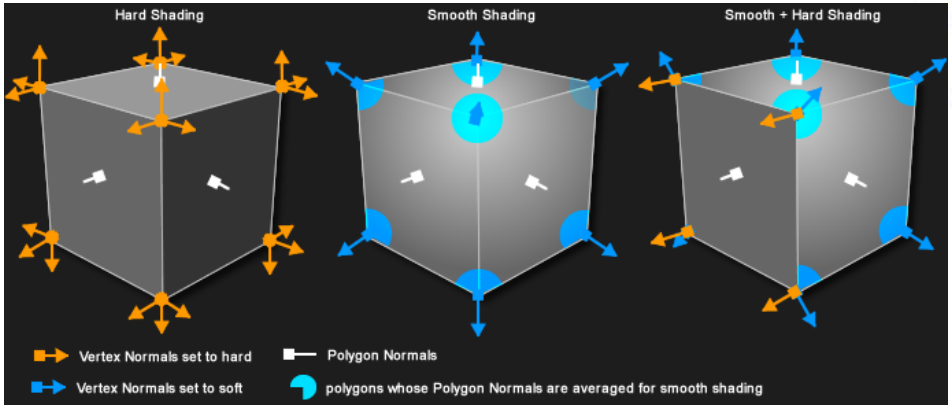
Podría ser la del plano:

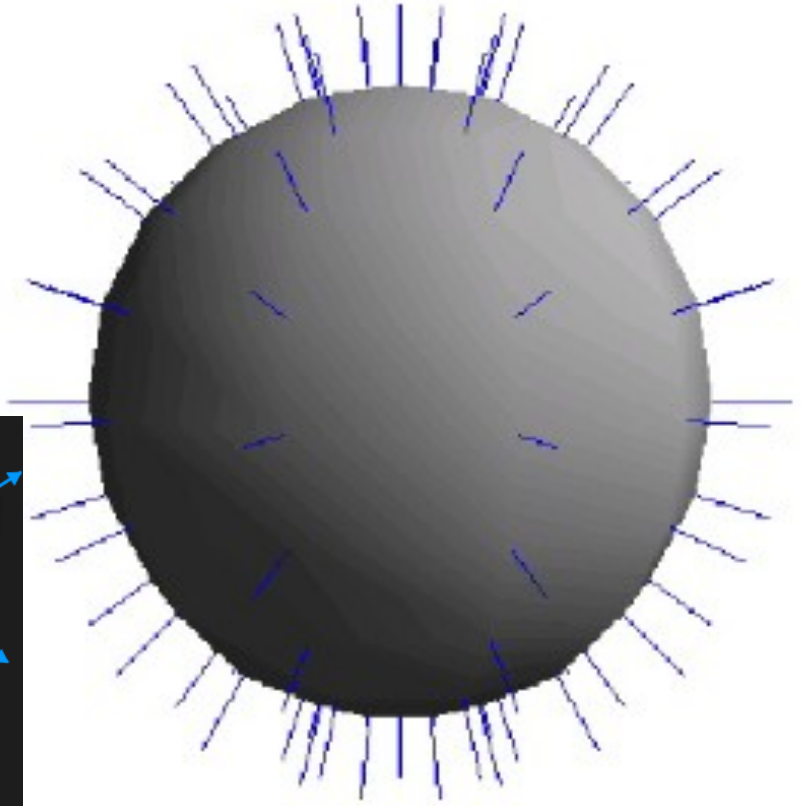
Se podrá suavizar?



Primitivas - Triángulo

Si la definición de cada triángulo contiene 3 normales, estos podrían compartir las normales con sus vecinos.



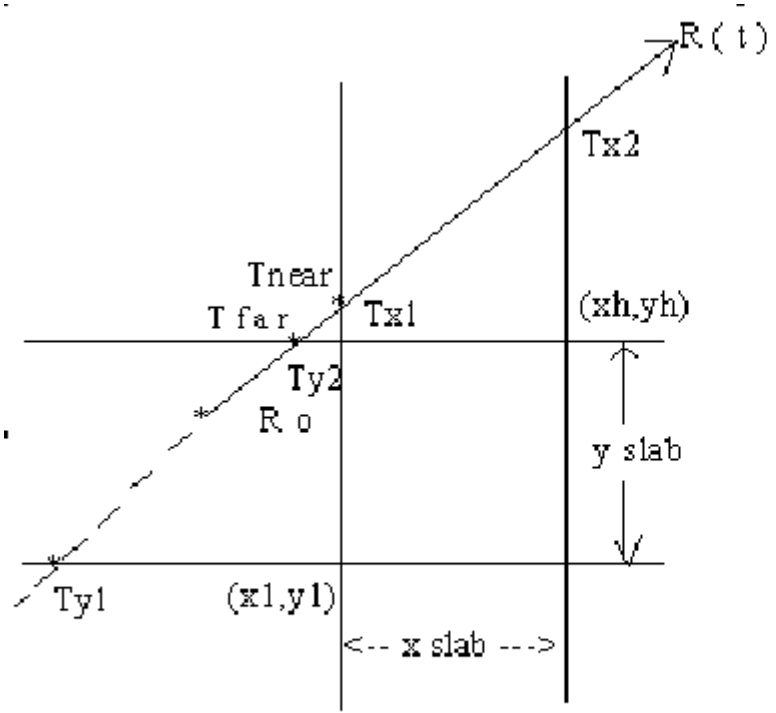


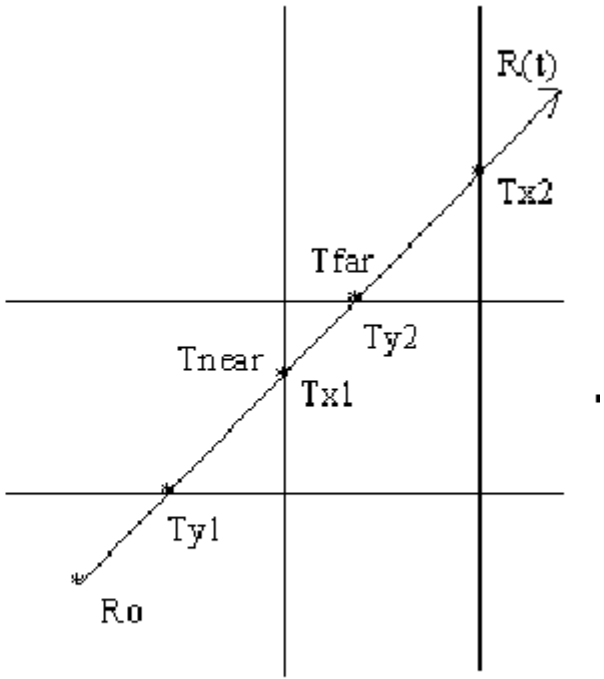
Primitivas - Caja

- Definido por un punto y tres valores que definen su tamaño.
- 6 intersecciones rayo-plano finito

Referencia: An Efficient and Robust Ray–Box Intersection Algorithm,

Primitivas - Caja

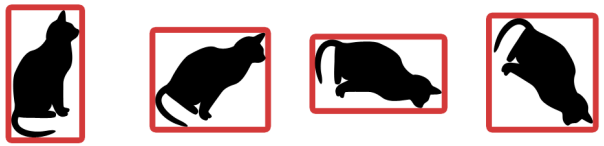




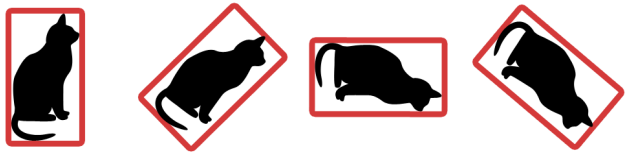
Primitivas - Axis Aligned Bounding Box

- Se utiliza para precalcular el mínimo volumen que contiene a un objeto.
- Definido como una caja, pero sin rotación

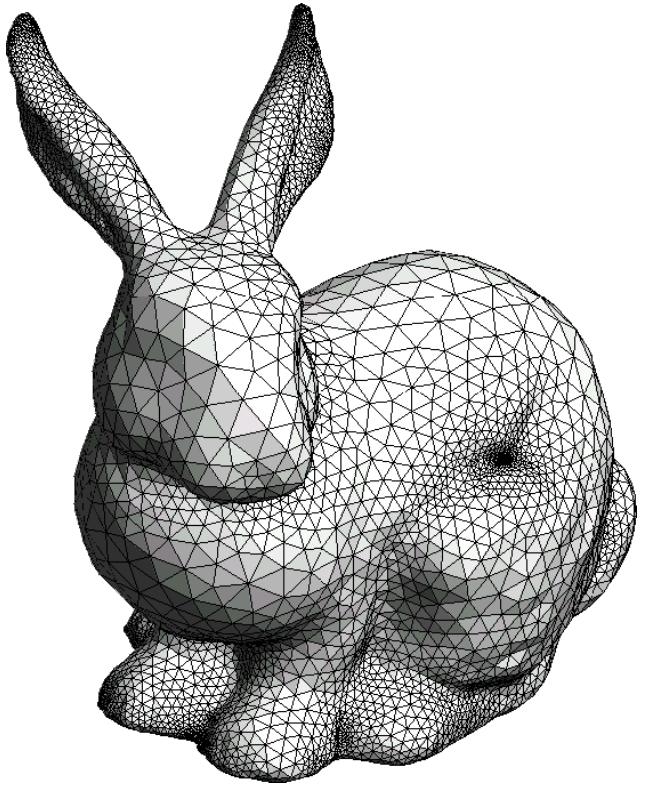
Sin rotación:



Con rotación:



Primitivas - Mesh



Primitivas - Mesh

Utilizando las primitivas, un Mesh puede definirse como un conjunto de triángulos.

Para acelerar el cálculo de la intersección, se suele precalcular un AABB para saber si el rayo puede llegar a intersectar

Colisiones

Estos algoritmos son de referencia, y hay constante research para mejorarlos.

www.realtimerendering.com/intersections.html

Función de Visibilidad

Define si dos puntos son mutuamente visibles.

$$\forall x, y \in A: \; V(x,y) = \begin{cases} 1 & \text{if x and y are mutually visible} \\ 0 & \text{if x and y are not mutually visible} \end{cases}$$

Escena

1278 objetos
465k triángulos

Escena

Una escena puede estar compuesta por miles de objetos.

Cómo encontrar la solución a la función de visibilidad de forma eficiente?

Lo mismo puede suceder con un Mesh.