Examen de Teoria (2n parcial) – 13 de Juny de 2017
Dolça Tellols. Revisat.

a) EXERCICI SOBRE ELS ATRIBUTS DE LES CÀMERES

Per aconseguir la visualització que demana l'enunciat, són necessàries 4 càmeres, una per cada jugador. A continuació es detallen tots els atributs de cadascuna.

Atributs en coordenades de món / Atributs en coordenades de càmera

	P1 X negatives	P2 Z positives	P3 X positives	P4 Z negatives
Posició de l'observador	(-50.000,0,0)	(0,0,5.000)	(50.000,0,0)	(0,0,-5.000)
VRP	(-49900,0,0)	(0,0,4.900)	(49900,0,0)	(0,0,-4.900)
VUP	(0,1,0)	(0,1,0)	(0,1,0)	(0,1,0)
Look (de l'obs. al VRP)	(1,0,0)	(0,0,-1)	(-1,0,0)	(0,0,1)
D (dist. de l'obs. al VRP)	100	100	100	100
Frustrum (Znear, Zfar)	Znear = 0 Zfar = 3000	Znear = 0 Zfar = 3000	Znear = 0 Zfar = 3000	Znear = 0 Zfar = 3000
Volum de visió (window)	O (-1250, -893) a = 2500 h = 1786	O (-1250, -893) a = 2500 h = 1786	O (-1250, -893) a = 2500 h = 1786	O (-1250, -893) a = 2500 h = 1786
Pla de projecció	Es troba a distància z=100	Es troba a dist. z=100	Es troba a dist. z=100	Es troba a dist. z=100
Tipus de projecció	Paral·lela	Paral·lela	Paral·lela	Paral·lela
Viewport	Origen (0,0) a = 1080 h = 772	Origen (0,0) a = 1080 h = 772	Origen (0,0) a = 1080 h = 772	Origen (0,0) a = 1080 h = 772

El centre de la Window sempre és l'origen (0,0) en coordenades de càmera.

b) VISUALITZACIÓ AMB ZBUFFER

El Phong shading s'implementa en el fragment shader.

La detecció de siluetes es fa amb la fòrmula:

- acos(vector visió, normal) > 5 => silueta

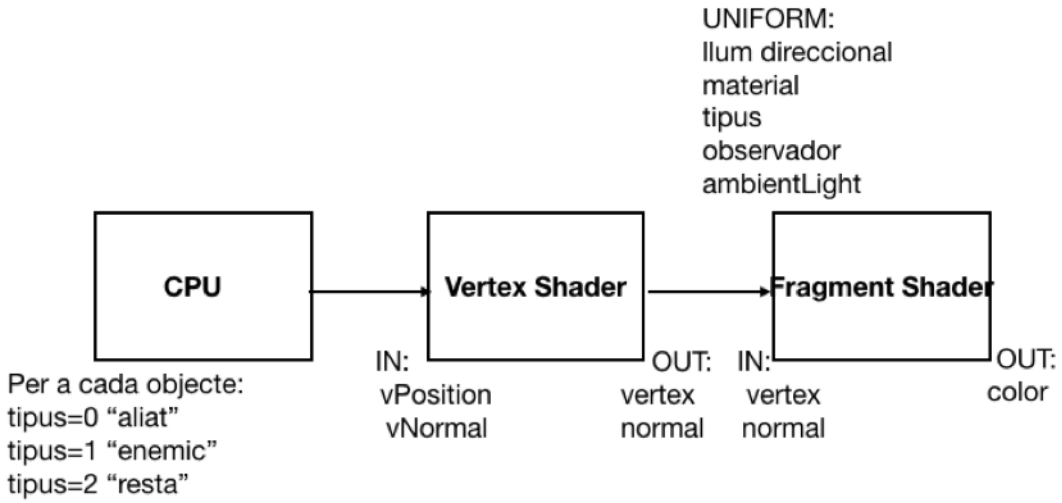
Voldríem preguntar en el cas aliat/enemic:

- if (silueta) – Pintem vermell o blau
- else – Pintem normal

Necessitem saber si el que estem pintant és:

- Aliat / Enemic / Objecte normal

Per tant, el pas de variables a la GPU es sintetitza en la següent figura:



A nivell de CPU els objectes necessiten un flag (tipus) per saber si són aliat/enemic o res. Aquest flag es passarà com a uniform des de CPU al fragment shader:

```
uint type = program->uniformLocation("tipus")
glUniform1i(type,typeFlag); // "0" aliat, "1" enemic, "2" resta
```

I aquesta informació l'aconseguim al fragment shader passant-li-la en una variable uniform.

Els shaders implementant PhongShading considerant només una llum direccional, quedarien de la següent manera (tot i que aquí està en codi, es pot fer en forma de pseudocodi):

Vèrtex shader

```
#version 330
layout (location = 0) in vec4 vPosition;
layout (location = 1) in vec4 vNormal;

uniform mat4 model_view;
uniform mat4 projection;

out vec4 vertex;
out vec4 normal;

void main(){
    gl_Position = projection*model_view*vPosition;
    gl_Position = gl_Position/gl_Position.w;

    //In Phong shading we want to interpolate the normals
    vertex = vPosition;
    normal = vNormal;
}
```

Fragment shader

```
#version 330

in vec4 vertex;
in vec4 normal;

struct Material {
    vec3 ka;
    vec3 kd;
    vec3 ks;
    float shineness;
    float transparency;
};
struct Light {
    vec3 direction;
    vec3 ia;
    vec3 id;
    vec3 is;
};

//Type of object "0" aliat, "1" enemic, "2" resta
uniform int tipus;

//Material
uniform Material obj_mat;

//Lights
uniform Light lightGPU;
uniform vec3 ambientLightGPU;

//Viewer position
uniform vec4 obs;
```

```
//Final colour
out vec4 colorOut;

void main() {
    vec3 ka = obj_mat.ka;
    vec3 kd = obj_mat.kd;
    vec3 ks = obj_mat.ks;
    float shineness = obj_mat.shineness;

    vec3 point = vec3(vertex.x,vertex.y,vertex.z);
    vec3 n = normalize(vec3(normal.x, normal.y, normal.z));
    vec3 pLight; //Light point
    vec3 vLight; //L = light vector normalized
    vec3 vVisio = normalize(vec3(obs.x, obs.y, obs.z)-point);

    vec3 h; //H = L+V normalized
    float ln; //LxN
    float nh; //NxH
    float blinn; //NxH to the power of shineness

    vec3 faseAmbientG = ambientLightGPU*ka;
    vec3 faseAmbient = vec3(0.0,0.0,0.0);
    vec3 faseDifusa = vec3(0.0,0.0,0.0);
    vec3 faseEspecular = vec3(0.0,0.0,0.0);

    vec3 color = vec3(0.0,0.0,0.0);
    vec4 new_color;

    //Angle for Silhouette emphasis
    float angle = acos(dot(vVisio, n));

    if(angle < 5 || type==2){ //Blinn-Phong
        vLight = normalize(-lightGPU.direction);
        h = normalize(vLight+vVisio);
        ln = dot(vLight,n);
        nh = dot(n,h);
        blinn = pow(nh, shineness);

        faseAmbient = ka*lightGPU.ia;
        faseDifusa = kd*lightGPU.id*max(ln,0.0);
        faseEspecular = ks*lightGPU.is*max(blinn,0.0);
        color = color + faseDifusa+faseEspecular + faseAmbient;
       color = color + faseAmbientG;
       new_color = vec4(color.x,color.y,color.z,obj_mat.transparency);
    } else {
       // Silhouette emphasis
       vec3 sColour;
       switch(tipus){
           case 0: //Aliat
               sColour = vec3(0.0,0.0,1.0);
               break;
           case 1: //Enemic
               sColour = vec3(1.0,0.0,0.0);
               break;
       }
       new_color = vec4(sColour*(1-cos(angle)), obj_mat.transparency);
    }

    colorOut = new_color;
}
```

c) VISUALITZACIÓ AMB RAY-TRACING

Mètode de la classe render.cpp (No canvia)

```
void Render::rendering() {
    // Recorregut de cada pixel de la imatge final
    for (int y = scene->cam->viewportY-1; y >= 0; y--) {
         for (int x = 0; x < scene->cam->viewportX; x++) {
            vec3 col(0, 0, 0);
            float u = float(x) / float(scene->cam->viewportX);
            float v = float(y) / float(scene->cam->viewportY);
            Ray r = scene->cam->getRay(u, v);
            col = scene->ComputeColor(r,0);
            //Antialiasing, gamma-correction and colour normalization
            setPixelColor(col, x, y);
         }
    }
}
```

Mètodes de la classe Scene.cpp

Es podria modificar el hitInfo per a que al fer el hit es guardés en un paràmetre (type) si es tracta de personatge o objecte normal. Aquest identificador es trobaria guardat en els objectes i s'"agafaria" al fer el hit amb ells.

El mètode **hit** quedaria modificat així:

```
bool Scene::hit(const Ray& raig, float t_min, float t_max, HitInfo&
info) const {
    bool inter = false;
    unsigned i;
    HitInfo *tempHI = new HitInfo();
    for (i=0;i<this->objects.size();i++){ //All objects checked
        if(this->objects.at(i)->hit(raig,t_min,t_max,info)){
            inter = true;
            if (info.t<tempHI->t){ //Check if it is closer
                tempHI->t = info.t;
                tempHI->p = info.p;
                tempHI->normal = info.normal;
                tempHI->mat_ptr = info.mat_ptr;
                tempHI->type = info.type;
            }
        }
    }
    if(inter){ //Update ray info with the closest one
        info.t = tempHI->t;
        info.p = tempHI->p;
        info.normal = tempHI->normal;
        info.mat_ptr = tempHI->mat_ptr;
        info.type = tempHI->type;
    }
    delete tempHI;
    return inter;

}
```

Així, el mètode **computeColor**, es modifica per contemplar els tres casos de hit:

- Si es tracta d'un personatge, cal remarcar la silueta, per tant, amb el nou camp de hitInfo es sap si és personatge i si per tant si cal remarcar la silueta. El mètode Blinn-Phong serà modificat conseqüentment per tenir-ho en compte.
- Si es tracta d'un altre objecte que no és personatge, guardaria el color calcular i després emetria un raig a partir del punt de xoc en la mateixa direcció que cerqués un possible personatge (nou mètode especial hitPer). En cas de trobar-lo, retornaria el color del personatge més proper (blau o vermell i remarcant la silueta). Finalment el color final seria una ponderació del color de l'objecte i del color del personatge (en cas d'haver-hi un darrere).

```
vec3 Scene::ComputeColor (Ray &ray, int depth) {
    vec3 color; //To save the point color
    HitInfo *hInfo = new HitInfo();
    std::vector<Ray> secRays;
    vec3 K = vec3(1.0,1.0,1.0);
    unsigned int i;
    vec3 compC(0.0f,0.0f,0.0f);
    float tmin = 0.0f; //No epsilon is applied to the primary ray
    if(depth>0){
        tmin = 0.01f; //Epsilon
    }
    if (this->hit(ray,tmin,100,*hInfo)){
        switch(hInfo->type){
           case "personatge":
                //Cal destacar la silueta al pintar el color
                color = this->Shade(*hInfo,ray);
               if(depth<MAXDEPTH){ //MAXDEPTH == 1
                   hInfo->mat_ptr->scatter(ray,*hInfo,K,secRays);
                   for(i=0;i<secRays.size();i++){
                       compC += ComputeColor(secRays.at(i),depth+1);
                   }
                   if (secRays.size()>1){ //Multiple scattering
                       compC.x = compC.x/secRays.size();
                       compC.y = compC.y/secRays.size();
                       compC.z = compC.z/secRays.size();
                   }
                   color += K*compC;
               }

                break;
           case "objecte_normal":
                //Cal mirar si darrere hi ha un personatge
                color = this->Shade(*hInfo,ray);
               if(depth<MAXDEPTH){ //MAXDEPTH == 1
                   hInfo->mat_ptr->scatter(ray,*hInfo,K,secRays);
                   for(i=0;i<secRays.size();i++){
                       compC += ComputeColor(secRays.at(i),depth+1);
                   }
                   if (secRays.size()>1){ //Multiple scattering
                       compC.x = compC.x/secRays.size();
                       compC.y = compC.y/secRays.size();
                       compC.z = compC.z/secRays.size();
                   }
                   color += K*compC;
               }
```

```
                HitInfo *newhInfo = new HitInfo();
                Ray newRay = newRay(hInfo.point,ray.direction);
                vec3 newColour =this->hitPer(newRay,tmin,100,*newhInfo);
                if newColour!=vec3(0,0,0){
                    //Pondero el color del mur i el del personatge
                    color = 0.1*color+0.9*newColour;
                }
                break;
        }
    } else { //Background
        color = "background_colour";
    }
    secRays.clear();
    delete hInfo;
    return color;
}
```

El mètode Shade no varia.

```
vec3 Scene::Shade (HitInfo& info, const Ray& ray){
    vec3 color;
    color     =     blinnPhong(info.p,     info.normal,     info.mat_ptr,
ray.initialPoint(), info.type);
    }
    return color;
}
```

Nou mètode hit (hitPer) per cercar el personatge de darrere més proper i que retorna el seu color en cas de trobar-lo. Els materials dels objectes tindran guardat un nou atribut vec3 amb el "colorPropi" que els toca depenent de si són aliats (vec3(0.0,0.0,1.0)) o enemics (vec3(1.0,0.0,0.0)).

```
vec3 Scene::hitPer(const Ray& raig, float t_min, float t_max, HitInfo&
info) const {
    bool inter = false;
    unsigned i;
    vec3 color = vec3(0.0,0.0,0.0);
    HitInfo *tempHI = new HitInfo();
    for (i=0;i<this->objects.size();i++){ //All objects checked
        if(this->objects.at(i)->hit(raig,t_min,t_max,info)){
            inter = true;
               //Check if it is closer i és un personatge
            if (info.t<tempHI->t && info.type==”personatge”){
                tempHI->t = info.t;
                tempHI->p = info.p;
                tempHI->normal = info.normal;
                tempHI->mat_ptr = info.mat_ptr;
                 tempHI->type = info.type;
             }
        }
    }
    if(inter){ //Update ray info with the closest one
        info.t = tempHI->t;
        info.p = tempHI->p;
        info.normal = tempHI->normal;
        info.mat_ptr = tempHI->mat_ptr;
       info.type = tempHI->type;
       color = mat_ptr.colorPropi;
       }

    delete tempHI;
    return color;

}
```

Altres alternatives referents al mètode hitPer:

- Optimitzar el for recorrent només els objectes de tipus personatges si es trobessin guardats en una llista apart.
- O bé modificar el hit d'objecte per comprovar directament si és personatge (així no caldria calcular la intersecció amb objectes que no són personatges).

```
vec3 Scene::blinnPhong(vec3 point, vec3 normal, const Material* mat,
vec3 vPoint, int type){
    HitInfo *hInfo = new HitInfo();
    //Ambient, difuse and specular components of the material
    vec3 ka = mat->ka;
    vec3 kd = mat->kd;
    vec3 ks = mat->ks;
    float shineness = mat->shineness;
    vec3 n = normal; //N = hit normal
    vec3 vLight; //L = light ray vector normalized
    vec3 vVisio = normalize(vPoint-point); //V=vision ray vector norm.
    vec3 h;// H = L+V normalized
    float ln; //LxN
    float nh; //NxH
    float blinn; //NxH to the power of shineness

    Ray rayL;
    vec3 faseAmbientG = iAmbientGlobal->getIntensity()*ka;
    vec3 faseAmbient = vec3(0.f,0.f,0.f);
    vec3 faseDifusa = vec3(0.f,0.f,0.f);
    vec3 faseEspecular = vec3(0.f,0.f,0.f);
    float shadeFactor;
    vec3 color = vec3(0.f,0.f,0.f);

    vLight = normalize(-light->getdirection());
    h = normalize(vLight + vVisio);
    ln = dot(vLight,n);
    nh = dot(n,h);
    blinn = pow(nh,shineness);
    rayL = Ray(point, vLight);
    shadeFactor = calculateShade(rayL, 0.01, d, *hInfo);

    //Angle for Silhouette emphasis
    float angle = acos(dot(vVisio, n));

    if(angle < 5 or type==”objecte_normal”){ //Blinn-Phong
        faseAmbient = ka*light->getIAmbient();
        faseDifusa = kd*light->getIDifusa()*std::max(ln,0.f);
        faseEspecular = ks*light->getIEspecular()*std::max(blinn,0.f);

        color = shadeFactor*(faseDifusa+faseEspecular)+faseAmbient;
        color = color + faseAmbientG;
    } else {
       //Silouette emphasize
       vec3 sColour = mat->colorPropi;
        color = sColour*(1-cos(angle));
    }
    delete hInfo;
    return color;
}


float Scene::calculateShade(const Ray& ray, float tMin, float tMax,
HitInfo& hInfo){
    shadowF = 1.0f; // No shadow by default
    hitShade(ray,tMin,tMax,hInfo); //Shadow factor updated
    return shadowF; //No object between light and the point
}
```

```
bool Scene::hitShade(const Ray& raig, float t_min, float t_max,
HitInfo& info){
    bool inter = false;
    unsigned i;
    for (i=0;i<this->objects.size()&&!inter;i++){ //Check all objects
        if(this->objects.at(i)->hit(raig,t_min,t_max,info)){
            if(objects.at(i)->getAFactor()>0.0f){
                shadowF -= objects.at(i)->getAFactor()*objects.at(i)
                           ->calculateDist(raig);
                inter = true;
            } else {
                //Intersection with non transparent material =>
               complete shadow
                shadowF = 0.0f;
               inter = true;
            }
        }
    }
    if(shadowF < 0.0f){
        shadowF = 0.0f;
    }
    return inter;
}
```