# *HMIN317: Moteurs de jeux*

Rendu TP5 : Shadow Map

Tianning MA

M2 IMAGINA

14/12/2020

# Table de matière



# 1. Introduction

Ce compte rendu est dédié au TP5 Rendu - shadowMap au cours de Moteur des jeux.

Toutes les questions sont répondues.

Vous trouvez aussi une démonstration sous format d'une vidéo et le code source à l'adresse du mon espace git pour ce tp : https://github.com/matianning/Moteurs_de_jeux

# 2. Rendu et explication des exercices

## 2.1 Construction des shadow maps

Pour définir une caméra appropriée pour la lumière de la scène, j'ai utilisé la fonction glm ::ortho et glm ::lookAt. Glm ::ortho nous permet d'avoir une matrice pour la lumière directionnel et glm ::lookAt nous permet d'avoir la matrice de vue en fonction de la position de la lumière et le centre de la scène.

Mon implémentation est comme l'image ci-dessous.

```
void setupCameraForShadowMapping(glm::vec3 scene_center , float scene_radius) {
    float near_plane = 1.0f, far_plane = 7.5f;
    glm::mat4 lightProjection = glm::ortho(-scene_radius, scene_radius, -scene_radius, scene_radius, near_plane, far_plane);
    glm::mat4 lightView = glm::lookAt(m_position, scene_center, glm::vec3( 0.0f, 1.0f,  0.0f));
    depthMVP = lightProjection * lightView;
}
```

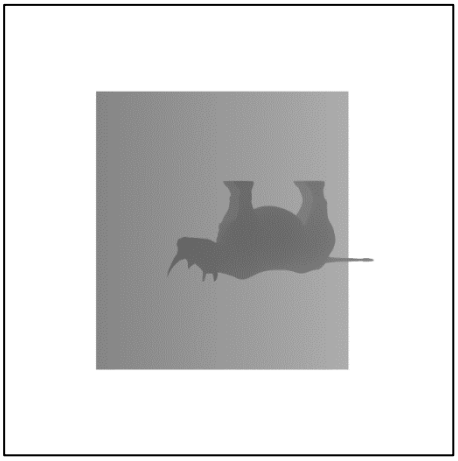


*Figure 1 shadowMap*

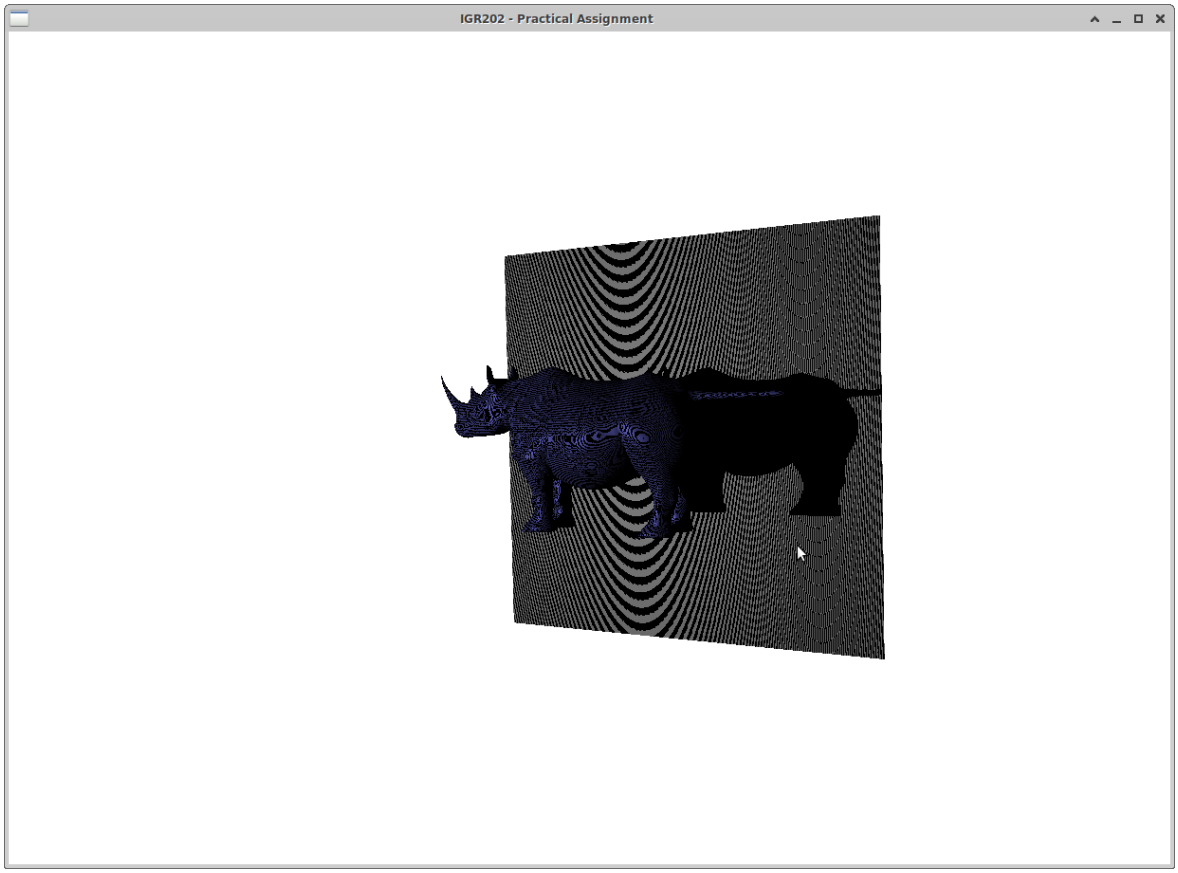
Ensuite, avec l'interface et la touche S, j'obtiens la carte de profondeur comme ci-dessous.

On peut constater que la forme de maillage est bien sur l'image. En effet, cette image est la carte de profondeur à partir de la position de la lumire. On va s'en servir pour générer l'ombre après.

## 2.2 Rendre les ombres

Pour rendre les ombres, on a besoin de transférer le shadowMap (calculé par l'étape précédent) et la matrice de transformation au shader principal. Ensuite, pour chaque fragment, on va déterminer s'il est dans la zone d'ombre ou non. Selon cela, on va changer la couleur du fragment courant différement.

Pour calculer l'ombre (la fonction consiste à déterminer si c'est dans l'ombre ou non), j'ai utilisé la fonction vue en cours (ShadowCalculation).

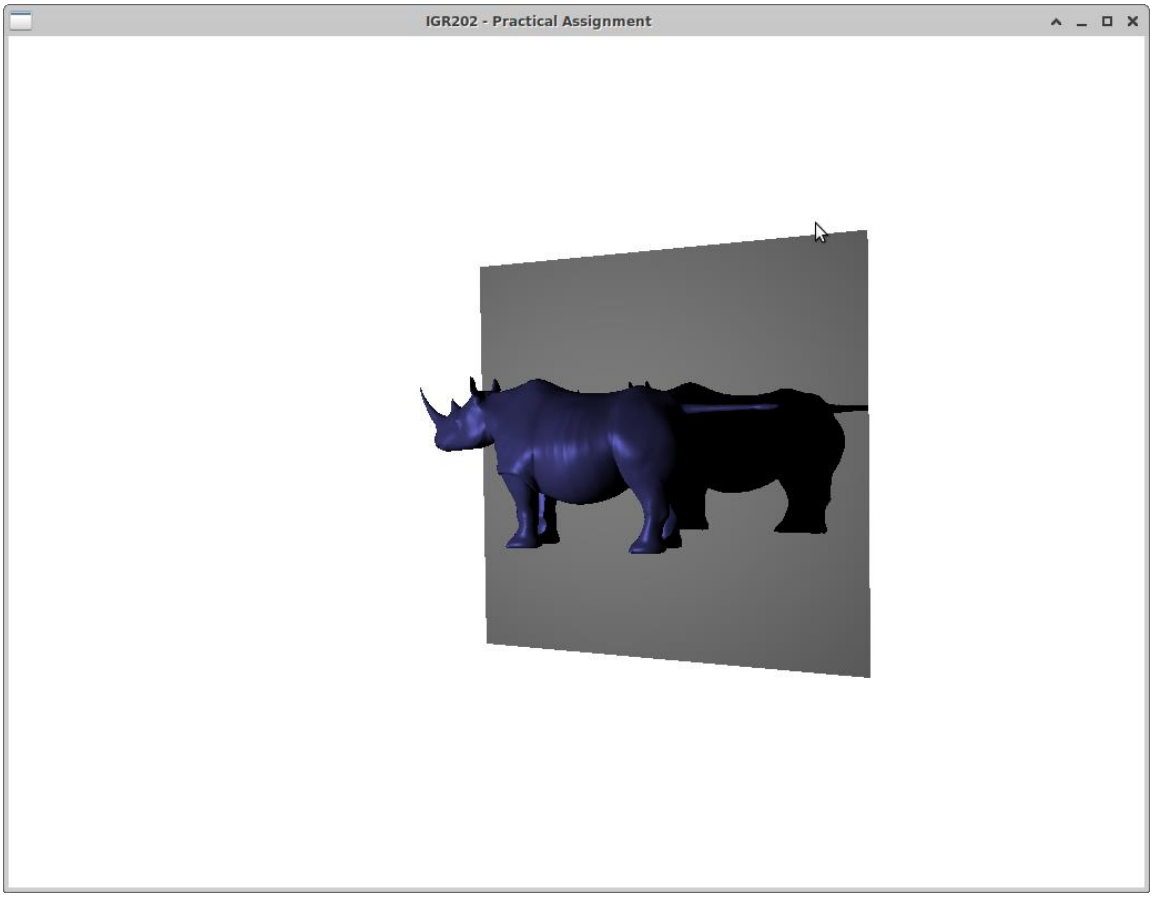



On s'aperçoit l'artefact d'auto-ombrage sur l'image. En effet, ces shadow acne sont apparaits à cause de limite de résolution de la shadowmap : Plusieurs fragments ont la même valeur de la carte de profondeur. Pour corriger cela, on pourra ajouter un bias, donc chaque fragment sera considéré au dessus de la surface.

On pourra simplement mettre une constante comme bias, mais ce bias dépendra l'angle entre la source de la lumière et la surface. Donc, ici j'ai calculé le bias en fonction de cet angle (dans la fonction ShadowCalculation de FragmentShader).

```
//Calculer bias pour éviter Shadow acne
vec3 normal = normalize(fNormal);
vec3 lightDir = normalize(lightSource[i].position - fPosition);
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);

shadow = currentDepth - bias > closestDepth  ? 1.0 : 0.0;
```

En ajoutant ce bias, on constate (l'image ci-dessous) que l'artefact est bien disparu, et l'ombrage du maillage est bien affiché sur la surface.




## 2.3 Ajout des lumieres

Cette partie consiste à ajouter plusieurs lumières dans la scène. Chaque lumière va avoir son shadowMap, et pour le fragment Shader final, il faut prendre en compte toutes les lumières pour calculer l'ombrage.

Donc, dans le fragment shader, j'ai changé LightSource en un tableau pour sauvegarder toutes les lumières, et une variable uniforme nb_light pour indiquer combien de lumière dans la scène.

```
uniform LightSource lightSource[2];
uniform int nb_light;
```

(Par exemple ici, on a deux lumières dans la scène). Dans le main du shader, on va parcourir ce tableau de lumière, et pour chaque lumière, ajouter les valeurs dans la radiance de sortie pour chaque fragement.

```
for(unsigned int i = 0; i < nb_light; i++){
```

Dans le **main.cpp** :

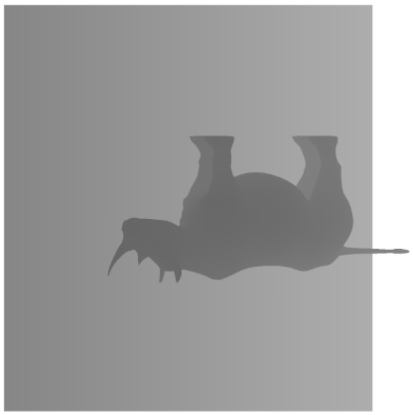
setup des lumières :

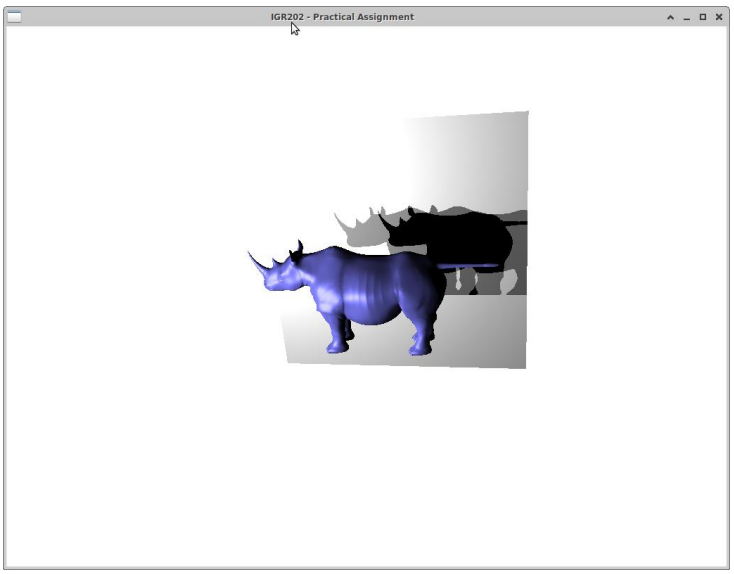
```
unsigned int shadow_map_width = 2000 , shadow_map_height = 2000; // play with these parameters

{
    scene.scene_lights.resize( scene.scene_lights.size() + 1 );
    Light & newLight = scene.scene_lights[ scene.scene_lights.size() - 1 ];
    newLight.m_position = glm::vec3 (-2*cos(app_timer), 2*sin(app_timer), 3.5);
    newLight.m_color = glm::vec3(1.0, 1.0, 1.0);
    newLight.m_intensity = 0.5f;
    newLight.allocateShadowMapFBO(shadow_map_width , shadow_map_height);
    newLight.shadowMapTexOnGPU = texture_slot_available;   ++texture_slot_available;
}

{
    scene.scene_lights.resize( scene.scene_lights.size() + 1 );
    Light & newLight = scene.scene_lights[ scene.scene_lights.size() - 1 ];
    newLight.m_position = glm::vec3 (-2*cos(app_timer), 2*sin(app_timer), 1.5);
    newLight.m_color = glm::vec3(1.0, 1.0, 1.0);
    newLight.m_intensity = 0.8f;
    newLight.allocateShadowMapFBO(shadow_map_width , shadow_map_height);
    newLight.shadowMapTexOnGPU = texture_slot_available;   ++texture_slot_available;
}
```

Transférer les informations des lumières au shader :

```
// Set the lights in the shader :
{

    Light & light = scene_lights[0];
    shaderProgramPtr->set (std::string("lightSource[0].position"), light.m_position);
    shaderProgramPtr->set (std::string("lightSource[0].color"), light.m_color);
    shaderProgramPtr->set (std::string("lightSource[0].intensity"), light.m_intensity);
    shaderProgramPtr->set (std::string("lightSource[0].isActive"), 1);

    shaderProgramPtr->set (std::string("depthMVP[0]"), light.depthMVP);
    shaderProgramPtr->set (std::string("shadowMap[0]"), (int) light.shadowMapTexOnGPU);


    //*****************Deuxième light******************
    Light & light2 = scene_lights[1];
    shaderProgramPtr->set (std::string("lightSource[1].position"), light2.m_position);
    shaderProgramPtr->set (std::string("lightSource[1].color"), light2.m_color);
    shaderProgramPtr->set (std::string("lightSource[1].intensity"), light2.m_intensity);
    shaderProgramPtr->set (std::string("lightSource[1].isActive"), 1);

    shaderProgramPtr->set (std::string("depthMVP[1]"), light2.depthMVP);
    shaderProgramPtr->set (std::string("shadowMap[1]"), (int) light2.shadowMapTexOnGPU);


}
```

**Résultats :**






L'image ci-contre est le résultat après l'illumination de deux lumières.

Lumière 1 :

Position = (-2*cos(app_time), 2*sin(app_time), 3.5)

Intensité = 0.5

Lumière 2 :

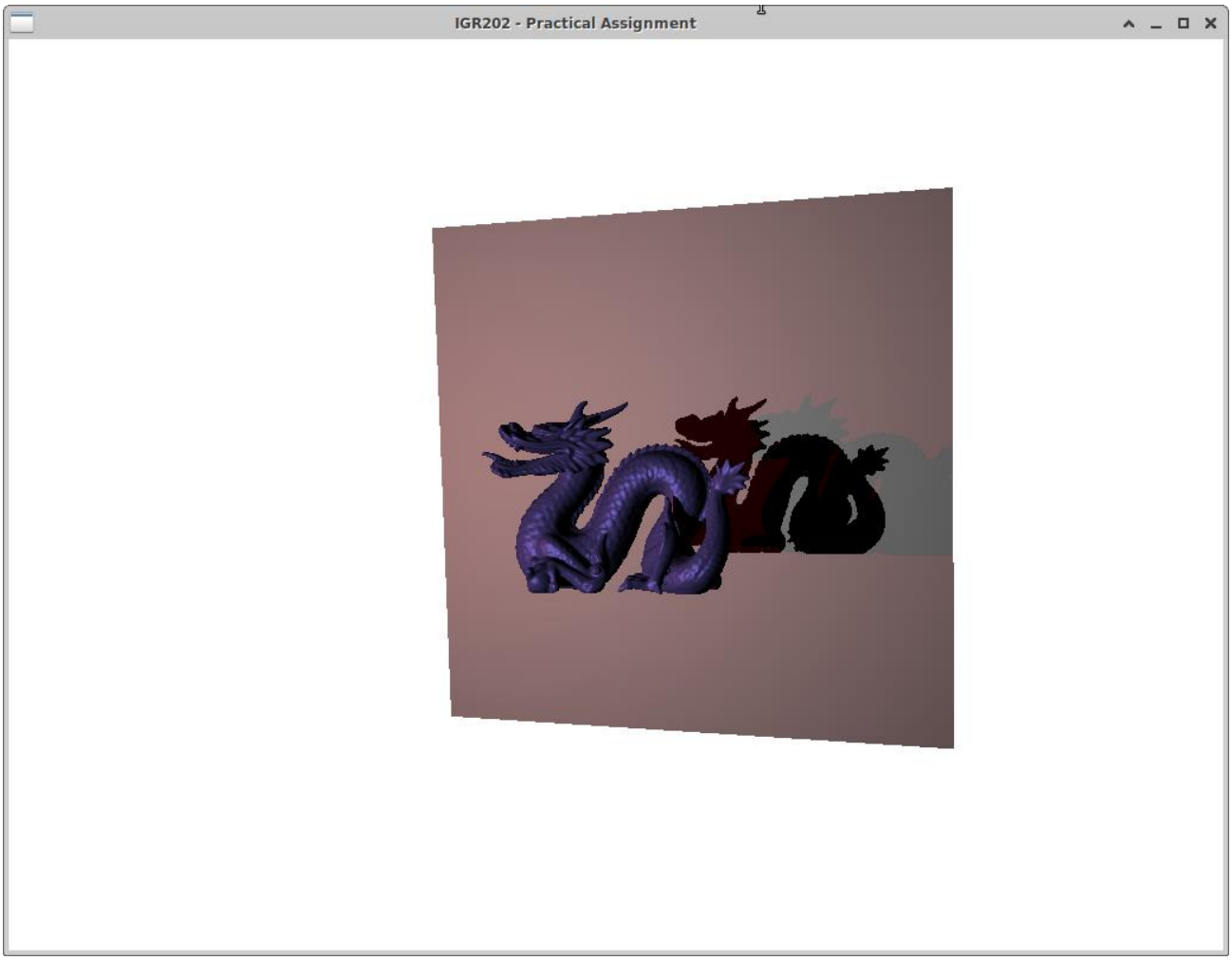
Position = (-2*cos(app_time), 2*sin(app_time), 1.5)

Intensité = 0.8

Résultat sur un autre exemple :

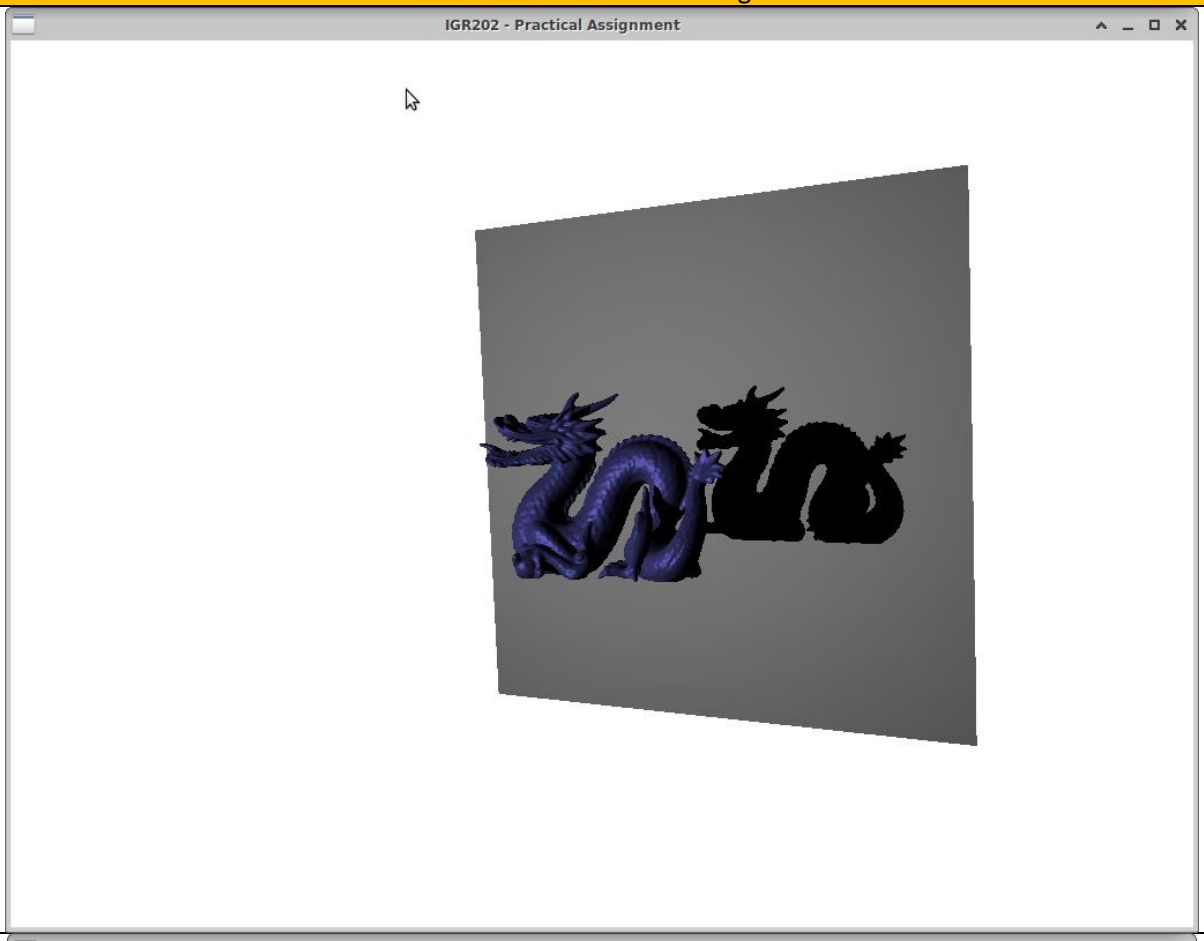
Lumière 1 : position= (-2*cos(app_time), 2*sin(app_time), 3.5), intensité(0.5f), couleur(1.0,1.0,1.0)

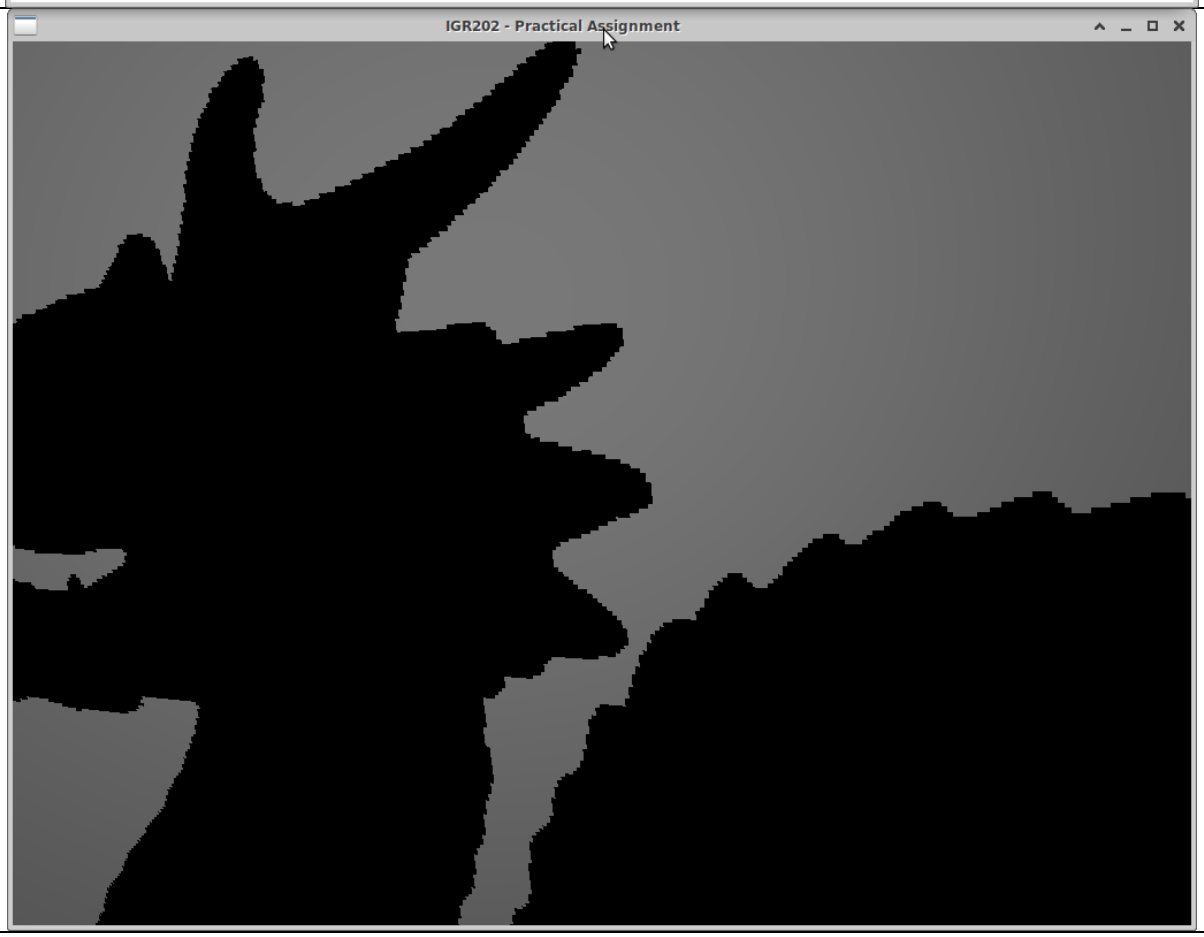
Lumière 2 : position= (-2*cos(app_time), 2*sin(app_time), 1.5), intensité(0.8f), couleur(0.2,0.0,0.0)



## 2.4 Shadow filtering (Bonus)

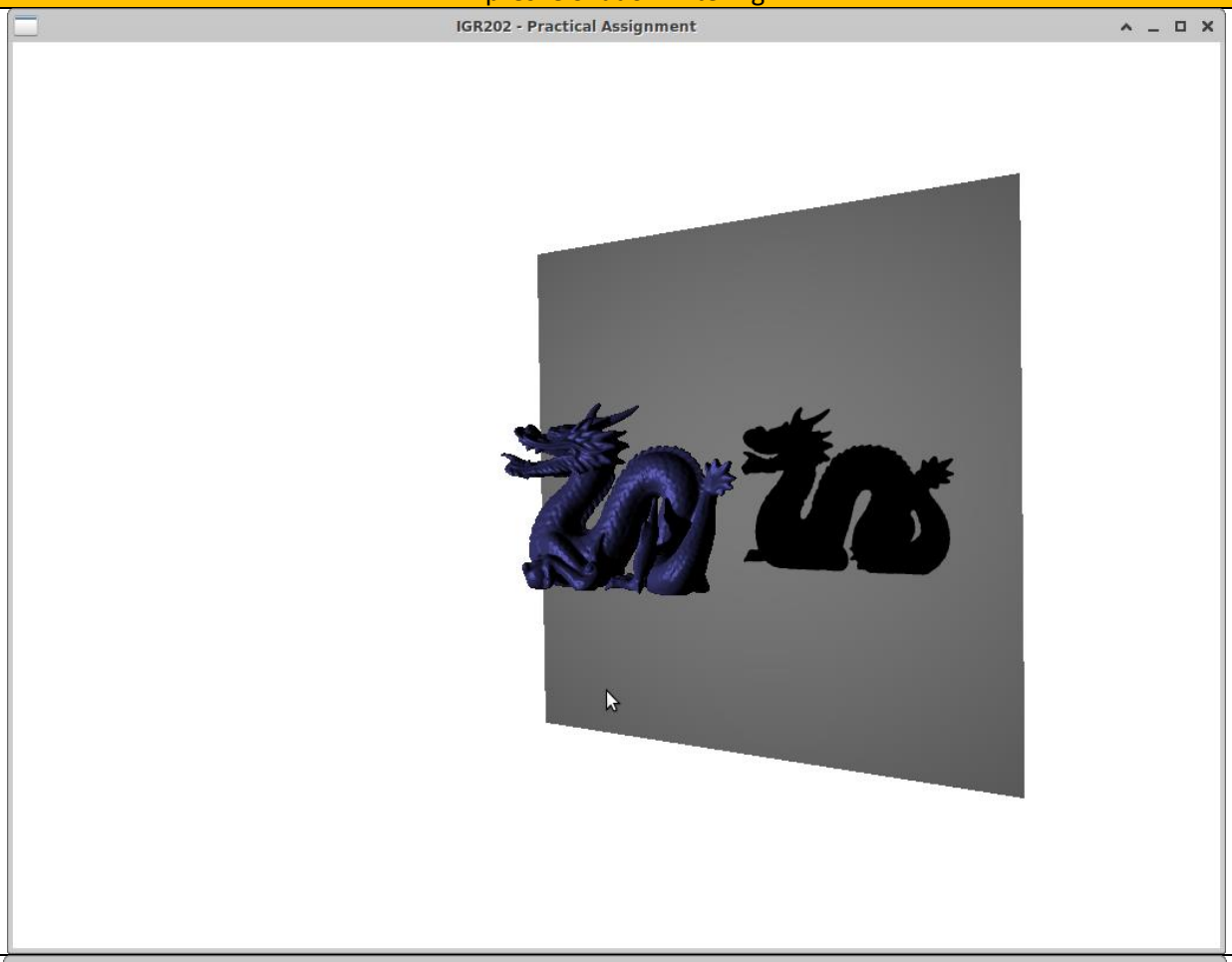
Avant le Shadow filtering

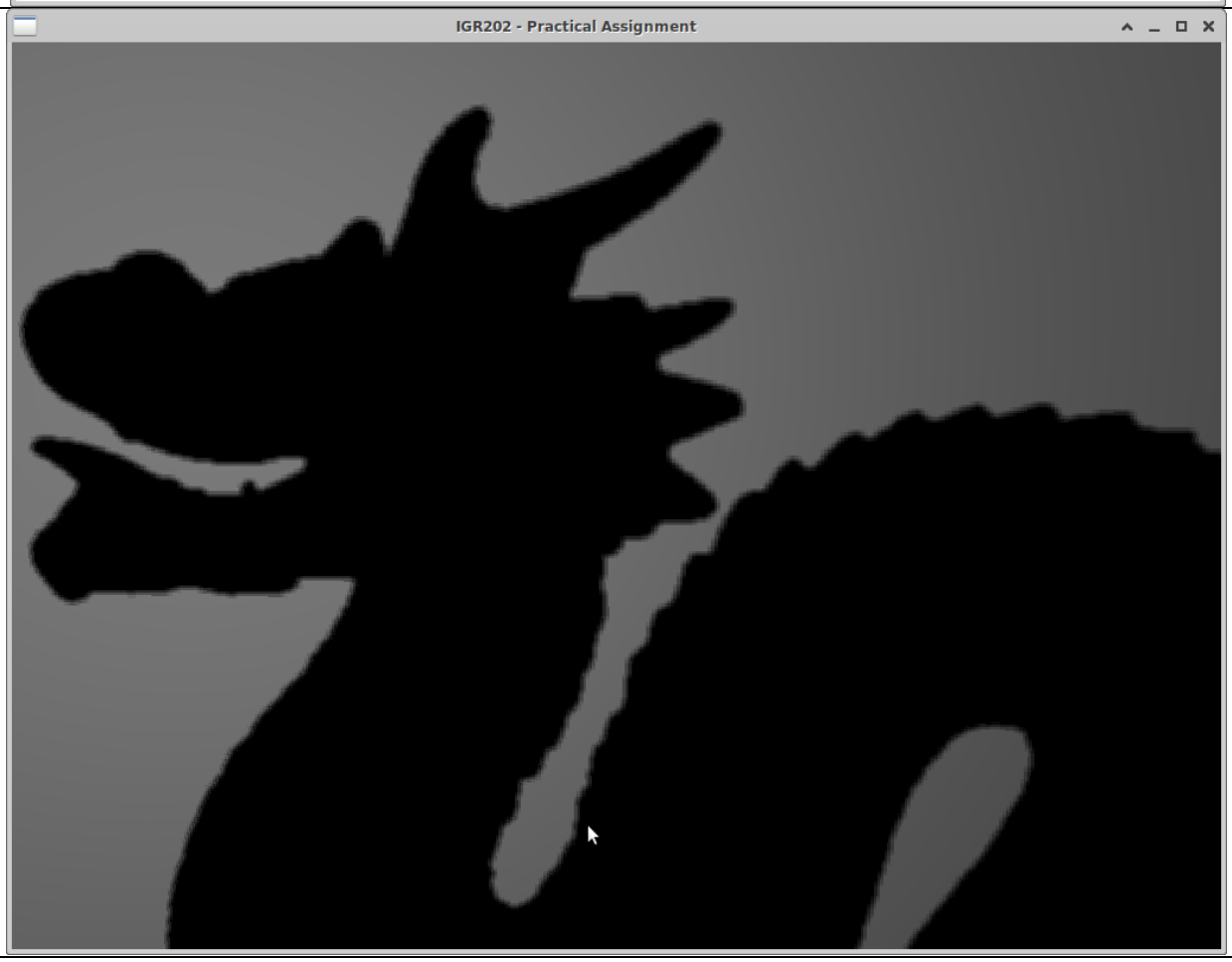



A cause de la limite de la carte de profondeur (à une résolution fixe), une profondeur pourrait s'étendre souvent sur plusieurs fragment par texel. Donc, plusieurs fragments pourraient avoir la même valeur de profondeur et donneront même valeur, ce qui produira le bord en blocs irréguliers en dent de scie.

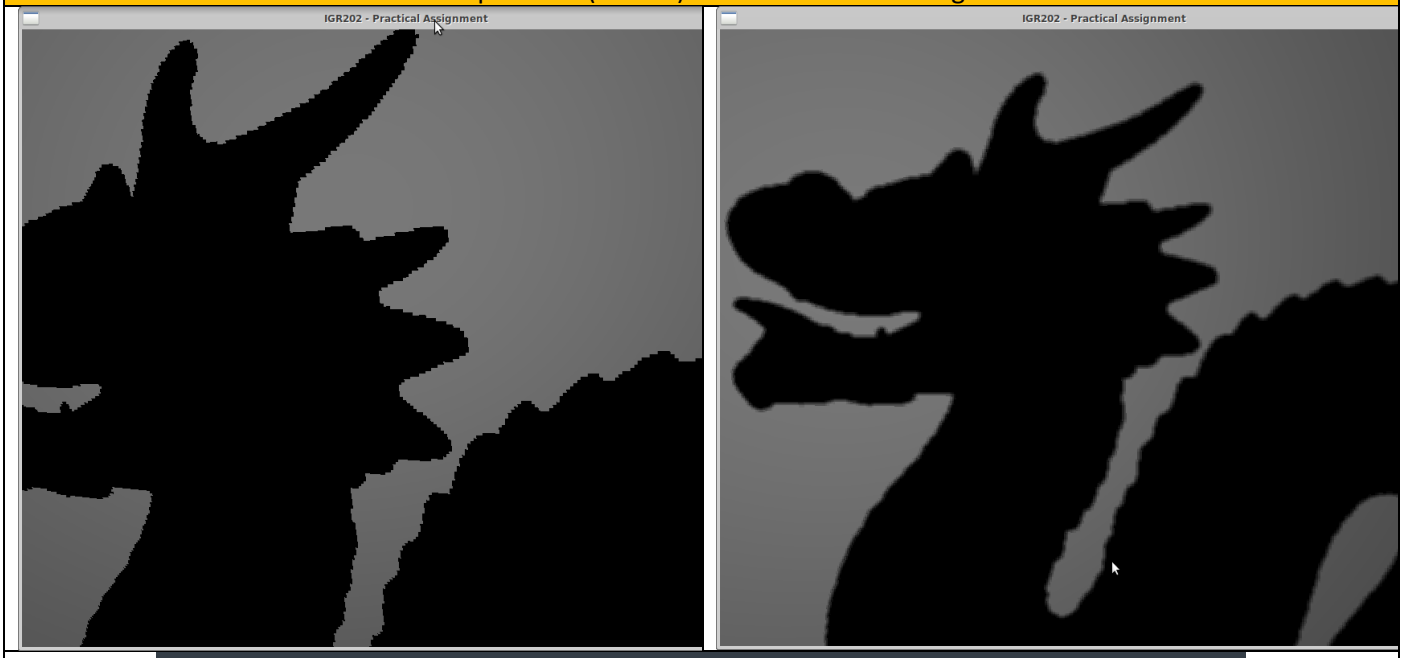
Après le Shadow filtering




En utilisant plus d’échantillons, on pourra augmenter la qualité des ombres douces.

Comparaison (zoomé) sur l'ombre du maillage




```
vec2 texelSize = 1.0 / textureSize(shadowMap[i], 0);
for(int x = -1; x <= 1; ++x)
{
    for(int y = -1; y <= 1; ++y)
    {
        float pcfDepth = texture(shadowMap[i], projCoords.xy + vec2(x, y) * texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
shadow /= 9.0;
```

Par exemple, on pourra échantillonner 9 valeurs autour de (x,y), et tester l'occlusion de l'ombre en faisant la moyenne des résultats par le nombre total d'échantillons choisis. Cela nous permet d'éviter que les fragments du bords ont les même valeurs et avoir une ombre plus douce sur la surface.