CSE 461 COMPUTER GRAPHICS

Dr. Gökhan KAYA

PROGRAMMING ASSIGNMENT 1

MUHAMMED SİNAN PEHLİVANOĞLU

1901042664

Implementation Details:

Parsing Process:

Tinyxml2 open source xml parser is used for parsing the elements of ray tracing.

Example

```
pLight = pElement->FirstChildElement("triangularlight");
while(pLight != nullptr)
{
    eResult = pLight->QueryIntAttribute("id", &id);

    lightElement = pLight->FirstChildElement("vertex1");
    str = lightElement->GetText();
    sscanf(str, "%f %f %f", &v1.x, &v1.y, &v1.z);
    //std::cerr << v1.x << v1.y << v1.z << std::endl;

    lightElement = pLight->FirstChildElement("vertex2");
    str = lightElement->GetText();
    sscanf(str, "%f %f %f", &v2.x, &v2.y, &v2.z);
    lightElement = pLight->FirstChildElement("vertex3");
    str = lightElement->GetText();
    sscanf(str, "%f %f %f", &v3.x, &v3.y, &v3.z);
    lightElement = pLight->FirstChildElement("intensity");
    str = lightElement->GetText();
    sscanf(str, "%f %f %f", &intensity.r, &intensity.g, &intensity.b);

    lightSources.push_back(new TriangularLightSource(id, intensity, v1, v2 , v3));

    pLight = pLight->NextSiblingElement("triangularlight");
}
```

All properties of triangular light component are stored in above code. All the components are stored as an child of root (scene) components.

Rendering Process:

```
void Scene::render(Image &image, const char *name) {

        auto startTime = std::chrono::high_resolution_clock::now();

    std::vector<std::thread> threads;
    std::atomic<int> nextRow(0);

    for (int i = 0; i < std::thread::hardware_concurrency(); ++i) {
        threads.emplace_back(renderRows, std::ref(image), this, std::ref(nextRow));
    }

    for (std::thread& thread : threads) {
        thread.join();
    }
     auto endTime = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(endTime - startTime);

    std::cout << "Elapsed time: " << duration.count() << " milliseconds" << std::endl;
    image.saveImage(name);
}
```

To render each row , available hardware threads are used. The number of next row to be rendered is atomic. So Each threads takes different row number to render.

```cpp
void renderRows(Image &image, Scene* scene, std::atomic<int> &nextRow) {
    Vector3f pixelColor;
    ImagePlane* imagePlane = &scene->camera.imagePlane;
    RayTracing rayTracing;
    int row;
    while ((row = nextRow.fetch_add(1)) < image.height) {
        for (int i = 0; i < (*imagePlane).nx; ++i) {
            Hit minIntersection = {INF, {}, -1};
            Ray primaryRay = rayTracing.findPrimaryRay(scene->camera, row, i);
            Hit hit = rayTracing.intersect(primaryRay, scene->objects);
            if (hit.t != INF) {
                pixelColor = rayTracing.calculateRadiance(primaryRay, scene->objects, scene->lightSources, scene->ambientLight, hit, scene->materials, scene->maxraytracedepth);
            } else {
                // background color
                pixelColor = scene->backgroundColor;
            }
            pixelColor.r = std::min(std::max(0.0f, pixelColor.r), 255.0f);
            pixelColor.g = std::min(std::max(0.0f, pixelColor.g), 255.0f);
            pixelColor.b = std::min(std::max(0.0f, pixelColor.b), 255.0f);
            image.setPixelValue(i, row, {static_cast<unsigned char>(pixelColor.r),
                                         static_cast<unsigned char>(pixelColor.g),
                                         static_cast<unsigned char>(pixelColor.b)});
        }
    }
}
```

- We find the primary ray that goes into the each pixel in image plane.
- We apply intersection for primary ray for each objects in scene.
- If intersection occurs , we find the radiance of this pixel.
- If there is no intersection , the pixel color is equal to the background color.
- Set the resulting pixel color applying clipping operation.

Calculating Radiance

```cpp
Vector3f RayTracing::calculateRadiance(const Ray & ray, const vector <Mesh*> & objects, const vector<LightSource *> &lightSources, const Vector3f & ambientLight,
                    const Hit & hit ,const  vector <Material*>& materials  , const int maxRayTraceDepth,float shadowRayEps) const{

    Vector3f color ;
    Material &material = *materials[hit.materialID-1];
    color = calculateAmbient(material, ambientLight);
    Vector3f intersectionPoint = ((ray.direction * hit.t) + ray.origin);
    Vector3f irradiance;
    Vector3f wo = normalize( ray.origin - intersectionPoint );

    for(int i = 0 ; i < lightSources.size(); ++i){
        //compute shadow

        const DirectionalLightSource* dirLight = dynamic_cast<const DirectionalLightSource*>(lightSources[i]);
        const PointLightSource* pointLight = dynamic_cast<const PointLightSource*>(lightSources[i]);
        const AreaLightSource* areaLight = dynamic_cast<const AreaLightSource*>(lightSources[i]);


        Vector3f lightDirection;
        if (dirLight) {

            lightDirection = dirLight->getDirection();
        } else if (pointLight) {
            lightDirection = pointLight->getDirection(intersectionPoint);
        }
        else if(areaLight){

            lightDirection = areaLight->getDirection();

        }
```

- First calculate ambient illumination contribution.
- Since there can be more than one light sources as well as different type of light source, we need to calculate radiance for each light source.
- We determine the type of light source. Base class of different type of light sources is LightSource.

- We determine the direction of light . Each type of light sources's direction is calculated different.

```
 //get normalized light direction
 Vector3f normalizedLightDirection = normalize(lightDirection);


//Apply shadow offset
 Ray shadowRay;
 Vector3f shadowSurfaceOffset =  normalizedLightDirection* shadowRayEps;
 shadowRay.direction = normalizedLightDirection;
 shadowRay.origin = intersectionPoint + shadowSurfaceOffset ;
```

- We find the normalized light direction to be used for finding the shadow ray.
- Find the shadow ray to apply shadow intersection test.

```
// there is no intersect
//for area lights also calculate intersection between area light and shadow ray
if (isShadowReachToLight(shadowRay, lightSources[i], objects )){
    //calculate diffuse
    //calculate specular
    //calculate mirror

    if(pointLight){

        irradiance = pointLight->computeLightContribution(intersectionPoint);

    }
    else if(areaLight){

        irradiance = areaLight->computeLightContribution(intersectionPoint);

    }
    Vector3f diffuseIllimunation = this->calculateDiffuseShading(normalizedLightDirection, hit.normal,
                    material, irradiance);

    color += diffuseIllimunation;

    //specular

    Vector3f normalizedHalfVector = normalize(normalizedLightDirection + wo );

    color+= calculateSpecularShading(hit.normal,normalizedHalfVector,irradiance, material);
```

- If shadow ray coming out from the intersection point reaches to the light source , then there is no shadow in that point.
- We calculate illuminance of different type of light sources. In our scene there are two types of light sources. This code can be expanded for another type of light source.
- If the object is in shadow for desired light source , check other light sources.
- If there is no shadow , then calculate diffuse illumination contributions from diffuse reflection.
- Then find the normalized half vector that is used for finding the specular shading.
- Add the specular illimunation contribution to the color of the pixel.

```
// compute mirror effect

if (material.hasMirror()) {
    // Calculate reflected direction
    Vector3f reflectedDirection = (reflect(wo, hit.normal));

    // Trace reflected ray recursively
    Ray reflectedRay;
    reflectedRay.origin = intersectionPoint + (reflectedDirection * shadowRayEps);
    reflectedRay.direction = reflectedDirection;
    Hit reflectedHit = intersect(reflectedRay, objects);

    // Compute radiance recursively
    if (reflectedHit.t != INF && maxRayTraceDepth > 0) {
        Vector3f reflectedRadiance = calculateRadiance(reflectedRay, objects, lightSources, ambientLight, reflectedHit, materials, maxRayTraceDepth - 1, shadowRayEps);

        Vector3f reflectedColor;

        reflectedColor.r = reflectedRadiance.r * material.mirrorReflectance.r;
        reflectedColor.g = reflectedRadiance.g * material.mirrorReflectance.g;
        reflectedColor.b = reflectedRadiance.b * material.mirrorReflectance.b;
        color += reflectedColor;
    }
}
```

- If the material that locates in hit point mirrorreflactance property, then calculate mirror reflectance recursively.
- Find the reflected ray.
- Apply intersection test for reflected ray to all objects.
- If reflected ray hits some point of objects then add the reflection illumination contribution to the actual color of the pixel.
- Do this recursively until the max ray tracing depth is reached , or ray does not intersect any object in scene.

- **Ambient Shading**

```
Vector3f RayTracing::calculateAmbient(const Material& material, const Vector3f& ambientLight ) const{
    return {material.ambient.r * ambientLight.r,
            material.ambient.g * ambientLight.g,
            material.ambient.b * ambientLight.b};
}
```

- ## Diffuse Shading

```
Vector3f RayTracing::calculateDiffuseShading(const Vector3f & wi, const Vector3f& surfaceNormal ,const Material & material, Vector3f & intensity )const{


    float cos_teta = dotProduct(surfaceNormal,wi);
    if(cos_teta < 0) cos_teta = 0;

    Vector3f diffuse;
    diffuse.r = material.diffuse.r * (intensity.r*cos_teta);
    diffuse.g = material.diffuse.g * (intensity.g*cos_teta);
    diffuse.b = material.diffuse.b * (intensity.b*cos_teta);

    return diffuse;

}
```

Find the geometric term . If the geometric term is negative then there is no diffuse reflection.

- ## Specular Shading

```
Vector3f RayTracing::calculateSpecularShading(const Vector3f & normal, const Vector3f& halfway , const Vector3f & irradiance , const Material & material) const{

    Vector3f specularIllumination;
    float cosphi = dotProduct(normal, halfway);
    if(cosphi <0) cosphi = 0;


    specularIllumination = material.specular*  pow(cosphi, material.phongExponent);
    specularIllumination.r *= irradiance.r;
    specularIllumination.g *= irradiance.g;
    specularIllumination.b *= irradiance.b;



    return specularIllumination;

}
```

- Find the cosphi (cosine of angle between halfway vector and normal vector object).
- Then apply specular illumination contributions.

COMPLEXITY:

- **Primary Rays Generation**

For each pixel, a viewing ray needs to be computed. This involves iterating over each pixel, which contributes $O(N)$, where N is the total number of pixels.

- **Intersection Tests**

For each primary ray, intersection tests are performed with each object in the scene. If there are M objects in the scene, this step contributes $O(M)$ per pixel.

- **Shadow Ray Generation**

For each intersection point x (where the viewing ray intersects an object), shadow rays need to be computed to each light source.
For each light source, shadow rays are generated, contributing $O(L)$, where L is the number of light sources.

- **Shadow Ray Intersection Tests**

For each shadow ray, intersection tests need to be performed with each object in the scene to check for shadows.
If there are P objects in the scene, and 1 shadow rays per light source, this step contributes $O(P)$ per light source.

- **Shading Computations**

If a point is not in shadow, diffuse and specular shading contributions are computed for each light source.
This involves basic arithmetic operations and is generally considered constant time. O(1)

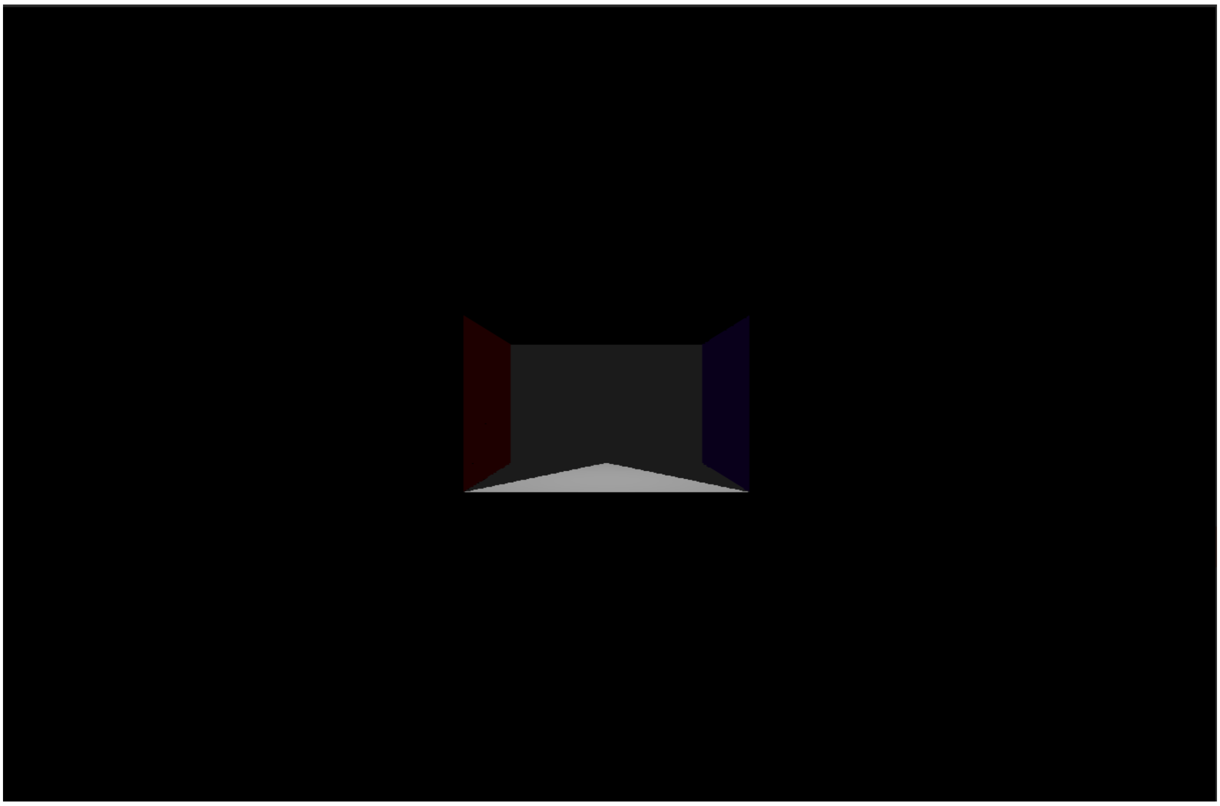
- **Perfect Mirror Reflection Computation**

All the process is done as count as max ray tracedepth recursively.

Using available hardware thread , we reduce the computation nearly as much as (computation / thread size).

RESULTS:

- **Scene 1**



Elapsed time: 64 milliseconds

There is one triangular light source in scene. Perpendicular to surface.

```xml
<lights>
<ambientlight>20 20 20</ambientlight>
<triangularlight id="1">

<vertex1> -10 20 10 </vertex1>
<vertex3> 0 20 -10 </vertex3>
<vertex2> 10 20 10 </vertex2>
<intensity> 62800 62800 62800 </intensity>
</triangularlight>
</lights>
```

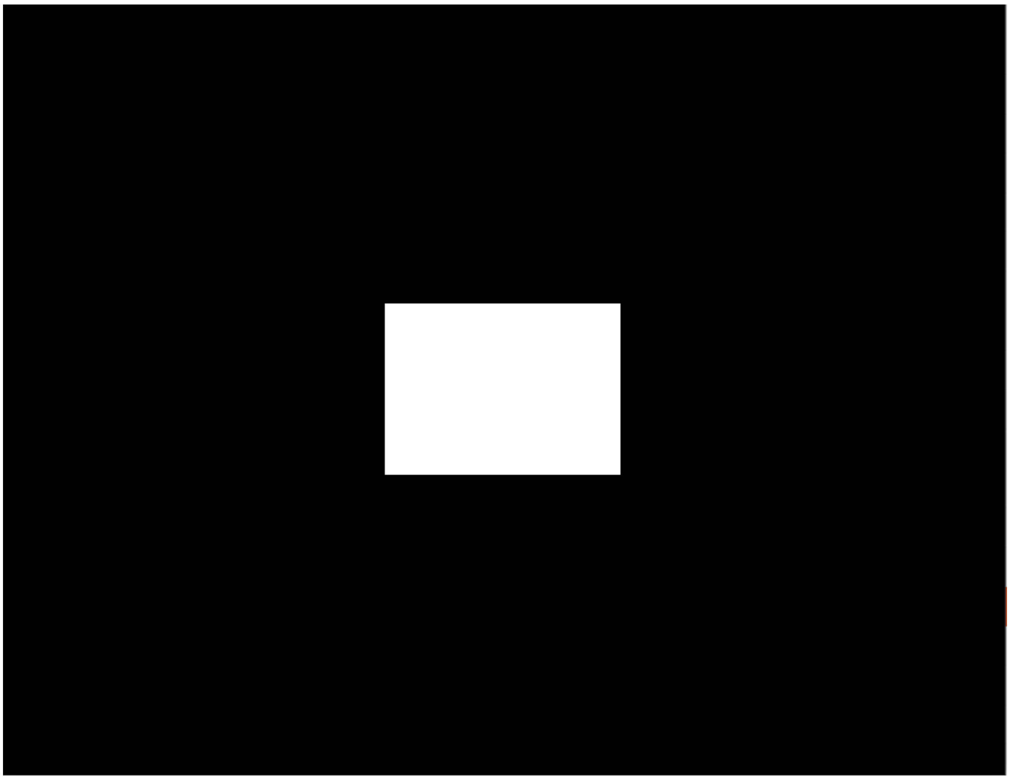
There are 8 face and 4 object. Each object has 2 face.

```
<objects>
    <mesh id="1">
        <material>1</material>
        <faces>
            1 2 6
            6 5 1
        </faces>
    </mesh>
    <mesh id="2">
        <material>1</material>
        <faces>
            5 6 7
            7 8 5
        </faces>
    </mesh>

    <mesh id="3">
        <material>2</material>
        <faces>
            8 4 1
            8 1 5
        </faces>
    </mesh>
    <mesh id="4">
        <material>3</material>
        <faces>
            2 3 7
            2 7 6
        </faces>
    </mesh>

</objects>
```

- **Scene 2**



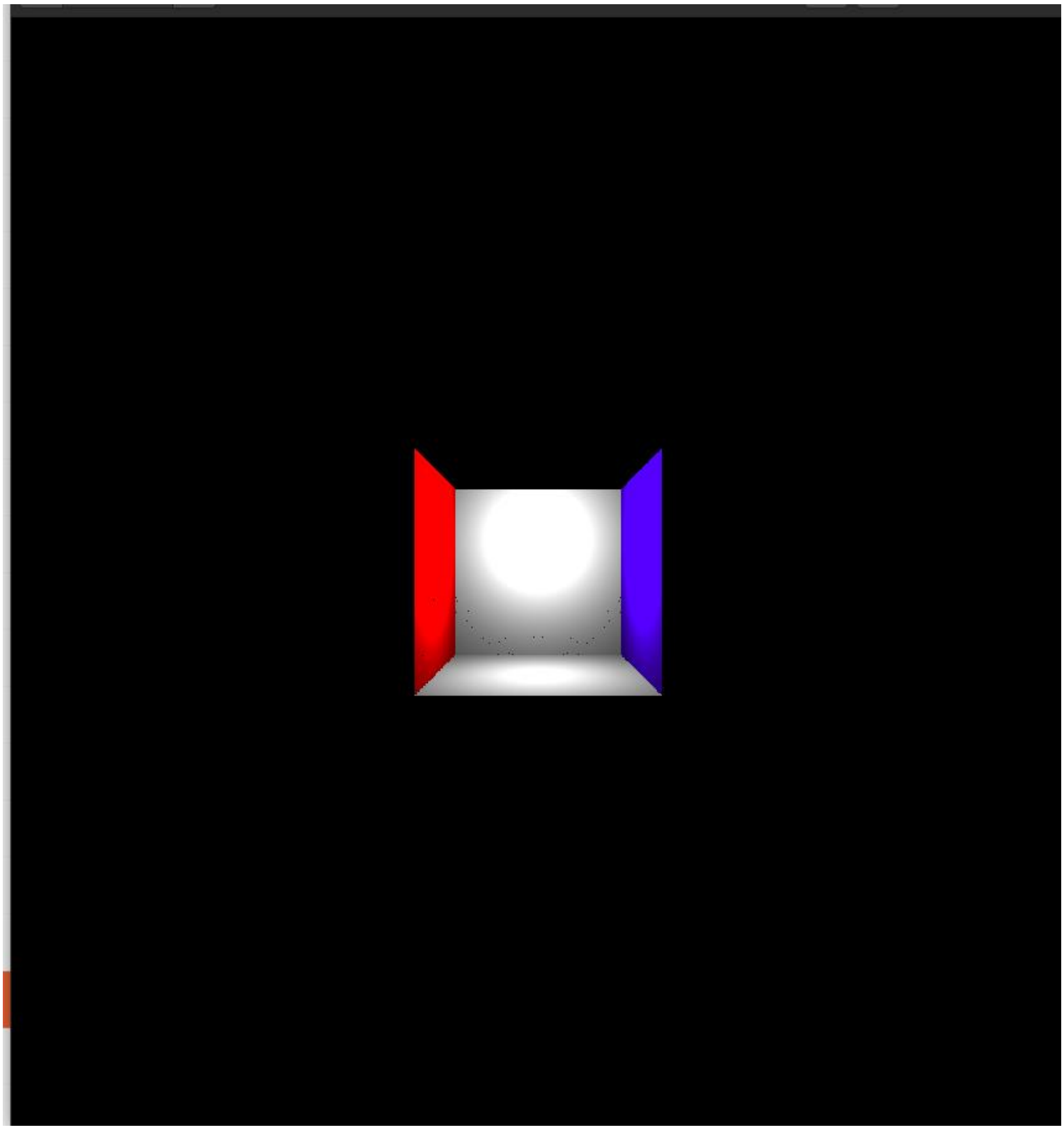
Elapsed time: 29 milliseconds

There are only two face.

```
<lights>
3<ambientlight>25 25 25</ambientlight>
<pointlight id="1">
<position>0 0 0 </position>
<intensity>1000 1000 1000</intensity>
</pointlight>
```

Lights of the scene.

- **Scene 3**

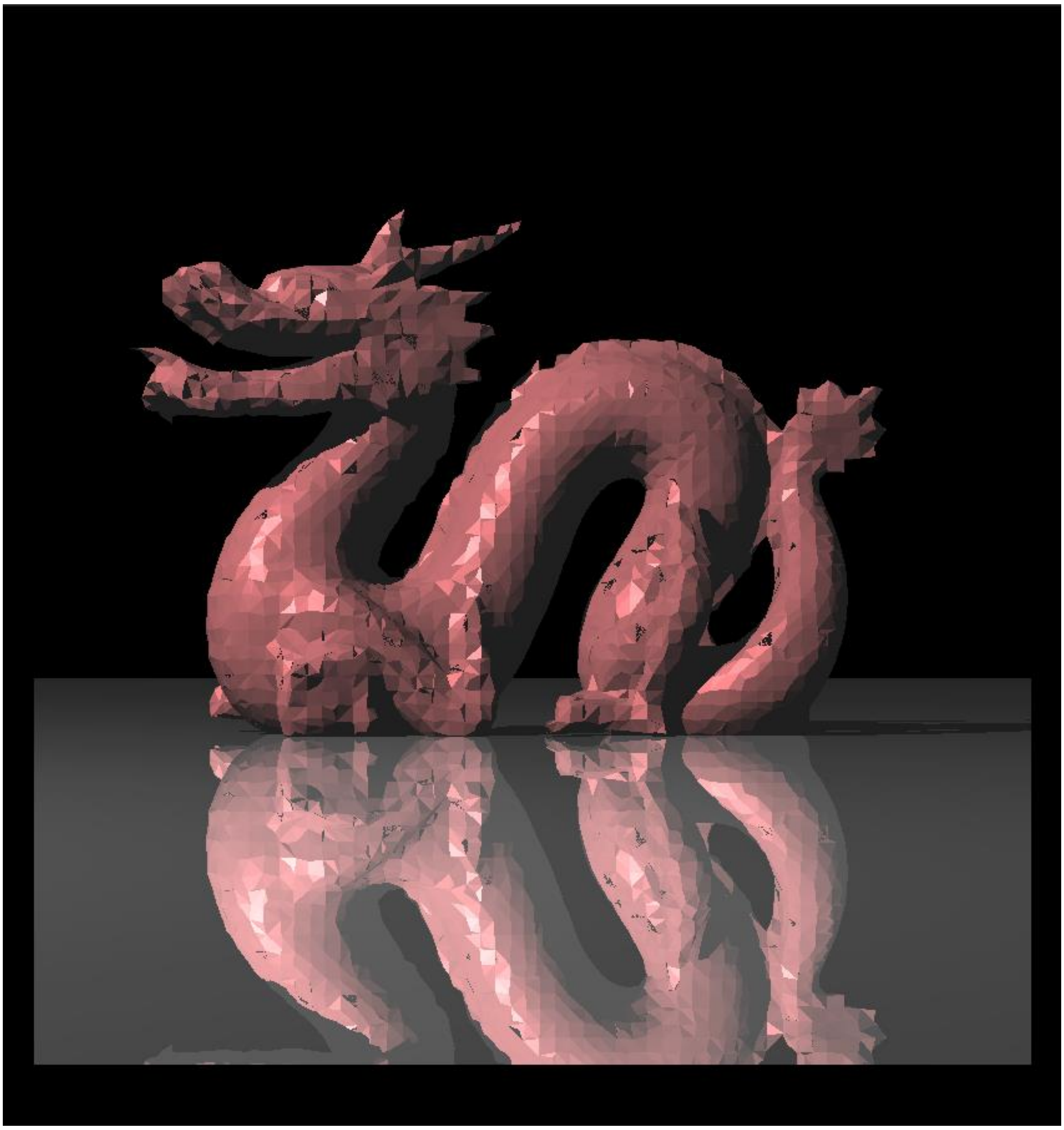


Elapsed time: 57 milliseconds

One Point Light. Same as figuare 1 instead point light is used for.

- **Scene 4**

Far more complicated dragon figuire. I found it from METU Computer Graphics Homework resources.



Elapsed time: 165672 milliseconds

It has 11073 face. One point light source.
2 object. Surface and dragon.