Pinhole Camera

In this part of the project, I implemented an arbitrary pinhole camera, made the integration between my renderer and the Assimp library to be able to import external 3D models and also added the color attribute on the primitive class, so that it was able to render images with a random RGB color inside each primitive.

Through the pinhole camera (derived from the Camera class), the renderer is now able to show images in perspective and with focus on everything that is shown. It's header file can be seen below:

pinhole_camera.h

```cpp
#ifndef PINHOLE_CAMERA_H_
#define PINHOLE_CAMERA_H_

#include "camera.h"
#include "ray.h"

class PinholeCamera : public Camera
{
public:

    PinholeCamera( void );

    PinholeCamera( const float min_x,
                   const float max_x,
                   const float min_y,
                   const float max_y,
                   const float focal_distance,
                   const glm::ivec2 &resolution,
                   const glm::vec3 &position,
                   const glm::vec3 &up_vector,
                   const glm::vec3 &look_at );

    Ray getWorldSpaceRay( const glm::vec2 &pixel_coord ) const;

    float focal_distance_;

    float min_x_;

    float max_x_;

    float min_y_;

    float max_y_;

};

```

pinhole_camera.h

And next, it’s implementation:

pinhole_camera.h | pinhole_camera.cpp

```cpp
#include "pinhole_camera.h"

PinholeCamera::PinholeCamera( void )
{}

PinholeCamera::PinholeCamera( const float min_x,
                              const float max_x,
                              const float min_y,
                              const float max_y,
                              const float focal_distance,
                              const glm::ivec2 &resolution,
                              const glm::vec3 &position,
                              const glm::vec3 &up_vector,
                              const glm::vec3 &look_at ) :
        Camera::Camera{ resolution,
                        position,
                        up_vector,
                        look_at },

        focal_distance_{ focal_distance },
        min_x_{ min_x },
        max_x_{ max_x },
        min_y_{ min_y },
        max_y_{ max_y }
{}

Ray PinholeCamera::getWorldSpaceRay( const glm::vec2 &pixel_coord ) const
{
    glm::vec3 a(max_x_ - min_x_, 0, 0);
    glm::vec3 b(0, max_y_ - min_y_, 0);
    glm::vec3 c(min_x_, min_y_, -focal_distance_);

    float u = (pixel_coord.x + 0.5) / resolution_.x;
    float v = (pixel_coord.y + 0.5) / resolution_.y;

    glm::vec3 s = c + u * a + v * b;

    return Ray{ position_,
                glm::normalize( onb_.getBasisMatrix() * s ) };
}
```

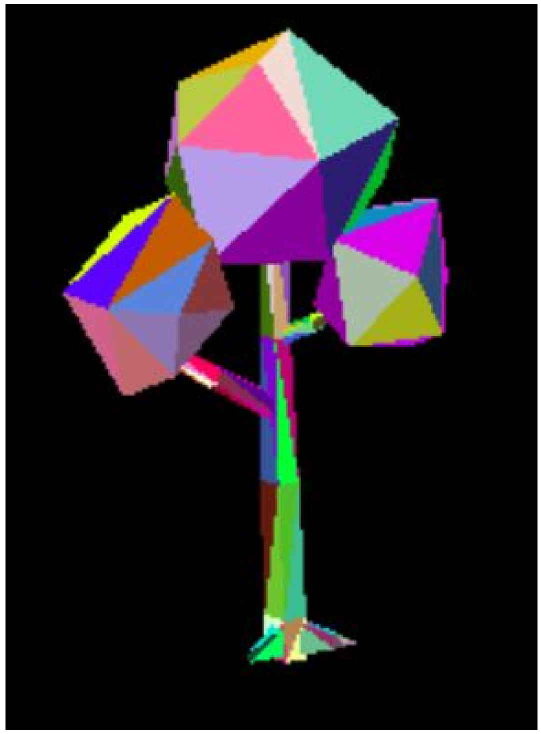
pinhole_camera.cpp

The use of the Assimp library was made through the function “load” presented on the Scene class, in which were also generated three random numbers to be used as RGB color to each triangle on the model. The way it was done can be seen below:

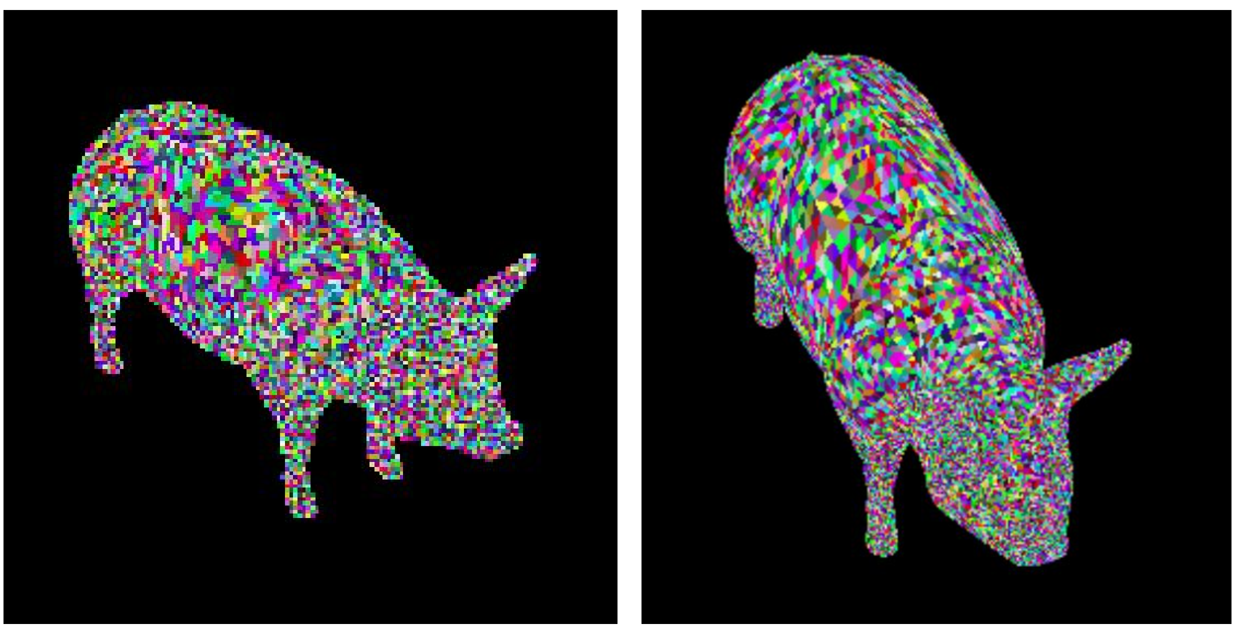
```cpp
Assimp::Importer importer;

const aiScene *scene = importer.ReadFile("/home/ryuugami/Documents/models/tree.obj",
                                         aiProcess_CalcTangentSpace |
                                         aiProcess_Triangulate |
                                         aiProcess_JoinIdenticalVertices |
                                         aiProcess_SortByPType);

for (unsigned int j = 0; j < scene->mNumMeshes; j++) {
    auto mesh = scene->mMeshes[j];

    for (unsigned int i = 0; i < mesh->mNumFaces; i++) {
        auto face = mesh->mFaces[i];

        auto v1 = mesh->mVertices[face.mIndices[0]];
        auto v2 = mesh->mVertices[face.mIndices[1]];
        auto v3 = mesh->mVertices[face.mIndices[2]];

        float r, g, b;
        r = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        b = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);
        g = static_cast<float>(rand()) / static_cast<float>(RAND_MAX);

        Triangle *triangle = new Triangle(glm::vec3(v1.x, v1.y, v1.z),
                                        glm::vec3(v2.x, v2.y, v2.z),
                                        glm::vec3(v3.x, v3.y, v3.z));

        triangle->color_ = glm::vec3(r, g, b);
        primitives_.push_back( Primitive::PrimitiveUniquePtr(triangle));
    }
}
```

Assimp library use

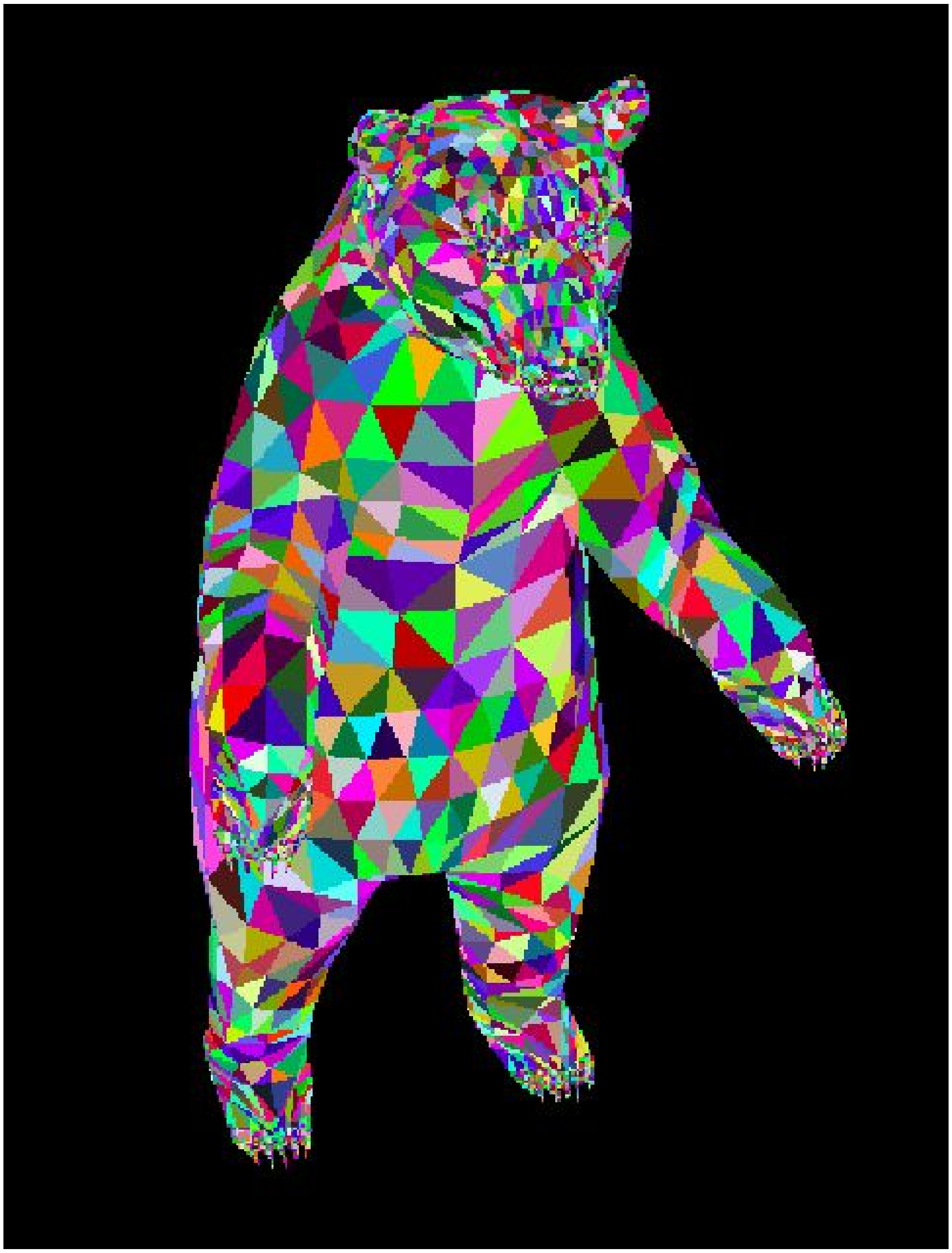
Now we can see some of the results obtained after all the new implementations:



tree.obj



pig.obj



Bear.obj

As it can be seen, the tree was the simplest model, taking 790.401 ms to be rendered. The bear was the biggest one, but it had a fair amount of primitives, so, it took 68298.9 ms. The pig has a lot more primitives than the previous ones, as it can be noticed through the number of colors, and it took 282971.0 ms of rendering time.