Report for Assignment 3

Global Environment Lighting
Jiajia Fan ---2016/12/13
fanjj@shanghaitech.edu.cn

Structure

This Program is developed based on assignment 2, i.e. basic ray tracer. And it is consisted of three main parts: environment light mapping, KD-Tree for mesh rendering, and sample the light map with quad-tree.

Class Name	Purpose
MeshObject	Create a window and set the buttons
CubeLight	Create things needed in a scene and render the image
SceneRenderer	Determine the image plane and objects and light sources

The core class for environment lighting is CubeLight and MeshObject. And in SceneRenderer Mont Carlo integration is used to calculate radiance.

Detail in Core Class

CubeLight

Quad-Tree & Node:

```
////////////////////// QuadTree for Sampling lights ////////////////////
#define QUAD_VARIANCE_THRESHOLD 1000
#define QUAD_SAMPLE_SEED 100
#define QUAD_REGION_LIMIT 10
class QuadNode
{
public:
    QuadNode(int in_region[2][2], int in_degree);
    ~QuadNode();
    void IsLeaf(bool in_set){ qtn_isleaf_ = in_set; }
    bool IsLeaf(){ return qtn_isleaf_; }
    void GrowChidren();
    QuadNode* GetChildren(int i){ return qtn_child_[i]; }
    int         qtn_region_[2][2];
private:
    int         qtn_degree_;
    bool        qtn_isleaf_;
    QuadNode*   qtn_child_[4];
    // ...
};
class QuadTree
{
public:
    QuadTree(QVector<QVector<lw::Color> >* in_map);
    ~QuadTree();
    QVector<int>                    sample_list_;

private:
    QVector<QVector<lw::Color> >*   qt_map_;
    QuadNode*                       qt_root_;
    QVector<QVector<double> >       radiance_map_;
    void buildTree();
    void splitNode(QuadNode* in_node);
    void doSampling(QuadNode* in_node, double in_max);
    void updateRadianceMap();
    double getVariance(QuadNode* in_node, double & out_max);
};
////////////////////////////////////////////////////////////////////////
```

Variant

```
private:
    int                                 c_radius_;
    //lw::Image*                            c_cubemap_;
    QVector<QVector<lw::Color> >        c_colormap_[6];
    int                                 c_scalar_;
    QVector3D                           c_center_;
    PlaneObject*                        c_planes_[6];
    QuadTree*                           c_sample_[6];
```

Functions

```
public:
    CubeLight(int in_r = 2560);
    ~CubeLight();

    // lighting source parameters
    void SetRadius(int in_r);
    int GetRadius(){ return c_radius_; }

    // obtain lighting source parameters
    QVector3D LightCenter(){ return c_center_; };

    // intersection
    bool RayHitTest(Ray in_ray, IntersectionPoint& out_point);
    // diffuse lighting
    QVector<QVector3D> GetSampleList(QVector<QVector3D>* out_color);
```

```
    void initLightMap();
    void updatePlanes();
    QVector3D getWorldPos(int i, int j, FACE_DIR in_face);
    QVector3D getColorFromMap(int i, int j, FACE_DIR in_face);
    QVector3D getColorFromMap(QVector3D in_p, FACE_DIR in_face);
};
```

Core Codes

Sampling:

```
void QuadTree::splitNode(QuadNode* in_node){
    double max_rad;
    double it_var = getVariance(qt_root_, max_rad);
    bool is_limit = (in_node->qtn_region_[0][1] - in_node->qtn_region_[0][1]) / 2 < QUAD_REGION_LIMIT;
    is_limit = is_limit || (in_node->qtn_region_[0][1] - in_node->qtn_region_[0][1]) / 2 < QUAD_REGION_LIMIT;
    if (it_var < QUAD_VARIANCE_THRESHOLD || is_limit)
    {
        in_node->IsLeaf(true);
        // do sampling
        doSampling(in_node, max_rad);
        return;
    }
    in_node->GrowChidren();
    for (int i = 0; i < 4; i++)
    {
        splitNode(in_node->GetChildren(i));
    }
}
```

```
void QuadTree::doSampling(QuadNode* in_node, double in_max){
    int w_step = in_node->qtn_region_[0][1] - in_node->qtn_region_[0][0],
        h_step = in_node->qtn_region_[1][1] - in_node->qtn_region_[1][0];
    int scalar_step = in_max / QUAD_SAMPLE_SEED;
    scalar_step = max_ab(scalar_step, 1);
    w_step /= scalar_step;
    h_step /= scalar_step;
    for (int i = in_node->qtn_region_[0][0]; i < in_node->qtn_region_[0][1]; i += w_step)
    {
        for (int j = in_node->qtn_region_[1][0]; j < in_node->qtn_region_[1][1]; j += h_step)
```

CubeLight mapping:

```
void CubeLight::initLightMap(){
    lw::Image cubemap;
    cubemap.load("Resources/cubemaps/grace_cross.hdr");
    int it_width = cubemap.width() / 3;
    c_scalar_ = c_radius_ / it_width;

    // initialize the map size
    for (int i = 0; i < 6; i++)
    {
        c_colormap_[i].resize(it_width);
        // ...
    }
    // set colors for every plane
    for (int w_step = 0; w_step < it_width; w_step++)
    {
        for (int h_step = 0; h_step < it_width; h_step++)
        {
            // top: x,z
            c_colormap_[TOP][w_step].push_back(cubemap.pixel(it_width + w_step, h_step));
            // left: y,z
            c_colormap_[LEFT][w_step].push_back(cubemap.pixel(w_step, it_width * 2 - h_step));
            // back: x,y
            c_colormap_[BACK][w_step].push_back(cubemap.pixel(it_width + w_step, it_width * 2 - h_step));
            // right: y,z
            c_colormap_[RIGHT][w_step].push_back(cubemap.pixel(it_width * 3 - 1 - w_step, it_width * 2 - h_step));
            // down: x,z
            c_colormap_[DOWN][w_step].push_back(cubemap.pixel(it_width + w_step, it_width * 3 - h_step));
            // front: x,y
            c_colormap_[FRONT][w_step].push_back(cubemap.pixel(it_width + w_step, it_width * 3 + h_step));
        }
    }
}
```

MeshObject

KD-Tree & Node:

```
#define KDTREE_FACE_NUM_MIN 20
class KDNode
{
public:
    KDNode(AABB in_ab, QVector<int>* in_faceidx, int in_d = 0){ ... }
    ~KDNode(){};

    QVector<int>* GetFaceList(){ return kdn_faceidx_; }
    void SetAABB(AABB in_ab){ kdn_ab_ = in_ab; }
    AABB GetAABB(){ return kdn_ab_; }
    int GetDegree(){ return kdn_degree_; }
    bool IsLeaf(){ return kdn_isleaf_; }
    bool IsLeaf(bool to_set ){ ... }
    KDNode* GetChild(int i){ return kdn_child_[i]; }
    void SetChild(KDNode* in_node, int i){ kdn_child_[i] = in_node; }
private:
    int             kdn_degree_;
    AABB            kdn_ab_;
    KDNode*         kdn_child_[2];
    bool            kdn_isleaf_;
    QVector<int>*   kdn_faceidx_;
};
```

```
class KDTree
{
public:
    KDTree(QVector3D in_ct, objLoader* in_obj);
    ~KDTree();

    void SetCenter(QVector3D in_ct);
    void BuildTree();
    void RayHitLeafNode(Ray in_ray);
    QVector<KDNode*> GetHitNodes(){ return kd_hitlist_; }
private:
    QVector3D           kd_center_;
    AABB                kd_root_ab_;
    objLoader*          kd_objdata_;
    KDNode*             kd_root_;
    QVector<KDNode*>    kd_hitlist_;
    //QVector<QVector3D>*   kd_faces_;

    AABB updateAABB();
    void initRoot();
    void SplitNode(KDNode* in_node);
    AABB GetFaceAABB(obj_face* in_face);
    void hitNode(Ray in_ray, KDNode* in_node);
    void Unify(float size);
};
```

Variant

```
private:
    objLoader*          m_objdata_;
    KDTree*             m_kdtree_;
    QVector3D           m_center_;
```

Functions

```
public:
    MeshObject(QVector3D in_ct = QVector3D(0,0,50), QVector3D in_color = QVector3D(200, 100, 100));
    ~MeshObject();
    // intersection operators
    void SetPosition(QVector3D in_ct){ m_center_ = in_ct; m_kdtree_->SetCenter(in_ct); };
    bool RayHitTest(Ray in_ray, IntersectPoint& out_hit);
```

```
    bool RayHitFace(obj_face* in_face, Ray in_ray, IntersectPoint& out_hit);
    void initObj();
```

Core Codes

Building KD-Tree:

```
void KDTree::SplitNode(KDNode* in_node){
    QVector<int>* n_faces = in_node->GetFaceList();
    if (n_faces == NULL)
    {
        in_node->IsLeaf(true);
        return;
    }
    if (n_faces->size() <= KDTREE_FACE_NUM_MIN)
    {
        in_node->IsLeaf(true);
        return;
    }
    AABB n_ab = in_node->GetAABB();
    int n_d = in_node->GetDegree();
    // splitting
    AABB child_ab[2];
    child_ab[0] = child_ab[1] = n_ab;
    QVector<int>* child_faceidx[2];
    child_faceidx[0] = new QVector<int>;
    child_faceidx[1] = new QVector<int>;
    // axis-aligned splitting X-Y-Z
    int split_axis = n_d % 3;
    // split the Bounding-Box
    double split_value = (n_ab.BB_Max[split_axis] + n_ab.BB_Min[split_axis]) / 2;
    child_ab[0].BB_Min[split_axis] = split_value;
    child_ab[1].BB_Max[split_axis] = split_value;
```

```
// split the faces
for (QVector<int>::iterator iface = n_faces->begin(); iface != n_faces->end(); iface++)
{
    obj_face* cur_face = kd_objdata_->faceList[(*iface)];
    if (IsCollidAABB(child_ab[0], GetFaceAABB(cur_face)))
    {
        child_faceidx[0]->push_back(*iface);
    }
    if (IsCollidAABB(child_ab[1], GetFaceAABB(cur_face)))
    {
        child_faceidx[1]->push_back(*iface);
    }
}

// ...

// restore into children nodes
in_node->SetChild(new KDNode(child_ab[0], child_faceidx[0], n_d + 1), 0);
in_node->SetChild(new KDNode(child_ab[1], child_faceidx[1], n_d + 1), 1);
// delete father's faces
n_faces->clear();
// ...
SplitNode(in_node->GetChild(0));
SplitNode(in_node->GetChild(1));
return;
```

Intersection:

```
void KDTree::hitNode(Ray in_ray, KDNode* in_node){
    // ...
    if (in_node->GetAABB().RayHitTest(in_ray))
    {
        if (in_node->IsLeaf())
        {
            kd_hitlist_.push_back(in_node);
            return;
        }
        else
        {
            hitNode(in_ray, in_node->GetChild(0));
            hitNode(in_ray, in_node->GetChild(1));
        }
    }
}
```

```
bool IsInTriangle(const QVector3D A, const QVector3D B, const QVector3D C, const QVector3D D)
{
    // #1：重心法求交点是否在三角面片内部
    QVector3D v0 = C - A;
    QVector3D v1 = B - A;
    QVector3D v2 = D - A;

    float dot00 = QVector3D::dotProduct(v0, v0);
    float dot01 = QVector3D::dotProduct(v0, v1);
    float dot02 = QVector3D::dotProduct(v0, v2);
    float dot11 = QVector3D::dotProduct(v1, v1);
    float dot12 = QVector3D::dotProduct(v1, v2);

    float inverDeno = 1 / (dot00 * dot11 - dot01 * dot01);
    float u = (dot11 * dot02 - dot01 * dot12) * inverDeno;
    if (u < 0 || u > 1) // if u out of range, return directly
    {
        return false;
    }

    float v = (dot00 * dot12 - dot01 * dot02) * inverDeno;
    if (v < 0 || v > 1) // if v out of range, return directly
    {
        return false;
    }

    return u + v <= 1;
```

Algorithms

The process of ray-tracing can be divided into 3 parts listing below:

1. Sampling with Quad-Tree:

   Here I consider variance between the max radiance and min radiance within current region as the criteria to subdivide the tree. And in a leaf node, if the max radiance is big, then the number of samples in this region will be more.

2. Mapping the HDR image:

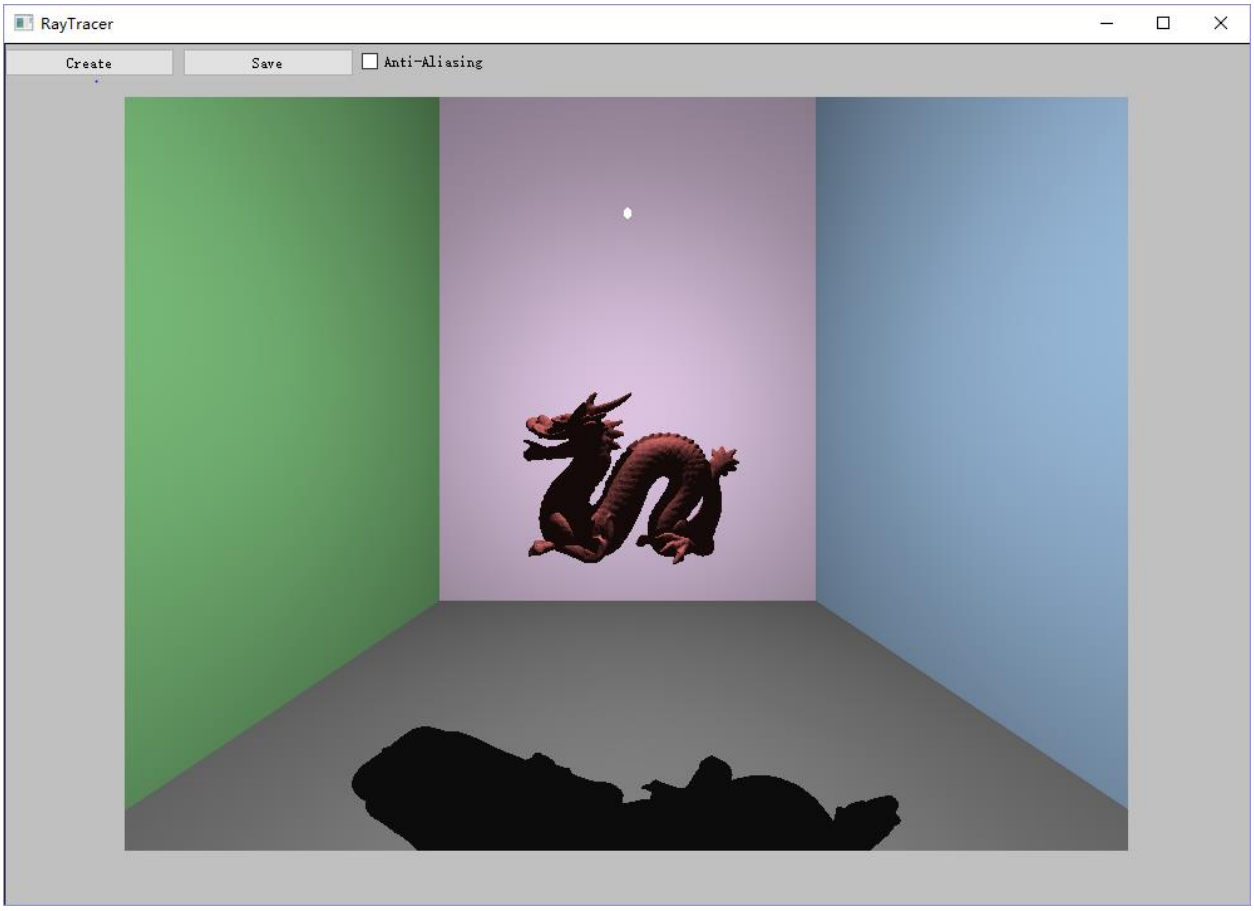
After observing the map image, I split the image into 6 parts, and then use bilinear interpolation to get the environment light color.

3. Rendering mesh objects:

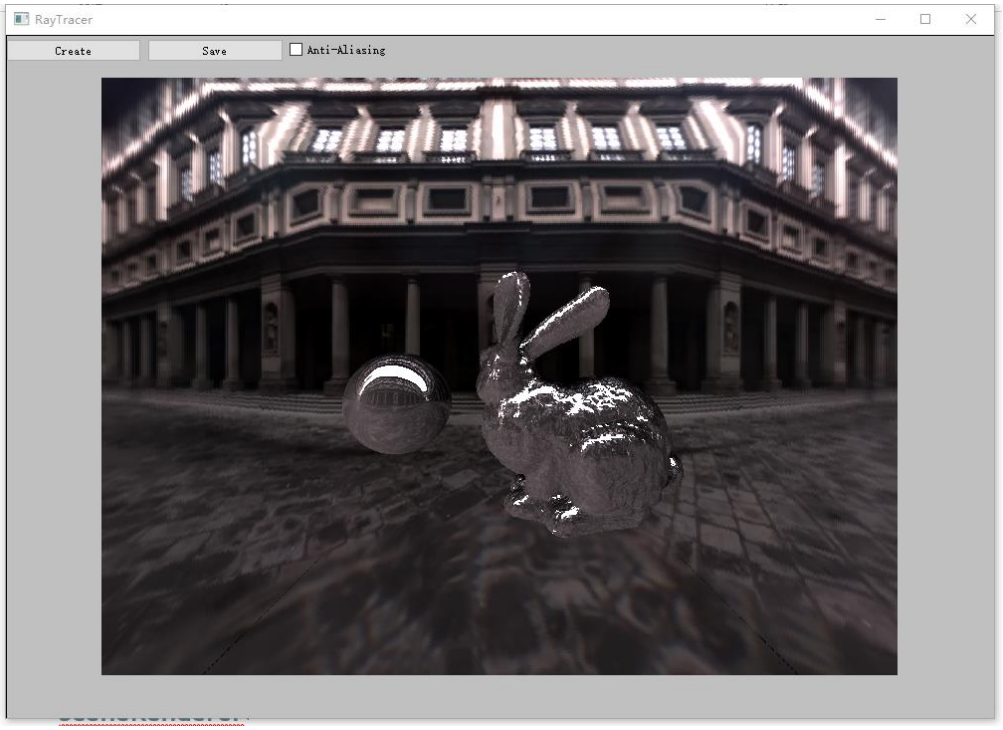
   KD-Tree is used here to make it faster to hit the triangles. When the number of faces in a node is bigger than the threshold, then it is marked as a leaf node. And the space (Bounding-Box) is divided into 2 nodes, aligning in order X-Y-Z.

Example Images

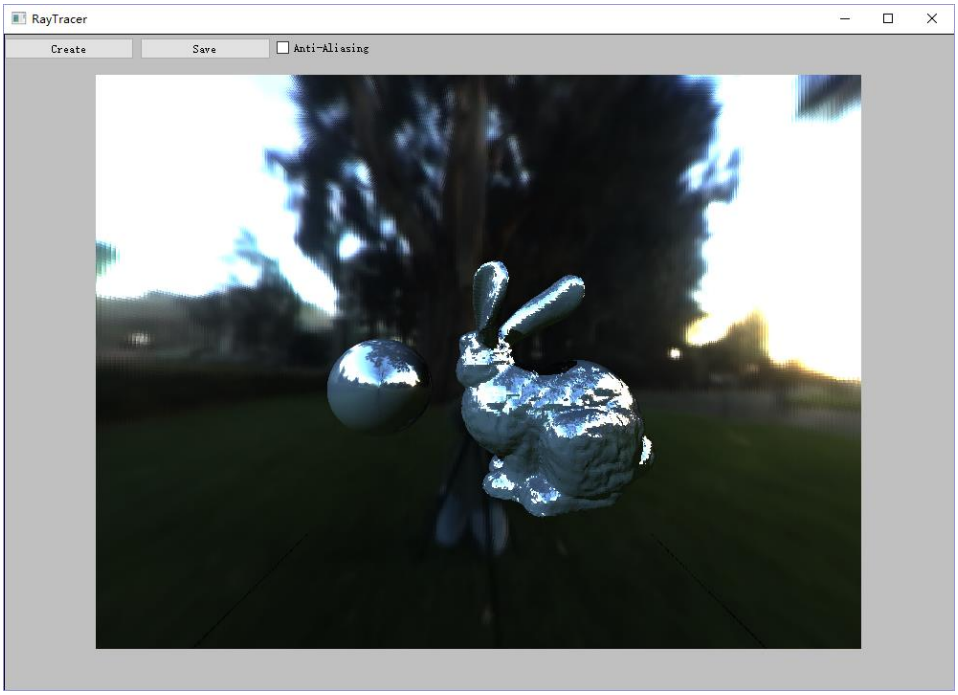
Only Mesh Rendering (point light)



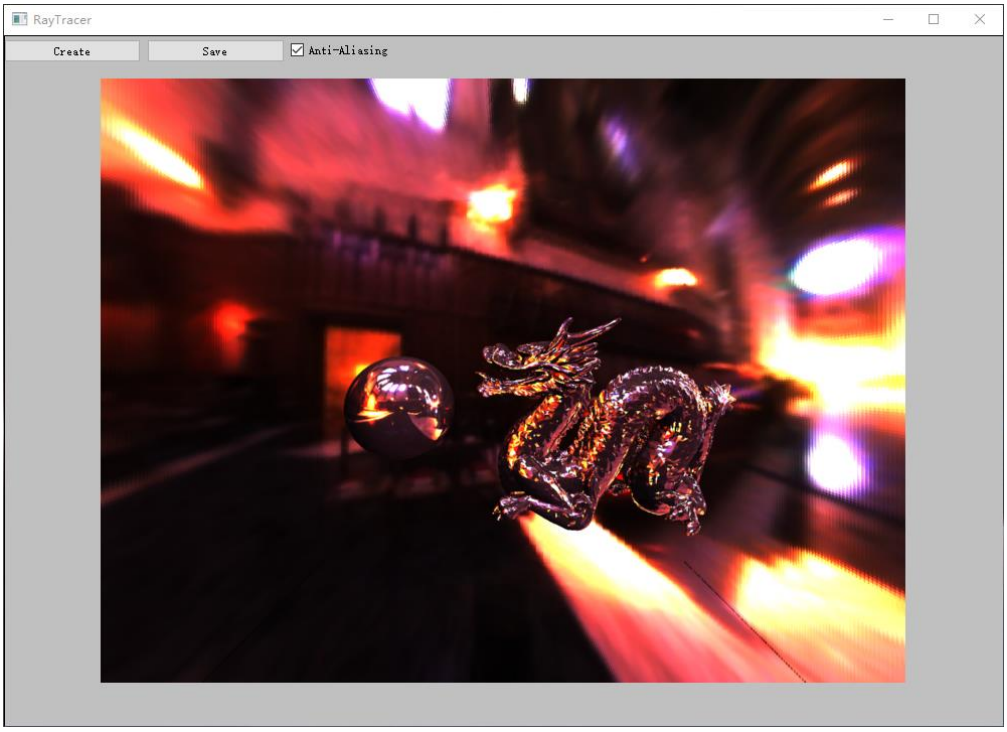
With Environment Lighting (a simple scene)



With Environment Lighting



With Anti-aliasing (dragon with pearl)


RayTracer
Create
Save
Anti-Aliasing