Camera & Scene

Steve Rotenberg

CSE168: Rendering Algorithms

UCSD, Spring 2014

Rendering

- Rendering uses a virtual *camera* to generate a 2D image of a 3D *scene*
- The camera and the scene are therefore key components in a rendering system

Cameras

Camera Data

- The camera contains all of the information about how we are going to generate a 2D image from our 3D scene
- This includes:
  - Matrix: position & orientation of the camera
  - Field of view (FOV): determines the ‘zoom’ of the lens (i.e., ranges from wide angle to telephoto)
  - Resolution: x & y resolution of the image we want to render

Additional Camera Data

- Later in the quarter, we will extend the concept of a camera to include:
  - Focus range (depth of field)
  - Sub-pixel sampling (antialiasing)
  - Exposure settings (high dynamic range imaging)
  - Lens diffusion and imperfections
  - Shutter speed (motion blur)

Camera Matrix

- The camera matrix is a 3D matrix that positions the camera in the world
- Like any 3D matrix, it contains the **a**, **b**, **c**, and **d** vectors
- The **d** vector is the position of the camera
- The **c** vector points backwards (-**c** is the direction of viewing)
- The **a** vector points to the right of the camera
- The **b** vector points to the up direction, relative to the camera
- The camera matrix is almost always orthonormal, so **a**, **b**, and **c** are unit length and perpendicular to each other

‘Look-At’ Function

- It is often convenient to define the camera matrix using a ‘look-at’ function
- This takes a camera position, a target object position, and a world ‘up’ vector (typically the y-axis), and then builds a matrix:

$$\mathbf{d} = \mathbf{pos}$$
$$\mathbf{c} = \mathbf{d} - \mathbf{target}$$
$$\mathbf{c}.Normalize()$$
$$\mathbf{a} = \mathbf{up} \times \mathbf{c}$$
$$\mathbf{a}.Normalize()$$
$$\mathbf{b} = \mathbf{c} \times \mathbf{a}$$

- Note: **b** will be length 1.0 automatically, being the cross product of two orthogonal unit vectors

Field of View

- If we are rendering a rectangular image, then the viewable camera volume is shaped like a pyramid, with the tip at the camera position
- We therefore have two relevant *field of view (FOV)* angles- one for the horizontal FOV and one for the vertical FOV
- The image itself is a rectangle, and therefore, its shape can be defined by as *aspect ratio*, which is the ratio of the image width to the height:

$$aspect = \frac{width}{height}$$

- However, it is important to notice that this is not the same as the ratio of the horizontal to vertical FOVs:

$$aspect \neq \frac{hfov}{vfov}$$

- Because the final image aspect ratio is typically determined by the viewing format (HDTV, film, etc.), it is useful to be able to set up the virtual camera lens by specifying the aspect and one of the FOVs

- It is traditional to define a camera lens by its vertical FOV and image aspect ratio

Field of View

- By examining some right triangles, the relationship between vertical and horizontal field of view can be found:

$$hfov = 2 \cdot \tan^{-1}\left(aspect \cdot \tan\frac{vfov}{2}\right)$$

Pixel Aspect

- The image aspect ratio is the ratio of the width to the height of the actual image
- However, this says nothing about the aspect ratio of the actual pixels of the image
- On most modern display hardware (monitors, tablets, phones…) the pixels themselves are square (aspect 1.0)
- If the pixels are square then the image aspect ratio is the same as the ratio of the horizontal to vertical resolution XRes/YRes
- However, it isn't too uncommon to come across display systems with non-square pixels
- In that case, the actual pixel aspect ratio is:

$$\text{Pixel Aspect} = (\text{ImageWidth} * \text{YRes}) / (\text{ImageHeight} * \text{XRes})$$

Camera Class

```
class Camera {
public:
        Camera();

        void SetFOV(float f);
        void SetAspect(float a);
        void SetResolution(int x,int y);
        void LookAt(Vector3 &pos,Vector3 &target,Vector3 &up);

        void Render(Scene &s);
        void SaveBitmap(char *filename);
private:
        int XRes,YRes;
        Matrix34 WorldMatrix;
        float VerticalFOV;
        float Aspect;
        Bitmap BMP;
};
```

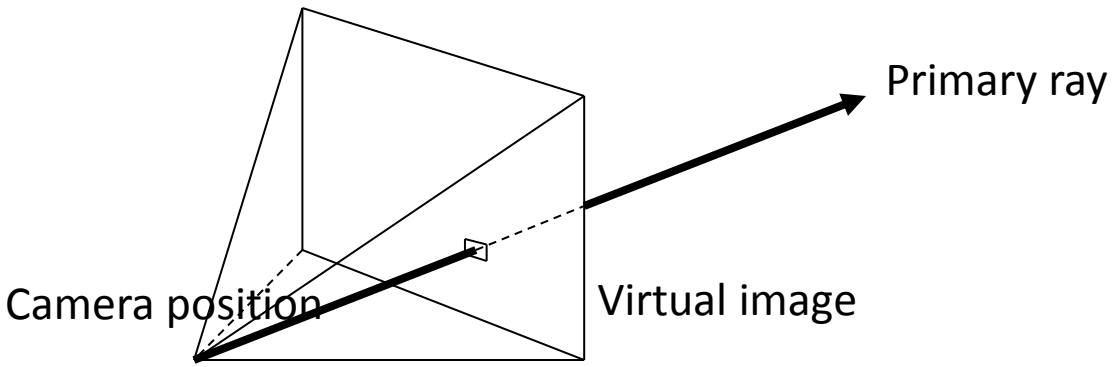
Image Rendering

- When we ray-trace an image, we start by generating rays at the camera and shooting them through each pixel into the scene
- For example, we have a loop something like this:

```
Camera::Render() {
        int x,y;
        for(y=0; y<YRes; y++) {
                for(x=0; x<XRes; x++) {
                        RenderPixel(x,y);
                }
        }
}
```

Camera Rays

- We start by ‘shooting’ rays from the camera out into the scene
- We can render the pixels in any order we choose (even in random order!), but we will keep it simple and go from top to bottom, and left to right
- We loop over all of the pixels and generate an initial *primary ray* (also called a *camera ray* or *eye ray*)
- The ray origin is simply the camera’s position in world space
- The direction is computed by first finding the 4 corners of a virtual image in world space, then interpolating to the correct spot, and finally computing a normalized direction from the camera position to the virtual pixel



Scenes

Scenes

- The scene contains *lights* and *objects*
- Objects include *geometry*, *spatial data structures*, and *scene graph components*
- Geometry includes *triangles* (and/or other primitives) and *materials*

Scene Class

```
class Scene {
public:
    Scene();

    void AddObject(const Object &obj)       {Objects.push_back(&obj);}
    void AddLight(const Light &lgt)         {Lights.push_back(&lgt);}
    void SetSkyColor(const Color sky)       {SkyColor=sky;}

private:
    std::vector<const Object*> Objects;
    std::vector<const Light*> Lights;
    Color SkyColor;
};
```

Lights

Lights

- Lots of different things can emit light in the real world
- For rendering, we often start with simplified virtual lights such as point light sources or infinite directional lights
- For photoreal rendering, we can add area light sources, or allow light to be emitted off of triangles and arbitrary geometry
- Because we will want to support several types of light sources, it is a nice opportunity to use derived classes and virtual functions for lights
- We will create a base Light class and derive a PointLight and a DirectLight for now. Later, we will add more types

Light Base Class

```
class Light {
public:
    Light() {
            Intensity=1.0;
            BaseColor=Color::White;
    }
    void SetBaseColor(const Color &col) {BaseColor=col;}
    void SetIntensity(float i)                  {Intensity=i;}
    virtual float  Illuminate(const Vector3 &pos, Color &col, Vector3 &toLight, Vector3
&ltPos)=0;

protected:
    float Intensity;
    Color BaseColor;              // Actual color is Intensity*BaseColor
};
```

Light Class Details

- It is convenient to assign a color and intensity as properties of the base Light
- However, this doesn't really restrict us to having light sources with a single color or intensity, since we can derive other types later
- The BaseColor values are kept in the 0…1 range, but the Intensity has no upper limit (it does have a lower limit of 0 though). The actual 'color' of the light is the product of the two (Intensity*BaseColor), but it is nice to keep them separate to allow for easier tuning
- The 'Illuminate()' function is the main function for a light source. It takes an input target position 'pos', and determines the intensity and color of light arriving at that position. The final color is returned in the 'col' field and limited to the 0…1 range, while the intensity is the return value of the function itself
- The Illuminate() function also sets a Vector3 &ltPos, which is the position of the light source, and a normalized Vector3 &toLight which points from pos to ltPos. Later, when we add support for shadows and area lights, the necessity for these will become apparent.

Point Lights

- We can start by creating a simple point light source
- We will derive PointLight off of the base Light
- The only new information it needs is a 3D position
- The point light will behave according to physics and will have an inverse square falloff of intensity with respect to distance

PointLight Class

```
class PointLight:public Light {
public:
    PointLight();
    float Illuminate(const Vector3 &pos, Color &col, Vector3 &toLight, Vector3 &ltPos) {
            toLight = Position – pos;
            float bright = Intensity / toLight.Magnitude2();     // Inverse square falloff
            toLight.Normalize();
            col = BaseColor;
            ltPos = Position;
            return bright;
    }

private:
    Vector3 Position;
};
```

Directional Lights

- We will also add a directional light to simulate very distant light sources such as the sun
- As the distance is assumed to be very large, we can ignore the effects of inverse square falloff, and we can have the intensity (and direction) be the same everywhere

DirectLight Class

```
class DirectLight:public Light {
public:
    DirectLight();
    float Illuminate(const Vector3 &pos, Color &col, Vector3 &toLight, Vector3 &ltPos) {
            toLight = -Direction;
            col = BaseColor;
            ltPos = pos – (1000000.0 * Direction);          // Create virtual position
            return Intensity;
    }
    void SetDirection(Vector3 &dir)       {Direction=dir; Direction.Normalize();}

private:
    Vector3 Direction;
};
```

Other Light Types

- Historically in computer graphics, people have defined several common light types such as point, directional, spot, area, projections, and others
- We could derive new light types if we want, but for now, we will stick with point & directional
- Later in the course, we will discuss area lights and some others

Objects

Objects

- The concept of an ‘Object’ is intended to include all of the things we might need to intersect rays with
- This includes geometry such as triangles and/or other primitives like spheres and planes
- It could potentially include more complex surface types such as curved surfaces, NURBS, subdivision surfaces, etc.
- Objects can also include spatial data structures used for optimization
- Objects can also include other scene management components, such as *instances*, which allow copies of other objects to be positioned and oriented with a matrix

Object Base Class

```
class Object {
public:
    virtual ~Object();
    virtual bool Intersect(const Ray &ray, Intersection &hit)=0;
};
```

NOTE: The virtual destructor ~Object() is necessary in order to allow for derived objects to have destructors. For example, if we derive a MeshObject that has to allocate arrays for vertices & triangles, it will need a destructor to delete that memory. Creating a virtual destructor in the base class allows higher level classes to overload the destruction routine.

Meshes

- Rather than derive the Triangle off of Object directly, we will use a MeshObject that contains a mesh of Triangles, as well as the supporting Vertexes and Materials
- This is also more efficient as it allows the Triangle::Intersect() routines to be called within a tight loop, and not have to use virtual function calls
- A mesh class will also allow us to add convenient routines such as MakeBox(), and LoadFile()

MeshObject Class

```
class MeshObject:public Object {
public:
        MeshObject();
        bool Intersect(const Ray &ray, Intersection &hit);

        bool Load(const char *filename);
        void MakeBox(float x,float y,float z);

private:
        int NumVertexes, NumTriangles, NumMaterials;
        Vertex *Vertexes;
        Triangle *Triangles;
        Material *Materials;
};
```

Materials

- Any actual rendered geometry (such as triangles) needs to have some sort of material information associated with it
- Materials may have a wide variety of optical appearances from shiny, dull, transparent, opaque, translucent, multicolored, etc.
- In order to allow for a wide variety of material types, we will want to define a base class material and allow specific types to be derived from it
- We will talk more about these in a couple lectures

Spatial Data Structures

- Earlier, we stated that Objects include geometry, spatial data structures, and scene graph data
- Of those, only geometry is actually visible
- Spatial data structures are invisible geometric data structures build around the geometry for the main purpose of speeding up ray intersection tests
- Some graphics people call them *acceleration structures*, but *spatial data structure* is a more general purpose term within computer science
- We will spend a couple lectures on spatial data structures soon

Scene Graph Components

- Objects can also include scene graph components
- These are other components that describe how a scene is organized
- Ray tracers might use a couple different types of scene graph components, but the most important type is an *instance*

Instancing

- An *instance* is essentially a copy of an object
- For example, if we wanted to render a room with 100 chairs, we would store the chair model one time and build a scene containing 100 instances of that model
- An instance would typically contain a pointer to the object being instanced and a matrix to position the object in space

InstanceObject Class

```
Class InstanceObject:public Object {
public:
        InstanceObject();
        bool Intersect(const Ray &ray, Intersection &hit);
        void SetChild(Object &obj);
        void SetMatrix(Matrix34 &mtx);

private:
        Matrix34 Matrix;
        Matrix34 Inverse;       // Pre-computed inverse of Matrix
        Object *Child;
};
```

Instancing

- Instancing works out particularly well with ray tracers
- Instead of using the instance matrix to transform the object into world space, we use the inverse of the matrix to *transform the ray into the object's space*
- If we find an intersection (in object space), then we use the instance matrix to transform that intersection back into world space
- This way, we do a couple quick computations per ray rather than having to transform several copies of objects into world space
- To avoid having to generate a matrix inverse every time, we simply precompute that before we start rendering and store it with the instance

```
bool Instance::Intersect(const Ray &ray, Intersection &hit) {
        Ray ray2;
        MtxInverse.Transform(ray.Origin, ray2.Origin);
        MtxInverse.Transform3x3(ray.Direction, ray2.Direction);
        if(ChildObject->Intersect(ray2, hit)==false) return false;
        Matrix.Transform(hit.Position, hit.Position);
        Matrix.Transform3x3(hit.Normal, hit.Normal);
        hit.HitDistance=ray.Origin.Distance(hit.Position);          // Correct for any scaling
        return true;
}
```

Project 1

Project 1

- For project 1, create a basic ray tracer and render some boxes lit with some lights
- It should be able to run the sample code on the next slide, as well as some more complex samples listed on the web page (future projects will also run similar samples)
- It should be able to generate an exact match of the sample image in a few seconds
- I suggest you start with the base code provided, but you don't have to if you don't want to
- Compile with optimization on (in VisualStudio, this means setting it to *release* build)
- Due on Friday 4/11/14 at 11:00 am

Sample Program

```
main() {
        // Set up scene
        Scene scn;
        scn.SetSkyColor(Color(0.8, 0.9, 1.0));

        // Create box
        MeshObject box;
        box.MakeBox(0.5,0.2,0.3);
        scn.AddObject(box);

        // Create light
        PointLight redlgt;
        redlgt.SetBaseColor(Color(1.0, 0.2, 0.2));
        redlgt.SetIntensity(10.0);
        redlgt.SetPosition(Vector3(1.0, 2.0, 1.5));
        scn.AddLight(redlgt);

        // Create camera
        Camera cam;
        cam.LookAt(Vector3(0,2,5),Vector3(0,0,0),Vector3(0,1,0));
        cam.SetFOV(50.0);          // NOTE: this is in degrees for UI purposes. Internally, it should be stored as radians
        cam.SetAspect(1.33);
        cam.SetResolution(800,600);

        // Render image
        cam.Render(scn);
        cam.SaveBitmap("test.bmp");
}
```

Sample Image