Final Project: Smoke Rendering

YIHENG WU

The implementation is based on HW3 and HW4 of YiHengWu. It follows the PBRT naming style for variables and functions so that those who are familiar with PBRT will find it comfortable to read the code.

The PBRT-style framework splits the tracer into independent parts: shape, primitive, aggregate, sampler, distribution, scattering, material, texture, light, medium. These information accompanied with the camera are recorded in the scene settings. The scene is finally rendered by the integrator and an image is outputted.

We assume that the readers are familiar with PBRT so we will only focus on medium, integrator and aggregate.

3.1 Volume

See 'medium/volume.h' and 'medium/volume.cpp' for more details.

The volume framework is based on OpenVDB, thus a volume is simply a OpenVDB grid with some interfaces to access the internal value. A smoke can be mostly described by its density with a VDB float grid. A problem is that the grid is described in an unsigned integer coordinate (grid coordinate) so that rays should be transformed from the world coordinate to the grid coordinate before tracking. That is why 'LocalToGrid()' is a necessary virtual function of 'class Volume'.

```
class Volume:
public:
    virtual Transform LocalToGrid() const = 0;
```

We will focus on the child class 'class VolumeFloatGrid' and its child class 'class VolumeFloatKDTree'.

The reason why we also need a KDTree is that simply using the uniform grid for decomposition tracking will always result in bad performance because the minimum value of grid, which is chosen as μ_t^c, is always very small. So we use a KDTree to split the grid into blocks where the minimum value inside each block is expected to be higher. Our implementation of the KDTree is based on the article introduced above.

```
class VolumeFloatGrid : public Volume;
class VolumeFloatKDTree : public VolumeFloatGrid;
```

3.1.1 Grid. The implementation of 'class VolumeFloatGrid' is quite simple. It simply provides interfaces between our framework and OpenVDB data structure. Information about the grid including the max and min value and the boundary are recorded as fields to enable fast accessing.

3.1.2 KDTree. The KDTree is the child class of the grid, with node-level fields including the max and min value, split axis and split position. Information about a node is compressed and recorded into 64 bits. The format of the tree is shown here.

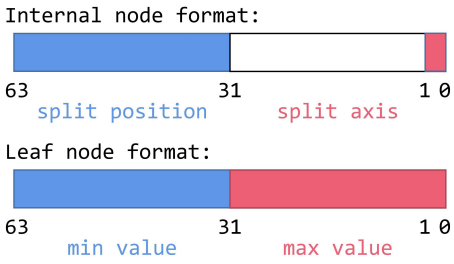

Fig. 1. KDTree node format.

The lowest two bits (split axis) is used to indicate whether a node is a internal or leaf node. For internal nodes, split axis $\in 0, 1, 2$ indicating x, y, z axis. For leaf nodes, the lowest two bits are fixed to 3.

You may have noticed that split axis shares two bits with max value so that for leaf nodes, the lowest two bits of the max value is always '0b11', which results in some acceptable errors but achieves great speedup.

You may also have noticed that parent nodes do not record the index of their child nodes. This makes our kdTree different from PBRT. We compressed the kdTree into a heap where children of node at array index i are located at $i << 1$ and $(i << 1) + 1$. We chose this design to compress the node into 64-bits, making it more cache-friendly for 64-bit-cacheline systems.

Following this format, 'class VolumeFloatKDTree' provides 4 inline functions to get split axis, split position of internal nodes and max, min value of leaf nodes. Here shows the interfaces, where 'idx' is the index of the kdTree array with 64-bit stride.

```
class VolumeFloatKDTree : public VolumeFloatGrid:
public:
    inline int KDTreeSplitAxis(int idx) const;
    inline int KDTreeSplitPos(int idx) const;
    inline int KDTreeMaxValue(int idx) const;
    inline int KDTreeMinValue(int idx) const;
private:
    void* kdTree;
```

Building of the KDTree is based on the article introduced above. Since the implementation is simple, it will not be discussed here.

© 2022

3.2 Medium

See 'medium/medium.h', 'medium/medium.cpp' for more details.

This section will focus on the interface design of the medium system. Readers can skip this part if they are familiar with PBRT.

As we have fully explained, the volume path tracing procedure primarily needs 3 building blocks: sampling scattering distance, evaluating transmittance and sampling scattering direction.

3.2.1 Sample Distance. Weighed delta tracking and weighed decomposition tracking are used. Since these two solutions are based on reject sampling, a random number generator (or a sampler in our framework) is needed. The current weight is also given as a parameter, which will be updated during samples. Sampling results including the position is recorded in a 'MediumInteraction'. Here is the interface.

```
class Medium:
public:
    virtual Spectrum Sample(
        const Ray& rayWorld,   // World-coord ray
        Sampler& sampler,      // Random generator
        Spectrum* weigh,       // Weighed tracking
        MediumInteraction* mi // Sampling results
    ) const = 0;
```

The return value is a spectrum used to update β in integrators. It will be explained in detail later.

3.2.2 Evaluate Transmittance. Our implementation of evaluating transmittance is based on radio tracking and residual radio tracking, thus a sampler is also needed. We follow the rule that the transmittance to evaluate is between $t = 0$ and $t =$ rayWorld.tMax. Here is the interface.

```
class Medium:
public:
    virtual Spectrum Tr(
        const Ray& rayWorld, // World-coord ray
        Sampler& sampler,    // Random generator
    ) const = 0;
```

3.2.3 Sample Direction. We simply sample the HenyeyGreenstein phase function so that the interface is similar to PBRT.

```
class PhaseFunction:
public:
    virtual Float P(
        const Vector3f& wo, // Omega out
        const Vector3f& wi  // Omega in
    ) const = 0;
    virtual Float Sample_P(
        const Vector3f& wo,
        Vector3f* wi,
        const Point2f& samples
    ) const = 0;
class HenyeyGreenstein : public PhaseFunction;
```

3.3 Surface

See 'medium/medium.h', 'medium/medium.cpp' for more details.

To combine surface rendering and volume rendering, it is necessary to know whether the ray is travelling through air or inside some medium such as smoke. The simplest approach is assigning every primitive a surface, which records the medium outside the primitive surface and inside the primitive surface.

```
class Surface:
public:
    const Medium* mediumIn; // Medium inside
    const Medium* mediumOu; // Medium outside
```

Then every geometric primitive should be constructed with a surface instance.

```
class GeometricPrimitive : public Primitive:
public:
    const Surface surface;
```

When 'GeometricPrimitive::Intersect()' is called, surface information should be recorded in a 'SurfaceInteraction' instance. So 'class SurfaceInteraction' should also contain a 'Surface' instance as its field.

```
class SurfaceInteraction : public Interaction:
public:
    Surface surface; // Surface of hit primitive
```

It should be noticed that it is difficult for the framework to handle overlaps of mediums. Take this picture as an example.

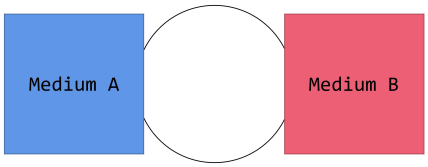

Fig. 2. Weakness of the surface interface.

Since 'class Surface' contains only one pointer for the internal and external medium, it is impossible to handle this two-medium case. So the middle sphere has to be split into two hemispheres.

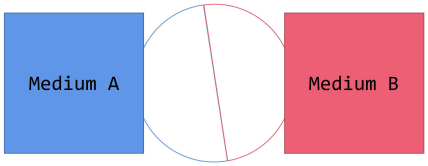

Fig. 3. Solution to the weakness of the surface interface.

Now that the data structures of the framework is clear, we will continue to the volume path tracing integrator.

3.4 Integrator

See 'volumePathSamplerIntegrator.h', 'volumePathSamplerIntegrator.cpp' for more details.

While 'class SamplerIntegrator' implements 'void Render()' to handle the rendering of an image, every child classes derived form 'class SamplerIntegrator' should implement 'Spectrum Integrate()' to evaluate the radiance of a given ray casted to the scene. The method 'Spectrum Integrate()' shares the similar meaning with 'Spectrum Li()' in PBRT.

To extend surface path tracing to volume path tracing, we first introduce the assumptions (restrictions) of our implementation. Then, we will mathematically prove the correctness of our integrator. Finally, we will explain the details of the implementation.

3.4.1 Assumptions. It is difficult to implement emissive mediums because of the complicity to randomly sample a volume light. Even the mathematical proof will be much more complicated, so we assume that all mediums are non-emissive.

3.4.2 Mathematics. We start from volume rendering equation (VRE) with surface contributions.

$$\begin{aligned} L(\mathbf{x},\omega) = &\int_0^z T(\mathbf{x},\mathbf{y})[\mu_a(\mathbf{y})L_e(\mathbf{y},\omega) + \mu_s(\mathbf{y})L_s(\mathbf{y},\omega)]dy \\ &+ T(\mathbf{x},\mathbf{z})L(\mathbf{z},\omega) \end{aligned} \tag{1}$$

For non-emissive mediums, $L_e(\mathbf{y},\omega) = 0$, so

$$L(\mathbf{x},\omega) = \int_0^z T(\mathbf{x},\mathbf{y})\mu_s(\mathbf{y})L_s(\mathbf{y},\omega)dy + T(\mathbf{x},\mathbf{z})L(\mathbf{z},\omega) \tag{2}$$

where

$$L_s(\mathbf{y},\omega) = \int_{S^2} f_p(\omega,\bar{\omega})L(\mathbf{y},\bar{\omega})d\bar{\omega} \tag{3}$$

Suppose camera is located at position p_0, p_1 is the first sampled position. Similarly define $\mathrm{p}_2, \mathrm{p}_3, \cdots$.

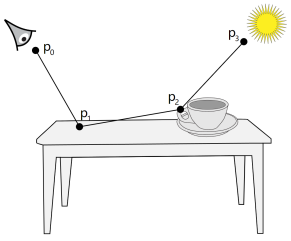

Fig. 4. Path tracing.

For path tracing, we firstly consider only the two simplest cases: both $\mathrm{p}_i, \mathrm{p}_{i+1}$ are in the air (no medium) and both $\mathrm{p}_i, \mathrm{p}_{i+1}$ are in some medium. Similar to the proof of surface path tracking in PBRT, we will derive the iteration of β and 'UniformSampleOneLight()' of these two cases. These two situations will be extended to more complicated cases later.

For the first case where both $\mathrm{p}_i, \mathrm{p}_{i+1}$ are in the air, taking $\mathrm{p}_0, \mathrm{p}_1$ as an example.

$$\begin{aligned} L(\mathrm{p}_1 \to \mathrm{p}_0) &= L_e(\mathrm{p}_1 \to \mathrm{p}_0) + L_r(\mathrm{p}_1 \to \mathrm{p}_0) \\ &= L_e(\mathrm{p}_1 \to \mathrm{p}_0) \\ &\quad + \int_{H^2(n)} L(\bar{\mathrm{p}} \to \mathrm{p}_1) f_r(\bar{\mathrm{p}} \to \mathrm{p}_1 \to \mathrm{p}_0)|\cos\bar{\theta}|d\bar{\omega} \end{aligned} \tag{4}$$

where L_e is the direct illumination from position p_1 to position p_0, L_r is the indirect illumination from position p_1 to position p_0. It should be noticed that $\bar{\mathrm{p}}$ is not ensured to be in air or in some medium so that domain of $\bar{\mathrm{p}}$ should be split to those in air (on another surface), donated as $\bar{\mathrm{p}}^s$ and those in some medium, donated as $\bar{\mathrm{p}}^m$.

$$\begin{aligned} L(\mathrm{p}_1 \to \mathrm{p}_0) = &L_e(\mathrm{p}_1 \to \mathrm{p}_0) \\ &+ \int_{H^2(n)} L(\bar{\mathrm{p}}^s \to \mathrm{p}_1) f_r(\bar{\mathrm{p}}^s \to \mathrm{p}_1 \to \mathrm{p}_0)|\cos\bar{\theta}|d\bar{\omega} \\ &+ \int_{H^2(n)} L(\bar{\mathrm{p}}^m \to \mathrm{p}_1) f_r(\bar{\mathrm{p}}^m \to \mathrm{p}_1 \to \mathrm{p}_0)|\cos\bar{\theta}|d\bar{\omega} \end{aligned} \tag{5}$$

where $L(\bar{\mathrm{p}}^s \to \mathrm{p}_1)$ can be evaluated using the surface contribution part of VRE and $L(\bar{\mathrm{p}}^m \to \mathrm{p}_1)$ can be evaluated using the volume contribution part of VRE.

$$\begin{aligned} L(\bar{\mathrm{p}}^s \to \mathrm{p}_1) &= T(\bar{\mathrm{p}}^s, \mathrm{p}_1)L_e(\bar{\mathrm{p}}^s \to \mathrm{p}_1) + T(\bar{\mathrm{p}}^s, \mathrm{p}_1)L_r(\bar{\mathrm{p}}^s \to \mathrm{p}_1) \\ L(\bar{\mathrm{p}}^m \to \mathrm{p}_1) &= \int_{\mathrm{p}_1}^{\bar{\mathrm{p}}^m} T(\mathrm{p}_1, \mathrm{p})\mu_s(\mathrm{p})L_s(\mathrm{p}, \omega_{\bar{\mathrm{p}}^m \to \mathrm{p}_1})ds \end{aligned} \tag{6}$$

Substitute $L(\bar{\mathrm{p}}^s \to \mathrm{p}_1)$ into the path tracing equation.

$$\begin{aligned} &L(\mathrm{p}_1 \to \mathrm{p}_0) \\ &= L_e(\mathrm{p}_1 \to \mathrm{p}_0) \\ &+ \int_{H^2(n)} T(\bar{\mathrm{p}}^s, \mathrm{p}_1)L_e(\bar{\mathrm{p}}^s \to \mathrm{p}_1) f_r(\bar{\mathrm{p}}^s \to \mathrm{p}_1 \to \mathrm{p}_0)|\cos\bar{\theta}|d\bar{\omega} \\ &+ \int_{H^2(n)} T(\bar{\mathrm{p}}^s, \mathrm{p}_1)L_r(\bar{\mathrm{p}}^s \to \mathrm{p}_1) f_r(\bar{\mathrm{p}}^s \to \mathrm{p}_1 \to \mathrm{p}_0)|\cos\bar{\theta}|d\bar{\omega} \\ &+ \int_{H^2(n)} L(\bar{\mathrm{p}}^m \to \mathrm{p}_1) f_r(\bar{\mathrm{p}}^m \to \mathrm{p}_1 \to \mathrm{p}_0)|\cos\bar{\theta}|d\bar{\omega} \end{aligned} \tag{7}$$

Apply Monte Carlo to the above equation.

$$\begin{aligned} &L(\mathrm{p}_1 \to \mathrm{p}_0) \\ &= L_e(\mathrm{p}_1 \to \mathrm{p}_0) \\ &+ \frac{T(\bar{\mathrm{p}}^s_{light}, \mathrm{p}_1)L_e(\bar{\mathrm{p}}^s_{light} \to \mathrm{p}_1) f_r(\bar{\mathrm{p}}^s_{light} \to \mathrm{p}_1 \to \mathrm{p}_0)|\cos\bar{\theta}|}{p(\omega_{light})} \\ &+ \begin{cases} \dfrac{L_r(\bar{\mathrm{p}}^s_{bsdf} \to \mathrm{p}_1) f_r(\bar{\mathrm{p}}^s_{bsdf} \to \mathrm{p}_1 \to \mathrm{p}_0)|\cos\bar{\theta}|}{p(\omega_{bsdf})} & \text{sample in air} \\ \dfrac{L(\bar{\mathrm{p}}^m_{bsdf} \to \mathrm{p}_1) f_r(\bar{\mathrm{p}}^m_{bsdf} \to \mathrm{p}_1 \to \mathrm{p}_0)|\cos\bar{\theta}|}{p(\omega_{bsdf})} & \text{sample in some medium} \end{cases} \end{aligned} \tag{8}$$

Define function

$$\begin{aligned}&\text{UniformSampleOneLight}(\mathrm{p}_1 \to \mathrm{p}_0)\\ &= \frac{T(\bar{\mathrm{p}}^s_{light}, \mathrm{p}_1) L_e(\bar{\mathrm{p}}^s_{light} \to \mathrm{p}_1) f_r(\bar{\mathrm{p}}^s_{light} \to \mathrm{p}_1 \to \mathrm{p}_0) |\cos\bar{\theta}|}{p(\omega_{light})}\end{aligned} \tag{9}$$

There is still a problem that it is difficult to evaluate $L(\bar{\mathrm{p}}^m_{bsdf} \to \mathrm{p}_1)$ because p_1 is in the air while $\bar{\mathrm{p}}^m_{bsdf}$ is in the medium. This problem can be solved by applying energy conservation while radiance is travelling in the air.

Since $\bar{\mathrm{p}}^m_{bsdf}$ is in the medium and p_1 is in the air, there exists a position where radiance travels exactly from medium to the air. Take this picture as an example.

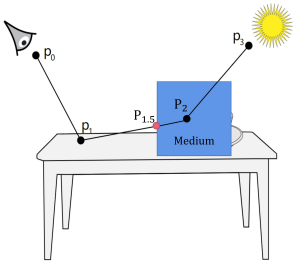

Fig. 5. Path tracing.

Let $\mathrm{p}_{1.5}$ be the boundary between volume and air, apply energy conservation.

$$L(\bar{\mathrm{p}}^m_{bsdf} \to \mathrm{p}_1) = L(\mathrm{p}_{1.5} \to \mathrm{p}_1) \tag{10}$$

Then the problem is converted from evaluating radiance from medium to air, to evaluating radiance from medium to medium because the boundary can be both considered as an in-air position and an in-medium position.

The mathematics about evaluating radiance from medium to medium will be discussed then, which is exactly the second case discussed above where both $\mathrm{p}_i, \mathrm{p}_{i+1}$ are in the air. Again take $\mathrm{p}_0, \mathrm{p}_1$ as an example. For in-medium case, p_1 is got by sampling distance, so we donate $\mathrm{p} = \mathrm{p}_1$ below.

$$\begin{aligned}&L(\mathrm{p} \to \mathrm{p}_0)\\ &= \int_{\mathrm{p}_0}^{\mathrm{p}} T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p}) L_s(\mathrm{p} \to \mathrm{p}_0) ds\\ &= \int_{\mathrm{p}_0}^{\mathrm{p}} T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p}) \int_{S^2} f_p(\bar{\mathrm{p}} \to \mathrm{p} \to \mathrm{p}_0) L(\bar{\mathrm{p}} \to \mathrm{p}) d\bar{\omega} ds\end{aligned} \tag{11}$$

Similar to the in-air case, $\bar{\mathrm{p}}$ is not ensured to be in air or in some medium.

$$\begin{aligned}&L(\mathrm{p} \to \mathrm{p}_0)\\ &= \int_{\mathrm{p}_0}^{\mathrm{p}} T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p}) \int_{S^2} f_p(\bar{\mathrm{p}}^s \to \mathrm{p} \to \mathrm{p}_0) L(\bar{\mathrm{p}}^s \to \mathrm{p}) d\bar{\omega} ds\\ &+ \int_{\mathrm{p}_0}^{\mathrm{p}} T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p}) \int_{S^2} f_p(\bar{\mathrm{p}}^m \to \mathrm{p} \to \mathrm{p}_0) L(\bar{\mathrm{p}}^m \to \mathrm{p}) d\bar{\omega} ds\\ &= \int_{\mathrm{p}_0}^{\mathrm{p}} T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p}) \int_{S^2} f_p(\bar{\mathrm{p}}^s \to \mathrm{p} \to \mathrm{p}_0)\\ &\qquad \left(T(\bar{\mathrm{p}}^s, \mathrm{p}) L_e(\bar{\mathrm{p}}^s \to \mathrm{p}) + T(\bar{\mathrm{p}}^s, \mathrm{p}) L_r(\bar{\mathrm{p}}^s \to \mathrm{p})\right) d\bar{\omega} ds\\ &+ \int_{\mathrm{p}_0}^{\mathrm{p}} T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p}) \int_{S^2} f_p(\bar{\mathrm{p}}^m \to \mathrm{p} \to \mathrm{p}_0) L(\bar{\mathrm{p}}^m \to \mathrm{p}) d\bar{\omega} ds\end{aligned} \tag{12}$$

Apply Monte Carlo for sampling scattering distance.

$$\begin{aligned}&L(\mathrm{p} \to \mathrm{p}_0)\\ &= \frac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} \int_{S^2} f_p(\bar{\mathrm{p}}^s \to \mathrm{p} \to \mathrm{p}_0) T(\bar{\mathrm{p}}^s, \mathrm{p}) L_e(\bar{\mathrm{p}}^s \to \mathrm{p}) d\bar{\omega}\\ &+ \begin{cases} \dfrac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} \displaystyle\int_{S^2} f_p(\bar{\mathrm{p}}^s \to \mathrm{p} \to \mathrm{p}_0) T(\bar{\mathrm{p}}^s, \mathrm{p}) L_r(\bar{\mathrm{p}}^s \to \mathrm{p}) d\bar{\omega} & \text{sample in air} \\ \dfrac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} \displaystyle\int_{S^2} f_p(\bar{\mathrm{p}}^m \to \mathrm{p} \to \mathrm{p}_0) L(\bar{\mathrm{p}}^m \to \mathrm{p}) d\bar{\omega} & \text{sample in some medium} \end{cases}\end{aligned} \tag{13}$$

It should be noticed that for those tracking methods derived from delta tracking, $p(\mathrm{p}) = T(\mathrm{p}_0, \mathrm{p}) \mu_t(\mathrm{p})$. Then

$$\frac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} = \frac{\mu_s(\mathrm{p})}{\mu_t(\mathrm{p})}. \tag{14}$$

Apply Monte Carlo for sampling scattering direction.

$$\begin{aligned}&L(\mathrm{p} \to \mathrm{p}_0)\\ &= \frac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} \frac{f_p(\bar{\mathrm{p}}^s_{light} \to \mathrm{p} \to \mathrm{p}_0) T(\bar{\mathrm{p}}^s_{light}, \mathrm{p}) L_e(\bar{\mathrm{p}}^s_{light} \to \mathrm{p})}{p(\omega_{light})}\\ &+ \begin{cases} \dfrac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} \dfrac{f_p(\bar{\mathrm{p}}^s_{phase} \to \mathrm{p} \to \mathrm{p}_0) T(\bar{\mathrm{p}}^s_{phase}, \mathrm{p}) L_r(\bar{\mathrm{p}}^s_{phase} \to \mathrm{p})}{p(\omega_{phase})} & \text{sample in air} \\ \dfrac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} \dfrac{f_p(\bar{\mathrm{p}}^m_{phase} \to \mathrm{p} \to \mathrm{p}_0) L(\bar{\mathrm{p}}^m_{phase} \to \mathrm{p})}{p(\omega_{phase})} & \text{sample in some medium} \end{cases}\\ &= \frac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} \frac{f_p(\bar{\mathrm{p}}^s_{light} \to \mathrm{p} \to \mathrm{p}_0) T(\bar{\mathrm{p}}^s_{light}, \mathrm{p}) L_e(\bar{\mathrm{p}}^s_{light} \to \mathrm{p})}{p(\omega_{light})}\\ &+ \begin{cases} \dfrac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} T(\bar{\mathrm{p}}^s_{phase}, \mathrm{p}) L_r(\bar{\mathrm{p}}^s_{phase} \to \mathrm{p}) & \text{sample in air} \\ \dfrac{T(\mathrm{p}_0, \mathrm{p}) \mu_s(\mathrm{p})}{p(\mathrm{p})} L(\bar{\mathrm{p}}^m_{phase} \to \mathrm{p}) & \text{sample in some medium} \end{cases}\end{aligned} \tag{15}$$

Define function

$$\begin{aligned}&\text{UniformSampleOneLight}(\mathrm{p} \to \mathrm{p}_0)\\ =\ &\frac{T(\mathrm{p}_0,\mathrm{p})\mu_s(\mathrm{p})}{p(\mathrm{p})}\frac{f_p(\bar{\mathrm{p}}^s_{light}\to\mathrm{p}\to\mathrm{p}_0)T(\bar{\mathrm{p}}^s_{light},\mathrm{p})L_e(\bar{\mathrm{p}}^s_{light}\to\mathrm{p})}{p(\omega_{light})}\end{aligned} \tag{16}$$

There is also a similar problem that it is difficult to evaluate $L_r(\bar{\mathrm{p}}^s_{phase} \to \mathrm{p})$ because p is in some medium while $\bar{\mathrm{p}}^s_{phase}$ is in the air. This problem can also be solved similarly by energy conservation.

For now, the two simplest cases (both $\mathrm{p}_i, \mathrm{p}_{i+1}$ are in the air or in some medium) has been solved. Readers may have noticed that it is quite easy to derive them into more complicated cases by using an empty BSDF for the wrapping surface of the medium. That is, ray simply travels through the wrapping surface without scattering. The following picture shows the idea.

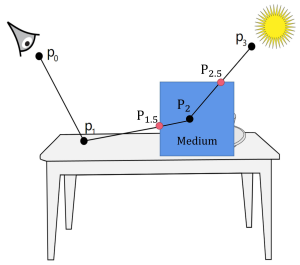

Fig. 6. Ultimate volume path tracing.

Path $\mathrm{p}_3 \to \mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0$ is extended to $\mathrm{p}_3 \to \mathrm{p}_{2.5} \to \mathrm{p}_2 \to \mathrm{p}_{1.5} \to \mathrm{p}_1 \to \mathrm{p}_0$, then nodes of each single path are located both in the air or in some medium. It is valid to combine them together, which derives our integrator for volume rendering.

Now we will derive the iteration of β and 'UniformSampleOneLight()', which is clearly proved above.

$$\beta(i) = \prod_{j=1}^{i}\begin{cases}\dfrac{f_r(\mathrm{p}_{j+1}\to\mathrm{p}_j\to\mathrm{p}_{j-1})|\cos\theta_j|}{p_\omega(\mathrm{p}_{j+1}\to\mathrm{p}_j)} & \text{sample in air}\\ \dfrac{T(\mathrm{p}_j,\mathrm{p}_{j+1})\mu_s(\mathrm{p}_{j+1})}{p(\mathrm{p}_{j+1})} & \text{sample in some medium}\end{cases} \tag{17}$$

$$\begin{aligned}&\text{UniformSampleOneLight}(\mathrm{p}_j \to \mathrm{p}_{j-1})\\ =\ &\begin{cases}\dfrac{T(\bar{\mathrm{p}}^s_{light},\mathrm{p}_j)L_e(\bar{\mathrm{p}}^s_{light}\to\mathrm{p}_j)f_r(\bar{\mathrm{p}}^s_{light}\to\mathrm{p}_j\to\mathrm{p}_{j-1})|\cos\bar{\theta}|}{p(\omega_{light})} & \text{sample in air}\\ \dfrac{T(\mathrm{p}_{j-1},\mathrm{p}_j)\mu_s(\mathrm{p}_j)}{p(\mathrm{p}_j)}\cdot\dfrac{f_p(\bar{\mathrm{p}}^s_{light}\to\mathrm{p}_j\to\mathrm{p}_{j-1})T(\bar{\mathrm{p}}^s_{light},\mathrm{p}_j)L_e(\bar{\mathrm{p}}^s_{light}\to\mathrm{p}_j)}{p(\omega_{light})} & \text{sample in some medium}\end{cases}\end{aligned} \tag{18}$$

Then the radiance of a pixel can be determined. For surface path tracing, the radiance of a pixel can be rewritten to the following iterative equation.

$$\begin{aligned}\text{radiance} = &L_e(\mathrm{p}_1\to\mathrm{p}_0)\\ &+\beta_0\text{UniformSampleOneLight}(\mathrm{p}_1\to\mathrm{p}_0)\\ &+\beta_1\text{UniformSampleOneLight}(\mathrm{p}_2\to\mathrm{p}_1)\\ &+\cdots\end{aligned} \tag{19}$$

For volume path tracing, the iterative equation is too complicated to be described by mathematics so it will not be shown here.

3.4.3 Implementation. The implementation simply translate the above mathematics into code. Readers may find that our implementation is quite similar to PBRT. That is right because we accidentally chose the similar implementation approach with PBRT (but PBRT does not prove the correctness of their implementation). We will further discuss the restriction of PBRT and show the better implementation in the next section while introducing spectral weighed delta tracking and spectral weighed decomposition tracking.

Here is the pseudo code written with PBRT literate programming.

```
Spectrum
VolumePathSamplerIntegrator::Integrate(const Ray& r, const Scene& scene) const:
  Spectrum radiance(Float(0));
  Ray ray(r);
  bool specularBounce = false;
  Spectrum beta(Float(1));
  Spectrum mediumScatteringWeigh(Float(1));
  const Medium* medium = camera->medium;
  for (int bounces = 0; ; ++bounces):
    <<Intersect ray with scene and store intersection in si>>
    <<Sample the participating medium and store intersection in mi, if present>>
    // Handle an interaction with a medium or a surface
    if (mi.IsValid()):
      // Handle scattering at point in medium for volumetric path tracer
      radiance += beta * UniformSampleOneLight(...);
      <<Spawn ray from mi>>
    else:
      // Handle scattering at point on surface for volumetric path tracer
      <<Possibly add emitted light at intersection>>
      <<Terminate path if ray escaped or max depth was reached>>
      // Compute scattering functions and skip over medium boundaries>>
      si.ComputeScatteringFunctions(ray);
      if (!si.bsdf):
        <<Spawn ray from si>>
        --bounces;
        continue;
      // Sample illumination from lights to find attenuated path contribution
      radiance += beta * UniformSampleOneLight(...);
      <<Sample BSDF to get new path direction>>
    <<Possibly terminate the path with Russian roulette>>
  return radiance;
```

For the integrator to run successfully, child classes of 'class Medium' should be implemented then, which will be introduced in the next section.

3.5 Density Medium

See 'densityMedium.h', 'densityMedium.cpp' for more details.

Information of volumes can be described by a floating point grid called density, accompanied by some mega information to translate density into μ_a, μ_s, μ_t.

It should be noticed that even density at a given position is simply a floating point, μ_a, μ_s, μ_t are spectrums. That is, they are wavelength-interdependent. So that spectral tracking (weighed delta tracking and weighed decomposition tracking) should be used. Readers familiar with PBRT may be confused since PBRT implements non-weighed delta tracking. That is because PBRT does not support wavelength-interdependent μ_t. See the source code of PBRT for more details.

In this section, we are proud to extend PBRT by implementing wavelength-interdependent mediums. We will first introduce the fields of the classes. We will then explain how to sample scattering distance and evaluate transmittance for mediums based on grid and KDTree.

3.5.1 Fields of Classes. Density is interpreted as μ_t in our implementation so the first step is to transform density to μ_t. This is done by providing a spectrum called 'toMut', then μ_t = density · toMut. Albedo = $\frac{\mu_s}{\mu_t}$ should also be provided to compute μ_a and μ_s from μ_t. Such an idea derives the interfaces of density mediums.

```
class HeteroMedium : public Medium;
class DensityGridMedium : public HeteroMedium:
protected:
    const VolumeFloatGrid* density;
    const Spectrum toMut;
    const Spectrum albedo;
    const PhaseFunction* phase;
    const Transform worldToMedium;
class DensityKDTreeMedium : public HeteroMedium:
    ... // The same
```

For delta tracking, $\bar{\mu}$ should be determined during the construction. Since density is given, we choose $\bar{\mu}$ = max(density). $\frac{1}{\bar{\mu}}$ is also computed to make the sampling process faster.

```
class DensityGridDeltaMedium :
      public DensityGridMedium:
private:
    Float mubar;
    Float mubar_inv;
```

For decomposition tracking, besides $\bar{\mu}$ and $\frac{1}{\bar{\mu}}$, μ_a^c, μ_s^c, μ_t^c, P_a^c, P_s^c, $P_{residual}$ should also be computed. Since it is not that easy to evaluate transmittance wavelength-interdependently by residual ratio tracking, a homogeneous choice of μ_t^c should also be computed, as well as the corresponding $\frac{1}{\mu^r}$.

```
class DensityGridDecompositionMedium :
      public DensityGridMedium:
private:
    Float mubar;
    Float mubar_inv;
    Spectrum muaC;
    Spectrum musC;
    Spectrum mutC;
    Float paC;
    Float psC;
    Float pResidual;
    Float mutC_homoC;
    Float mubarR_homoC_inv;
```

3.5.2 Sampling Scattering Distance. As has mentioned above, we follow the similar interface for 'Medium::Sample()'. That is, the return value is the updating multiplier for β, which is exactly $\frac{T(\mathrm{p}_{j-1},\mathrm{p}_j)\mu_s(\mathrm{p}_j)}{p(\mathrm{p}_j)}$. For tracking methods based on delta tracking including decomposition tracking, it can be simplified to $\frac{\mu_s(\mathrm{p}_j)}{\mu_t(\mathrm{p}_j)}$. Thus the return value is always the albedo. Our implementation assumes fixed albedo for a medium, different from PBRT.

The code simply translates weighed delta tracking and weighed decomposition tracking of Disney pseudo code into C++, except that 'weigh' is a parameter. When a ray is scattering inside a medium weigh is updated during each scattering process, until the ray finally travels out of the medium, then the weigh is reset to 'Spectrum(Float(1))'. The weigh is kept in the integrator as a variable called 'Spectrum mediumScatteringWeigh'.

Problem occurs when there are solid shapes inside a medium. Delta tracking and transmittance evaluation should never overreach these shapes. So a ray casting always has to be performed before calls to functions 'Sample()' and 'Tr()' in order to record the maximum sampling distance inside 'rayWorld.tMax'. 'tMax' should also be recomputed when ray is transformed from world coordinate to grid coordinate, as readers will see in our source code.

Pseudo code of weighed delta tracking for grid is shown here.

```
Spectrum
DensityGridDeltaMedium::Sample(
        const Ray& rayWorld, Sampler& sampler,
        Spectrum* weigh, MediumInteraction* mi) const:
  <<Transform ray from world coordinate to grid coordinate>>
  <<Ray casting to the bounding box and compute tMin, tMin>>
  // Weighed delta tracking
  Float t = tMin;
  while (true):
    // Step forward
    t -= std::log(1 - sampler.Get1D()) * mubar_inv;
    if (t > tMax):
      break;
    <<Compute sampling variables such as Pa, Ps, Pn>>
    if (kexi < pa):
      <<Set mi->emission>>
      return Spectrum(Float(0)); // Terminate path
    else if (kexi < 1 - pn):
      <<Update weigh>>
      <<Set mi>>
      return albedo; // Return albedo to update beta
    else:
      <<Update weigh>>
  return Spectrum(Float(1)); // Sampling failed
```

Code of weighed decomposition tracking for grid is shown in 'DensityGridDecompositionMedium::Sample()', which is quite similar so it will be shown here.

Tracking for KDTree is also straightforward. Readers can simply replace the intersection testing of traversal of PBRT KDTree with node-level delta tracking or decomposition tracking. But it should be noticed that if a real collision is sampled inside one node, other nodes should no longer be sampled, which is different from a KDTree containing primitives because a primitive may be located inside many nodes but a medium-grid block is located inside exactly one node.

The codes of 'DensityKDTreeDeltaMedium::Sample()' and 'DensityKDTreeDecompositionMedium::Sample()' is so simple that they will also not be included here.

3.5.3 Evaluating transmittance. Since ratio tracking is related to weighed delta tracking and residual ratio tracking is related to weighed decomposition tracking. We use ratio tracking for "delta" medium and residual ratio tracking for "decomposition" medium.

The following code shows the evaluation of transmittance by ratio tracking for grid.

```
Spectrum
DensityGridDeltaMedium::Tr(const Ray& rayWorld, Sampler& sampler) const:
  <<Transform ray from world coordinate to grid coordinate>>
```

```
<<Ray casting to the bounding box and compute tMin, tMin>>
// Ratio tracking
while (true):
  // Step forward
  t -= std::log(1 - sampler.Get1D()) * mubar_inv;
  if (t >= tMax):
    break;
  Float dense = density->Sample(rayMedium(t));
  tr *= Spectrum(Float(1))
      - Spectrum::Max(Spectrum(), dense * mubar_inv * toMut);
return tr;
```

However, for decomposition tracking, μ^c which is independent among wavelengths have to be used because those with interdependence is not supported by residual ratio tracking.

The code for the other three cases are not shown since they share too much similarities.

There is still one interface to implement for the integrator to comfortably evaluate transmittance. A ray may travel through many mediums before hitting a solid surface. For these cases, the accumulated transmittance should be evaluated. This is done by evaluating and multiplying the transmittance of all the mediums along the ray. So 'IntersectTr()' is used instead of 'Intersect()' in volume path tracing, which evaluates the transmittance along the ray and returns whether the ray hits a solid surface. See 'scene/scene.h', 'scene/scene.cpp' for the details of the interface.

```
Scene:IntersectTr(
    const Ray& ray,          // World-coord ray
    Sampler& sampler,        // Random generator
    const Medium* medium,    // Medium with ray.o
    Float* tHit,             // Hit time
    SurfaceInteraction* si, // Interaction info
    Spectrum* tr             // Transmittance
);
```

Now we are done.

3.6 Aggregate

See 'embreeAccel.h', 'embreeAccel.cpp' for more details.

Since volume global illumination is quite slow, we provide an accelerator based on Intel Embree. The scenes are rendered with 'embreeAccel' in default for performance, but we also provide interfaces for our self-written accelerators including KDTree and BVH.

```
#ifdef INTEL
shared_ptr<EmbreeAccel> aggregate =
    make_shared<EmbreeAccel>(primitives);
#else
shared_ptr<BVH> aggregate =
    make_shared<BVH>(primitives);
#endif
```

Readers can enable or disable Embree by setting 'INTEL' in cmake.

```
set(INTEL true)
```

4 TODO

We failed to implement everything because time is limited. Features not yet implemented are introduced in this section.

4.1 Emissive Medium

Has been proved above, our integrator does not support emissive medium. A better integrator should be mathematically proved.

4.2 Faster IntersectTr()

The function 'IntersectTr()' is quite slow because it may calls 'Intersect()' multiple times when there are multiple mediums in the scene. This can be solved by using Embree interfaces about masking.

4.3 MIS

We haven't updated the MIS for surface rendering to volume rendering because some interfaces of the framework have been updated to support volume rendering, so that MIS has to be written.

4.4 Better Interface for Surface

Has also been mentioned above, it is difficult for our interface to handle overlaps of solid shapes and mediums. A special pointer texture may be a solution, but storing and looking up such a texture may be expensive.

4.5 Packet Tracking

Since our implementation is originally not based on Embree, it does not support packet tracking very well. To implementation this feature, the whole framework should be rewritten.

4.6 Handling Floating Point Errors

PBRT restrictedly handle floating point errors especially nans. It is extremely important when rays are frequently bounced from surfaces because infinite loops will occur if rays are not correctly bounced.

5 RESULTS