Realistic Cloud Rendering Using Pixel Synchronization

Egor Yusov
Introduction

Clouds are integral part of outdoor scenes

Rendering good-looking and fast clouds is challenging

Different approaches to the problem exist

- Billboards
- Ray-marching
- Direct volume rendering (slicing)
Existing methods - Particles

- Represents the clouds as collection of camera-facing polygons (quads)
  - Can combine simple shapes (radial fall-off textures) as well as more complex objects
  - (+) Gives good control over clouds shape and location
  - (-) Billboards are flat
  - (-) Lighting is usually precomputed, clouds are static
- Impostors are related concept
  - Pre-renders clouds into camera-facing billboards
Existing methods – Ray Marching

- The cloud density is represented as 3D noise
- Ray marching is performed through the volume to accumulate lighting
  - (+) Good looking result
  - (-) Control over cloud shape and location is intricate
  - (-) Many ray marching steps can be required to eliminate aliasing
  - (-) Lighting usually limited to single scattering
Existing methods – Direct Volume Rendering

- Direct volume rendering methods can be applied to render clouds
- The volume is sliced with planes; the planes are alpha-blended to get final result
- Half-angle slicing can account for occlusion by light at the same time as rendering from the camera
  - (+) Lighting can be rather sophisticated (multiple forward scattering)
  - (-) Control over cloud shape and location is intricate
  - (-) Many slicing planes can be required to eliminate aliasing
Our method

• Attempts to combine control of particle-based approaches with quality of ray marching and slicing techniques

• Key ideas:
  • Use volumetric particles representing the actual 3D-shapes
  • Use physically-based lighting
  • Pre-compute lighting and other quantities to avoid expensive computations at run time
  • Perform volume-aware blending instead of alpha blending
Algorithm overview

Initial step – modeling clouds with spherical particles
Algorithm overview

Add pre-computed cloud density and transparency
Algorithm overview

Add pre-computed light scattering
Algorithm overview

Add light occlusion
Algorithm overview

Add volume-aware blending (enabled by Pixel Sync)
Algorithm overview

Add light scattering
Scattering physics

Light interacts with the tiny (2-8 µm) particles distributed in the cloud:

- A photon can be scattered
  - In-scattering is scattering in the view direction
  - Out-scattering is scattering out of the view direction
- Absorbed
Scattering physics
Scattering physics

Optical depth integral

Light gets attenuated while it travels through the cloud.

Since there is no absorption, only out-scattering attenuates the light.

Optical depth is the amount of scattering matter on the way of light:

$$T(A \to B) = \int_A^B \beta(P) \, ds$$

Transmittance through the cloud is the fraction of light survived out-scattering:

$$L = e^{-T(A \to B)} \cdot L_{In}$$
Scattering physics

In clouds, absorption is negligible and almost all the light is scattered

- The clouds color is defined by the scattered light

Phase function defines direction of a photon after scattering event

- The phase function of cloud particles exhibit strong forward peak
- Almost all light is scattered in forward direction
Scattering physics

Single-scattering integral:

\[ L_{In} = p(\theta) \int_{C}^{0} e^{-T(P \rightarrow C)} \beta(P) L(P) \, ds \]

- \( L(P) \) is the light intensity at point \( P \)
- \( \beta(P) \) is the scattering coefficient at point \( P \)
- \( T(P \rightarrow C) \) is the optical thickness of the media between points \( P \) and \( C \)
- \( p(\theta) \) is the phase function
Scattering physics

Light is also attenuated in the cloud before it reaches the scattering point:

\[ L(P) = L \ e^{-T(A \rightarrow P)} \]

\( L \) is the light intensity outside the cloud.

Let’s now look at our integral:

\[ L_{ln} = p(\theta) \int_{C}^{P} \beta(P) \, ds \]

\[ \int_{A}^{P} \beta(P) \, ds \]

\[ L_{ln} = p(\theta) \int_{C}^{0} e^{-T(P \rightarrow C)} \beta(P) L \ e^{-T(A \rightarrow P)} \, ds \]
Scattering physics

In clouds, a photon is usually scattered multiple times before it leaves the clouds.

This multiple scattering is crucial to cloud appearance and cannot be ignored.

- In contrast, air is much more optically thinner media thus single scattering models produce convincing results.
Scattering physics

Multiple scattering

\[ L = p(\theta) \int_{c}^{0} e^{-T(p \rightarrow c)} \beta(\mathbf{P}) \mathcal{H}(\mathbf{P}) \, ds \]

\[ J(\mathbf{P}) = \int_{\Omega} L p(\theta) d\omega \]

\( \Omega \) is the whole set of directions
Pre-computed lighting

The idea main idea is to

• Precompute physically-based lighting for simple shapes
• Construct clouds from these simple shapes
• The term Particle will now refer to these basic shapes (not individual tiny droplets)
Pre-computing optical depth

Typical way to evaluate optical depth is ray marching

- Impractical to do in real-time

For a known density distribution, the integral can be evaluated once and stored in a look-up table for all possible viewpoints and directions

- No ray marching at run-time
- Fast evaluation for the price of memory

\[ T(A \rightarrow B) = \int_{A}^{B} \beta(P) \, ds \]
Pre-computing optical depth

Parameterization

- We need to describe all start points on the sphere and all directions
- Two angles describe start point on the sphere
- Two angles describe view direction
- 4D look-up table is required

\[ T(A \rightarrow B) = \int_{A}^{B} \beta(P) \, ds \]
Pre-computing optical depth

Integration

- Integration is performed by stepping along the ray and numerically computing optical thickness
  - Cloud density at each step is determined through 3D noise
- 4D look-up table is implemented as 3D texture
  - For look-up, manual filtering across 4\textsuperscript{th} coordinate is necessary

\[ T(A \rightarrow B) = \int_{A}^{B} \beta(P) \, ds \]
Pre-computing optical depth

3D Noise generation

Radial falloff + 3D noise

Thresholding

Pyroclastic style
Pre-computing optical depth

Resolution

- 32x64x32x64 look-up table
- Interpolation artifacts can be visible from close look-ups
- OK from distance
Pre-computing optical depth
Pre-computing scattering

- Let's consider spherically symmetrical particle
- Any start point on the sphere can be described by a single angle
- View direction is described by two angles
- Thus 3 parameters are necessary to describe any start point and view direction -> 3D look-up table

\[ L = \int_{c}^{0} e^{-r(\mathbf{P} \rightarrow \mathbf{C})} \beta(\mathbf{P}) \left( \int_{\Omega} L p(\theta) d\omega \right) ds \]
Pre-computing scattering

Intermediate 4D table is used to store radiance for every point in the sphere

For each scattering order:

1. Compute $J(P)$ for every point and direction inside the sphere by integrating previous order scattering

$$J_n = \int_{\Omega} L_{n-1}(\omega)p(\theta) d\omega$$

2. Compute current order inscattering by numerical integration of $J_n$:

$$L_n = \int_{C} e^{-T(P\rightarrow C)} \beta(P) J_n(P) ds$$

3. Add current scattering order to the total look-up table
Pre-computing scattering

Pre-computed scattering for different light orientations
Pre-computing scattering

Combining pre-computed lighting and pre-computed cloud density
Pre-computing scattering
Compositing clouds
Computing light occlusion
Computing light occlusion

Tiling
- The scene is rasterized from the light over the tile grid
  - One tile is one pixel
- Each particle is assigned to the tile
  - Screen-size buffer is used to store index of the first particle in the list
  - Append buffer is used to store the lists elements
- Pixel Shader Ordering is used to preserve original particle order (sorted from the light)
Computing light occlusion

Traversing lists

- Processing is done by the compute shader
- Each particle finds a tile it belongs to
- The shader then goes through the list of the tile and computes opacity of particles on the light path
- Since particles are ordered from the light, the loop can be terminated as soon as current particle is reached
- The loop can also be terminated when total transparency reaches threshold (0.01)
- Early exit gives up to 2x speed-up for opacity calculation stage
Computing light occlusion
Volume-aware blending

No Pixel Sync – Conventional Alpha Blending
Volume-aware blending

Pixel Sync – Volume-Aware Blending
Volume-aware blending

Blending volumetric particles

- If particles do not overlap, blending is trivial
- How can we correctly blend overlapping particles?
Volume-aware blending

Blending volumetric particles

- Suppose we have two overlapping particles with color and density $C_0, \rho_0$ and $C_1, \rho_1$

- Back:
  - $T_{\text{Back}} = e^{-\rho_1 \cdot d_b \cdot \beta}$
  - $C_{\text{Back}} = C_1 \cdot (1 - T_{\text{Back}})$

- Front:
  - $T_{\text{Front}} = e^{-\rho_0 \cdot d_f \cdot \beta}$
  - $C_{\text{Front}} = C_0 \cdot (1 - T_{\text{Front}})$

- Intersection:
  - $T_{\text{Isec}} = e^{-(\rho_0 + \rho_1) \cdot d_i \cdot \beta}$
  - $C_{\text{Isec}} = \frac{C_0 \rho_0 + C_1 \rho_1}{\rho_0 + \rho_1} \cdot (1 - T_{\text{Isec}})$
Volume-aware blending

Blending volumetric particles

- Final color and transparency:

\[ T_{\text{Final}} = T_{\text{Front}} \cdot T_{\text{Isec}} \cdot T_{\text{Back}} \]

\[ C_{\text{Final}} = \frac{C_{\text{Front}} + C_{\text{Isec}} \cdot T_{\text{Front}} + C_{\text{Back}} \cdot T_{\text{Front}} \cdot T_{\text{Isec}}}{1 - T_{\text{Final}}} \]

- Division by \( 1 - T_{\text{Final}} \) because we do not want alpha pre-multiplied color
Volume-aware blending

- DirectX does not impose any ordering on the execution of pixel shader
  - Ordering happens later at the output merger stage
  - If two threads read and modify the same memory, result is unpredictable
Volume-aware blending

Pixel Shader Ordering assures that

- Read-modify-write operations are protected, i.e. no thread can read the memory before other thread finishes writing to it
- All memory access operations happen in the same order in which primitives were submitted for rendering
Volume-aware blending

Enabling pixel shader ordering

```cpp
#include "IntelExtensions.hls1"
...

void YourPixelShader(...) {
    IntelExt_Init();
    ...
    IntelExt_BeginPixelShaderOrdering();
    // Access UAV
}
```
Volume-aware blending

Blending volumetric particles - Implementation

- Pixel Shader Ordering must be enabled
- Color, density and min/max extent of the current particle are stored in the UAV buffer
- Each new particle is tested against the currently stored
  - If new particle is in front of the current, the current is blended into the back buffer and replaced with the new one
  - If new particle overlaps with the current, they are blended and stored
  - Particles need to be sorted
Volume-aware blending

Blending volumetric particles - Implementation
Volume-aware blending

Blending volumetric particles – comparison with traditional blending
Volume-aware blending
Rendering

Low-resolution rendering

- To improve performance, particles are rendered to a low-resolution buffer
- Bilateral filtering is then performed to upscale to original resolution and preserve edges
Particle generation

Cell grid

- Organized as a number of concentric rings centered around the camera
- Particles in each next ring have twice the size of the inner ring
- Each cell contains several layers of particles
- Density and size of particles in each cell are determined by the noise texture
Particle generation

Steps:

- Process cell grid and create a list of valid (non-empty) cells
  - One compute shader thread processes one cell
  - Append buffer is used to store indices of valid cells
Particle generation

Steps:
- Process each valid cell and create a list of valid particles in each cell
  - Use DispatchIndirect() to execute the required number of threads on GPU
  - One thread processes one valid cell and generates several particles
Particle generation

Animation:
Clouds are animated by changing particle size and transparency
Particle Rendering

Particle ordering

- Particles must be rendered in back to front order
  - Sorting on the GPU is very expensive
- We can sort cells on the CPU
  - Not all cells contain actual particles
- Solution:
  - Output particles only for valid cells
  - Use stream-out to preserve order
  - Process 32 particles by one GS thread
Particle Rendering

Particle processing

- DispatchIndirect() is used to execute CS computing light opacity for each valid particle
- DispatchIndirect() is used to execute CS computing visibility for each valid particle
Integration with light scattering technique

- Cloud density texture is rendered from light.
- At each ray marching step, it is determined if a point is above or under the cloud (clouds are assumed to have constant altitude).
- If point is under the clouds, the cloud density texture is sampled to get the occlusion by clouds.
- Cloud transparency and distance to clouds in screen-space are used to attenuate scattering along view rays.
Integration with light scattering technique
Integration with light scattering technique
Results
Results
Results
Results
Results
Performance

Pre-computation

Computing optical depth integral takes less than 100 ms

- Switching between different noise generation methods can be done at run time

Pre-computing scattering requires several minutes

- Final look-up table is only 1 MB and thus can be distributed with the application
Performance

3.5 ms on Iris Pro 5200, 1280x720

Grid size: 136x136x4x4; Half resolution rendering
Performance

12 ms on Iris Pro 5200, 1280x720

Grid size: 136x136x4x4; Half resolution rendering
Questions?

Thank You

Keep in touch with us at GDC and beyond:

• Game Developer Conference
  Visit our Intel® booth #1016 in Moscone South

• Intel University Games Showcase
  Marriott Marquis Salon 7, Thursday 5:30pm
  RSVP at bit.ly/intelgame

• Intel Developer Forum, San Francisco
  September 9-11, 2014
  intel.com/idf14

• Intel Software Adrenaline
  @inteladrenaline

• Intel Developer Zone
  software.intel.com
  @intelsoftware