02562 Rendering - Introduction

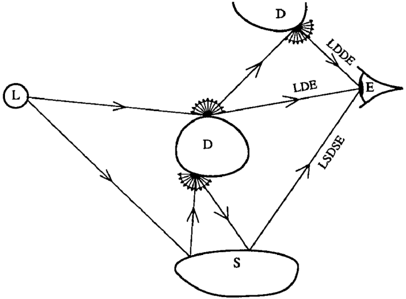
Progressive Unidirectional Path Tracing

Jeppe Revall Frisvad

October 2023

Global illumination

©Disney Animation Studios



Heckbert's light transport notation

L	–	Light
E	–	Eye
D	–	Diffuse surface
S	–	Specular surface
$*$	–	0 or more interactions
$+$	–	1 or more interactions
$?$	–	0 or 1 interaction
$\mid$	–	either the path on the left or the right side

- Global illumination includes all light paths: $L(S|D)*E$.

References:

- Heckbert, P. S. Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (SIGGRAPH 90) 24*(4), pp. 145–154. 1990.

The rendering equation

- The equation we are solving is [Nicodemus 1965, Kajiya 1986]

$$L_o(\boldsymbol{x}, \vec{\omega}_o) = L_e(\boldsymbol{x}, \vec{\omega}_o) + \int_\Omega f_r(\boldsymbol{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\boldsymbol{x}, \vec{\omega}_i) \cos\theta_i \, \mathrm{d}\omega_i \,,$$

- where

L_o is observed radiance,
L_e is emitted radiance,
L_i is incident radiance,
$\boldsymbol{x}$ is a surface position,
$\vec{\omega}_o$ is the direction toward the observer (direction of observation),
$\vec{\omega}_i$ is the direction toward the light source (direction of incidence),
f_r is the bidirectional reflectance distribution function (BRDF),
$\mathrm{d}\omega_i$ is a differential element of solid angle,
Ω is the 2π solid angle around the surface normal $\vec{n}$ at $\boldsymbol{x}$,
θ_i is the angle between $\vec{\omega}_i$ and the surface normal $\vec{n}$ at $\boldsymbol{x}$, such that $\cos\theta_i = \vec{\omega}_i \cdot \vec{n}$.

References

- Nicodemus, F. E. Directional reflectance and emissivity of an opaque surface. *Applied Optics 4*(7), pp. 767–775. July 1965.
- Kajiya, J. The rendering equation. *Computer Graphics (SIGGRAPH 86) 20*(4), pp. 143–150. August 1986.

Monte Carlo integration (the sampling approach)

- The rendering equation:

$$L_o(\boldsymbol{x}, \vec{\omega}_o) = L_e(\boldsymbol{x}, \vec{\omega}_o) + \int_{2\pi} f_r(\boldsymbol{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\boldsymbol{x}, \vec{\omega}_i) \cos\theta_i \,\mathrm{d}\omega_i \,.$$

- The Monte Carlo estimator:

$$L_N(\boldsymbol{x}, \vec{\omega}_o) = L_e(\boldsymbol{x}, \vec{\omega}_o) + \frac{1}{N} \sum_{j=1}^{N} \frac{f_r(\boldsymbol{x}, \vec{\omega}_{ij}, \vec{\omega}_o) L_i(\boldsymbol{x}, \vec{\omega}_{ij}) \cos\theta_i}{\mathrm{pdf}(\vec{\omega}_{ij})}$$

  with $\cos\theta_i = \vec{\omega}_{ij} \cdot \vec{n}$, where $\vec{n}$ is the surface normal at $\boldsymbol{x}$.

- The Lambertian BRDF:

$$f_r(\boldsymbol{x}, \vec{\omega}_i, \vec{\omega}_o) = \rho_d / \pi \,.$$

- A good choice of pdf would be:

$$\mathrm{pdf}(\vec{\omega}_{ij}) = \cos\theta_i / \pi \,.$$

Splitting the evaluation

- Distinguishing between:
  - Direct illumination L_{direct}.
    - Light reaching a surface directly from the source.
  - Indirect illumination L_{indirect}.
    - Light reaching a surface after at least one bounce.
- The rendering equation is then

$$L_o = L_e + L_{\text{direct}} + L_{\text{indirect}} .$$

  - L_e is emission.
  - L_{direct} is sampling of lights.
  - L_{indirect} is sampling of the BRDF excluding lights.
- You need an `emit` flag in your `HitInfo` struct to avoid adding emission when tracing indirect illumination.

Path tracing diffuse objects

- The diffuse BRDF: $f_r = \rho_d / \pi$.
- Computing direct illumination:
  - Sample positions in random light source triangles (Δ out of n): $\text{pdf}(\boldsymbol{x}_j) = \frac{1}{n}\frac{1}{A_{\Delta,j}}$.
  - Estimator for L_{direct} is

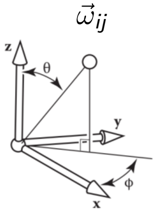
$$L_{\text{direct},N} = \frac{\rho_d(\boldsymbol{x})}{\pi}\frac{1}{N}\sum_{j=1}^{N} L_e(\boldsymbol{x}_j \to \boldsymbol{x}) V(\boldsymbol{x}_j \leftrightarrow \boldsymbol{x}) \frac{\cos\theta_i \cos\theta_{\text{light}}}{\|\boldsymbol{x} - \boldsymbol{x}_j\|^2} n A_{\Delta,j} \,.$$

- Computing indirect illumination:
  - Set $L_e = 0$.
  - Sample directions using cosine-weighted hemisphere: $\text{pdf}(\vec{\omega}_{ij}) = \cos\theta_i / \pi$.

$$\vec{\omega}_{ij} \quad \text{from} \quad (\theta_i, \phi_i) = (\cos^{-1}\sqrt{\xi_1},\, 2\pi\xi_2)\,, \quad \xi_1, \xi_2 \in [0,1]\,.$$

  - Estimator for L_{indirect} is

$$L_{\text{indirect},N} = \rho_d(\boldsymbol{x})\frac{1}{N}\sum_{j=1}^{N} L_i(\boldsymbol{x}, \vec{\omega}_{ij})\,.$$

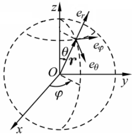


Sampling a cosine-weighted hemisphere

- We sample ${}_{\perp}\vec{\omega}_{ij}$ on a cosine-weighted hemisphere using spherical coordinates (θ, ϕ) in the local tangent space of surface point with normal $\vec{n} = (n_x, n_y, n_z)$.

$$(\theta, \phi) = (\cos^{-1}\sqrt{1-\xi_1}, 2\pi\xi_2) \quad , \quad \xi_1, \xi_2 \in [0, 1) \quad \text{(random numbers)}\,.$$

- In Euclidian coordinates, the corresponding vector in tangent space is



$${}_{\perp}\vec{\omega}_{ij} = (x, y, z) = (\cos\phi\sin\theta, \sin\phi\sin\theta, \cos\theta)\,.$$

- We can now make a change of basis from tangent space to world space using a formula derived from quaternion rotation [Frisvad 2012, Duff et al. 2017]:

$$\vec{\omega}_{ij} = \left(x \begin{bmatrix} 1 - n_x^2/(1+|n_z|) \\ -n_x n_y/(1+|n_z|) \\ -n_x \operatorname{sgn}(n_z) \end{bmatrix} + y \begin{bmatrix} -n_x n_y/(1+|n_z|)\operatorname{sgn}(n_z) \\ (1 - n_y^2/(1+|n_z|))\operatorname{sgn}(n_z) \\ -n_y \end{bmatrix} + z \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \right)\,.$$

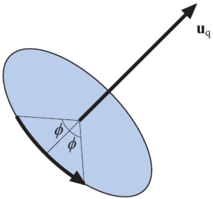
- Use a pseudo-random number generator to get $\xi_1, \xi_2 \in [0, 1)$ and `rotate_to_normal` for the quaternion rotation.

What is a quaternion?

- Generalization of complex numbers for representation of rotations.
- Specified as a 4-vector with a vector imaginary part $\boldsymbol{q}_v$ and a scalar real part q_w:

$$\hat{\boldsymbol{q}} = (\boldsymbol{q}_v, q_w) = iq_x + jq_y + kq_z + q_w \,, \quad \boldsymbol{q}_v = (q_x, q_y, q_z)\,, \quad i^2 = j^2 = k^2 = ijk = -1\,.$$

Multiplication:	$\hat{\boldsymbol{q}}\hat{\boldsymbol{r}}$	$=$	$(\boldsymbol{q}_v \times \boldsymbol{r}_v + r_w \boldsymbol{q}_v + q_w \boldsymbol{r}_v, q_w r_w - \boldsymbol{q}_v \cdot \boldsymbol{r}_v)\,.$
Addition:	$\hat{\boldsymbol{q}} + \hat{\boldsymbol{r}}$	$=$	$(\boldsymbol{q}_v + \boldsymbol{r}_v, q_w + r_w)\,.$
Conjugate:	$\hat{\boldsymbol{q}}^*$	$=$	$(-\boldsymbol{q}_v, q_w)\,.$
Norm:	$\mathrm{norm}(\hat{\boldsymbol{q}})$	$=$	$\sqrt{\hat{\boldsymbol{q}}\hat{\boldsymbol{q}}^*} = \sqrt{\boldsymbol{q}_v \cdot \boldsymbol{q}_v + q_w^2}\,.$
Identity:	$\hat{\boldsymbol{\imath}}$	$=$	$(\boldsymbol{0}, 1)\,.$
Inverse:	$\hat{\boldsymbol{q}}^{-1}$	$=$	$\hat{\boldsymbol{q}}^* / [\mathrm{norm}(\hat{\boldsymbol{q}})]^2\,.$

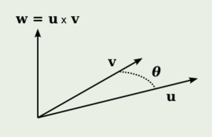


- Unit quaternion specifying rotation of 2ϕ radians around an axis $\boldsymbol{u}_q$:

$$\hat{\boldsymbol{q}} = (sin\phi\, \boldsymbol{u}_q, \cos\phi)\,.$$

- Use homogeneous coordinates to convert a position or direction vector $\boldsymbol{p} = (p_x, p_y, p_z, p_w)$ to a quaternion $\hat{\boldsymbol{p}}$.
- Apply quaternion rotation using $\hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^{-1}$. For a unit quaternion, use $\hat{\boldsymbol{q}}\hat{\boldsymbol{p}}\hat{\boldsymbol{q}}^*$.

Rotating from tangent space to world space

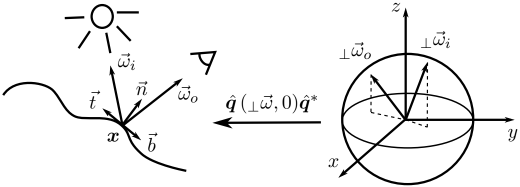


- Unit quaternion specifying rotation from $\boldsymbol{u}$ to $\boldsymbol{v}$:

$$\hat{\boldsymbol{q}} = \left(\sin\frac{\theta}{2}\boldsymbol{w}, \cos\frac{\theta}{2}\right) = \left(\frac{\boldsymbol{u}\times\boldsymbol{v}}{\sqrt{2(1+\boldsymbol{u}\cdot\boldsymbol{v})}}, \frac{1}{2}\sqrt{2(1+\boldsymbol{u}\cdot\boldsymbol{v})}\right).$$

- Let us set $\boldsymbol{u} = (0,0,1)$ and $\boldsymbol{v} = \vec{n} = (n_x, n_y, n_z)$, then

$$\hat{\boldsymbol{q}} = \left(\frac{(-n_y, n_z, 0)}{\sqrt{2(1+n_z)}}, \frac{1}{2}\sqrt{2(1+n_z)}\right).$$



- Applying $\hat{\boldsymbol{q}}$ to a direction vector ${}_\perp\vec{\omega} = (\omega_x, \omega_y, \omega_z)$, we rotate from tangent to world space: $\hat{\boldsymbol{q}}\,({}_\perp\vec{\omega}, 0)\hat{\boldsymbol{q}}^*$.

- This simplifies to the formula for `rotate_to_normal` [Frisvad 2012, Duff et al. 2017].

References:

- Frisvad, J. R. Building an orthonormal basis from a 3D unit vector without normalization. *Journal of Graphics Tools 16*(3), pp. 151–159. August 2012.
- Duff, T., Burgess, J., Christensen, P., Hery, C., Kensler, A., Liani, M., and Villemin, R. Building an orthonormal basis, revisited. *Journal of Computer Graphics Techniques 6*(1), pp. 1–8. March 2017.

Progressive unidirectional path tracing

1. **Generate rays** from the eye through pixel positions
2. **Trace the rays** and evaluate the rendering equation for each ray.
3. **Randomize the position** within the pixel area to Monte Carlo integrate (measure) the radiance arriving in a pixel.

- Path tracing is the idea of using $N = 1$ in the Monte Carlo estimators for the reflected radiance to generate a path instead of a tree.
- Noise is reduced by progressive updates of the measurement.
- Update the rendering result in a pixel L_j after rendering a new frame with result L_{new} using

$$L_{j+1} = \frac{L_{\text{new}} + jL_j}{j+1} \,.$$

- Progressive (stop and go) rendering is convenient for several reasons:
  - No need to start over.
  - Result can be stored and refined later if need be.
  - Convergence can be inspected during progressive updates.

Stopping paths probabilistically (Russian roulette)

- Do either one thing or another (not both).
- Sample an event.
  - Example: Reflection or absorption.
  - Example: Reflection or transmission.
- How? Sample a step function.
- What are the steps? The probabilities that events occur.
  - Example: Either diffuse reflection or absorption.
  - What are the probabilities?
  - The simplest idea is the average of the diffuse reflectance:

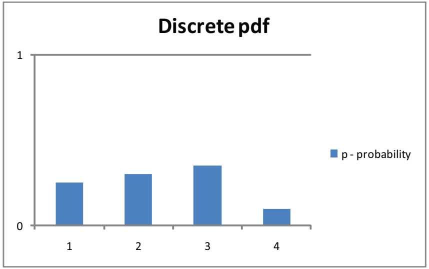
$$\text{probability of diffuse reflection} = \frac{\rho_{d,R} + \rho_{d,G} + \rho_{d,B}}{3} .$$

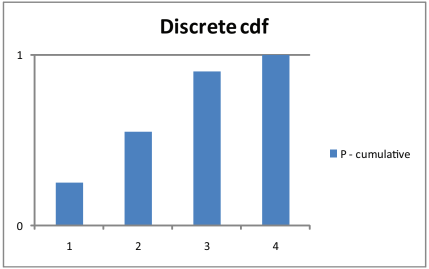
  - The probability of absorption is then one minus the probability of diffuse reflection (the else-case).
  - In the case of diffuse reflection, a ray with sampled direction is traced to evaluate the indirect illumination.

Sampling a step function (Russian roulette)

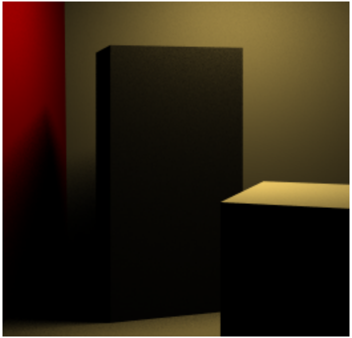
Algorithm:

sample $\xi \in [0, 1]$ uniformly;
if $(\xi < P_1)$
  call event 1;
  divide by p_1;
else if $(\xi < P_2)$
  call event 2;
  divide by p_2;
else if $(\xi < P_3)$
  ...
else if $(\xi < P_4)$
  ...

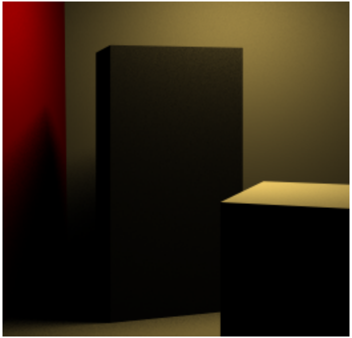




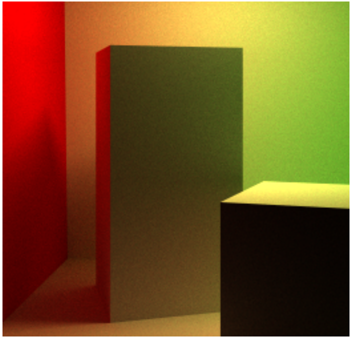
Direct versus global illumination (and jittered versus progressive)



progressive
1 minute
400 samples



jittered
1 minute
484 samples



progressive
13.4 minutes
1000 samples

- The worst case convergence is an error proportional to $\frac{1}{\sqrt{N}}$, where N is the number of samples.

How to get pseudo-random numbers on the GPU?

- We need a seeded pseudo-random number generator (PRNG) and a different seed for each thread.
- In rendering, a natural choice is one thread for each pixel in each frame.
- Given pixel index and frame number, we can use Marsaglia's xorshift bit scrambling to get a pseudo-random seed for the PRNG:

```
// PRNG xorshift seed generator by NVIDIA
fn tea(v0: u32, v1: u32) -> u32
{
  const N = 16u; // User specified number of iterations
  var s0 = 0u;
  for(var n = 0u; n < N; n++) {
    s0 += 0x9e3779b9;
    v0 += ((v1<<4)+0xa341316c)^(v1+s0)^((v1>>5)+0xc8013ea4);
    v1 += ((v0<<4)+0xad90777d)^(v0+s0)^((v0>>5)+0x7e95761e);
  }
  return v0;
}
```

- With a seed, we can use a multiplicative congruential PRNG:

$$Z_i = AZ_{i-1} \mod M$$

if we can find a good multiplier A for a modulus $M \in [0, 2^{32})$.

The good multiplier for a multiplicative congruential generator

- The good multiplier should have a full period (reach all integers from 0 to $M-1$) and exhibit good equidistribution if we use it to sample points in a kD hypercube.
- Fishman [1990] performed an exhaustive analysis to find the best multiplier for $M = 2^{32}$. This is good if we want random numbers $\xi \in [0,1]$.
- However, we want $\xi \in [0,1)$ (e.g. when sampling an index $i < n$ by $i = \lfloor \xi n \rfloor$).
- Hui-Chin Tang [2007] performed an exhaustive search to find the best multiplier for $M = 2^{31} - 1$ with good equidistribution (for $1 < k \leq 8$). Then

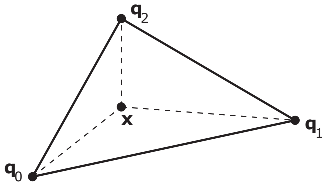
```
// Generate random unsigned int in [0, 2^31-1)
fn mcg31(prev: ptr<function, u32>) -> u32
{
  const LCG_A = 1977654935u; // Multiplier from Hui-Ching Tang [EJOR 2007]
  *prev = (LCG_A * (*prev)) & 0x7FFFFFFE;
  return *prev;
}

// Generate random float in [0, 1)
fn rnd(prev: ptr<function, u32>) -> f32
{
  return f32(mcg31(prev)) / f32(0x80000000);
}
```

Sampling a triangle mesh

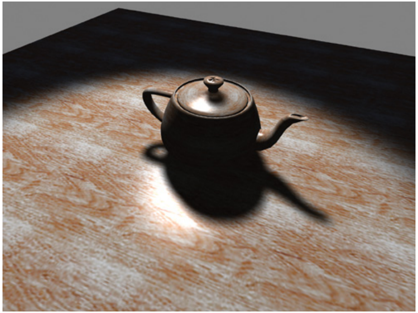
- Uniformly sample a triangle index $i = \lfloor \xi n \rfloor$ ($\text{pdf}(\Delta) = 1/n$, where n is the number of triangles in the mesh).
- Uniformly sample a position on the triangle ($\text{pdf}(\boldsymbol{x}_{\ell,j,\Delta}) = 1/A_\Delta$, where A_Δ is the triangle area):
  1. Sample barycentric coordinates $(\alpha, \beta, \gamma = 1 - \alpha - \beta)$:

$$\begin{aligned} \alpha &= 1 - \sqrt{\xi_1} \\ \beta &= (1 - \xi_2)\sqrt{\xi_1} \\ \gamma &= \xi_2\sqrt{\xi_1} \end{aligned}$$



  2. Use the barycentric coordinates for linear interpolation of triangle vertices $(\boldsymbol{q}_0, \boldsymbol{q}_1, \boldsymbol{q}_2)$ to obtain a point in the triangle: $\boldsymbol{x}_{\ell,j} = \alpha\boldsymbol{q}_0 + \beta\boldsymbol{q}_1 + \gamma\boldsymbol{q}_2$.
  3. Use the barycentric coordinates for linear interpolation of triangle vertex normals to obtain the normal $\vec{n}_{\ell,j}$ at the sampled surface point. Normalize after interpolation.
- The pdf of this sampling of the surface is: $\text{pdf}(\boldsymbol{x}_{\ell,j}) = \text{pdf}(\Delta)\text{pdf}(\boldsymbol{x}_{\ell,j,\Delta}) = \dfrac{1}{n}\dfrac{1}{A_\Delta}$.

Soft shadows



- Sampler:

$$\vec{\omega}'_j = \frac{\mathbf{x}_{\ell,j} - \mathbf{x}}{\|\mathbf{x}_{\ell,j} - \mathbf{x}\|} .$$

- From solid angle to area:

$$\mathrm{d}\omega' = \frac{\cos\theta_\ell}{r^2}\mathrm{d}A = \frac{\vec{n}_{\ell,j}\cdot(-\vec{\omega}'_j)}{\|\mathbf{x}_{\ell,j} - \mathbf{x}\|^2}\mathrm{d}A .$$

- Using the Lambertian BRDF, $f_r = \rho_d/\pi$, and triangle mesh sampling of a point $\mathbf{x}_{\ell,j}$ on the light, $\mathrm{pdf}(\mathbf{x}_{\ell,j}) = 1/(nA_\Delta)$, the Monte Carlo estimator for area lights is:

$$\begin{aligned} L_N(\mathbf{x},\vec{\omega}) &= \frac{1}{N}\sum_{j=1}^{N}\frac{f_r(\mathbf{x},\vec{\omega}'_j,\vec{\omega})L_i(\mathbf{x},\vec{\omega}'_j)\cos\theta\cos\theta_\ell}{\mathrm{pdf}(\mathbf{x}_{\ell,j})\ r^2} \\ &= \frac{\rho_d(\mathbf{x})}{\pi}\frac{1}{N}\sum_{j=1}^{N}\underbrace{L_e(\mathbf{x}_{\ell,j},-\vec{\omega}'_j)V(\mathbf{x},\mathbf{x}_{\ell,j})\frac{\vec{n}_{\ell,j}\cdot(-\vec{\omega}'_j)}{\|\mathbf{x}_{\ell,j}-\mathbf{x}\|^2}nA_\Delta}_{\text{returned as } L \text{ from area light sampler}}(\vec{\omega}'_j\cdot\vec{n}) . \end{aligned}$$

Exercises

- Use ping-pong rendering for progressive updates with sampling of a random positions in each pixel (replaces stratified jitter sampling).
- Implement area light triangle mesh sampling to render soft shadows.
- Implement sampling of a cosine-weighted hemisphere.
- Write a shader for Lambertian materials that includes path-traced indirect illumination.

Render-to-texture and ping-ponging (pipeline)

- The pipeline changes: we add an `f32` texture target for our render results.

```
const pipeline = device.createRenderPipeline({
  layout: "auto",
  vertex: {
    module: wgsl,
    entryPoint: "main_vs",
  },
  fragment: {
    module: wgsl,
    entryPoint: "main_fs",
    targets: [ { format: canvasFormat },
               { format: "rgba32float" } ]
  },
  primitive: {
    topology: "triangle-strip",
  },
});
```

Render-to-texture and ping-ponging (textures)

- Create one texture as a render target and one to load the previous result from.

```
let textures = new Object();
textures.width = canvas.width;
textures.height = canvas.height;
textures.renderSrc = device.createTexture({
  size: [canvas.width, canvas.height],
  usage: GPUTextureUsage.RENDER_ATTACHMENT | GPUTextureUsage.COPY_SRC,
  format: 'rgba32float',
});
textures.renderDst = device.createTexture({
  size: [canvas.width, canvas.height],
  usage: GPUTextureUsage.TEXTURE_BINDING | GPUTextureUsage.COPY_DST,
  format: 'rgba32float',
});
```

- The render target is a source for copying data to the destination texture binding.
- Use no sampler for the texture binding.

Render-to-texture and ping-ponging (render pass)

- The render pass changes: we include a second color attachment and a texture-to-texture copy.

```
function render(device, context, pipeline, textures, bindGroup, frame)
{
  const encoder = device.createCommandEncoder();
  const pass = encoder.beginRenderPass({
    colorAttachments: [
      { view: context.getCurrentTexture().createView(), loadOp: "clear", storeOp: "store" },
      { view: textures.renderSrc.createView(), loadOp: frame === 0 ? "clear" : "load", storeOp: "store" }]
  });
  pass.setPipeline(pipeline);
  pass.setBindGroup(0, bindGroup);
  pass.draw(4);
  pass.end();

  encoder.copyTextureToTexture({ texture: textures.renderSrc }, { texture: textures.renderDst },
    [textures.width, textures.height]);

  // Finish the command buffer and immediately submit it.
  device.queue.submit([encoder.finish()]);
}
```

- Note that we only clear the progressively updated render result if the frame number is zero (`frame === 0`).

Render-to-texture and ping-ponging (fragment shader)

- The fragment shader implements the progressive updating.

```
struct FSOut {
  @location(0) frame: vec4f,
  @location(1) accum: vec4f,
};

@fragment
fn main_fs(@builtin(position) fragcoord: vec4f, @location(0) coords: vec2f) -> FSOut
{
  let launch_idx = u32(fragcoord.y)*uniforms_ui.width + u32(fragcoord.x);
  var t = tea(launch_idx, uniforms_ui.frame);
  let jitter = vec2f(rnd(&t), rnd(&t))/f32(uniforms_ui.height);
  ⋮

  // Progressive update of image
  let curr_sum = textureLoad(renderTexture, vec2u(fragcoord.xy), 0).rgb*f32(uniforms_ui.frame);
  let accum_color = (result + curr_sum)/f32(uniforms_ui.frame + 1u);

  var fsOut: FSOut;
  fsOut.frame = vec4f(pow(accum_color, vec3f(1.0/uniforms_f.gamma)), 1.0);
  fsOut.accum = vec4f(accum_color, 1.0);
  return fsOut;
}
```

- The texture binding is in `renderTexture`. The texture target is in location 1 of the output. The frame resolution and number are uploaded as uniforms.