Project 5: History tracking
Project 6: Denoising

Why two projects in one?

Both projects 5 and 6 need access to more data than is currently recorded.

- This section creates the needed extra buffers, fills them with data, and tracks (i.e., copies) current and previous versions.
- The next section uses that data to replace one line in the ray generation shader.
- The last section uses that data for the denoising algorithm to be implemented in a compute shader.

New image buffers:

The current color buffer **m_rtColCurrBuffer** gains some new companion buffers. The total of six buffers are described here along with a suggested naming convention:

	3 **Curr** buffers store output of a path tracing pass:	3 **Prev** buffers will be copies of the previous frame's output.
Col buffers store the accumulated color output of the path tracing alg.	m_rtColCurrBuffer	m_rtColPrevBuffer
Nd buffers store the normal/depth of the front most object at a pixel.	m_rtNdCurrBuffer	m_rtNdPrevBuffer
Kd buffers will store the surface Kd of the front most object at a pixel.	m_rtKdCurrBuffer	m_rtKdPrevBuffer

Create these 5 new buffers in **createRtBuffers** following the same pattern as the existing **m_rtColCurrBuffer.** Destroy them where you destroyed **m_rtColCurrBuffer.**

In VkApp::createRtDescriptorSet:

Pass all six buffers to the ray generation shader via the **m_rtDesc** descriptor set. The 5 new buffers follow the pattern established by the existing **m_rtColCurrBuffer.** Be sure that the "binding" number (shown in yellow below) increments through successive integers. (Remember, bindings 0, 1, and 2 are already taken by the acceleration structure, the existing **m_rtColCurrBuffer** image buffer, and the light list.)

The setBindings pattern:

```
{1, VK_DESCRIPTOR_TYPE_STORAGE_IMAGE, 1,
        VK_SHADER_STAGE_RAYGEN_BIT_KHR},
```

The write pattern (implemented right after the setBindings call):

```
m_rtDesc.write(m_device, 1, m_rtColCurrBuffer.Descriptor());
```

The GLSL ray generation shader's pattern:

```
layout(set=0, binding=1, rgba32f) uniform image2D colCurr; //image: m_rtColCurrBuffer
```

VkApp::raytrace

At the end of **VkApp::raytrace**, find the line that copies the ray trace output buffer to the scanline buffer (from which the swap chain displays)

CmdCopyImage(m_rtColCurrBuffer, m_scImageBuffer);

Add three new copy commands from the 3 Curr buffers to the 3 Prev buffers, respectively:

CmdCopyImage(m_rtColCurrBuffer, m_rtColPrevBuffer);
CmdCopyImage(m_rtNdCurrBuffer, m_rtNdPrevBuffer);
CmdCopyImage(m_rtKdCurrBuffer, m_rtKdPrevBuffer);

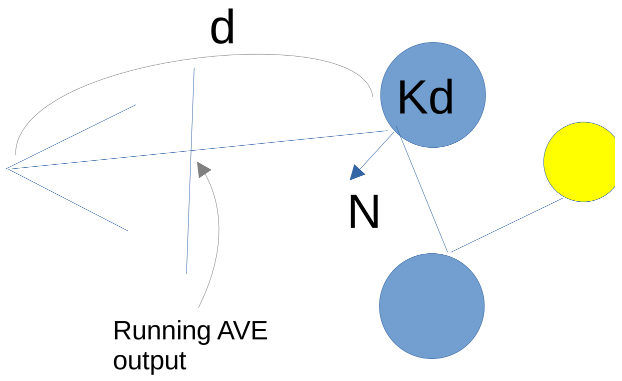
The ray generation shader raytrace.rgen

At this point, the ray generation shader can generate output into the three **Curr** buffers, with this knowledge:

- While writing contents to the 3 Curr buffers, the algorithm can access the previous frame's contents in the 3 Prev buffers, and
- Whatever contents is written to the 3 Curr buffers this frame will be available next frame in the 3 Prev buffers.

The contents of the 3 buffers should be:

- **m_rtColCurrBuffer:** The output of the path tracing algorithm – the accumulated **average:count** produced by the Monte-Carlo calculation.
- **m_rtNdCurrBuffer:** The normal and depth of the first object visible at a pixel.
- **m_rtKdCurrBuffer:** The surface color (Kd) of the first object visible at a pixel. Note: Kd may be from the material directly, or from the material's texture.



To generate these values, in the ray generation shader, find the Monte-Carlo loop in **shader/raytrace.rgen**.

```
for (int i=0; i<pcRay.depth;  i++) {…}
```

In the first pass through that loop (i.e., when **i==0**), and somewhere after the call to **rtraceRayEXT** gather these values:

if (i == 0) {

- **firstHit = payload.hit** A boolean, true if the first ray hit something.
- **firstDepth:** from the hit-shader's **gl_HitTEXT** passed back in the ray's payload structure
- **firstNrm, firstKd**: the normal and surface color at the first hit point, calculated in a previous project. Make sure the **firstKd** is the color **after** the texture lookup, if any.)

}

After the loop, when the output color C has been accumulated into the running average **newAve** along with its count **newC**, write the values to the 3 Curr buffers:

```
imageStore(colCurr,ivec2(gl_LaunchIDEXT.xy), vec4(newAve,newN));
imageStore(kdCurr, ivec2(gl_LaunchIDEXT.xy), vec4(firstKd,0));
imageStore(ndCurr, ivec2(gl_LaunchIDEXT.xy), vec4(firstNrm,firstDepth));
```

As an added precaution, I suggest testing for illegal values (infinities and NANs) before writing any of those values to the output image. Like this:

if (any(isnan(newAve)) || any(isinf(newAve))) …

Ray payload structure

Modify the ray payload (defined in **shaders/shared_structs.h**) to include a float value for communicating a ray's t parameter of intersection back to the calling shader.   Modify the hit-shader to fill in this value:

payload.hitDist = gl_HitTEXT;

Use this value as the **firstDepth** in the previous section.

5 History Tracking

Notes:

- All of project 5 is to replace this single line of code
  vec4 old = imageLoad(colCurr, ivec2(gl_LaunchIDEXT.xy)); //Running average:N
  with a line that reaches into the past to retrieve the running average from it's position in the previous frame:
  vec4 old = <*the selectively weighted bilinear combination of 4 pixels from colPrev*>
  or, if those pixels can't be found, then
  vec4 old = (0,0,0,0)
  to start the accumulation from scratch for this newly visible pixel.
- There will be three cases where the attempt to find an old running average fails. In those cases we will restart the accumulation, but camera motion will no longer force this.
- We will accumulate lighting calculations whenever possible, so remove the project 4 line that cleared the screen when the camera moved
  if (pcRay.clear) <write C,1 to colCurr>

Accumulating this frame's C value into a previous accumulation

In brief, to accumulate path tracing results across frames when the eye point moves, we will;

1. (NEW) Back-project the current frame's world position into the previous frame.
2. (NEW) Calculate the previous frame's accumulation as a weighted average of 4 pixels in a 2x2 pattern around the back-projected point.
3. (NEW) For the three failure cases (where the back-projection fails to find pixels to average), restart the current pixel's accumulation.
4. (OLD) Otherwise, sum the current frame's C value into the retrieved accumulation.
5. (OLD) Write the result to this frame's output image.

In detail:

1. **Back-project the current frame's world position into the previous frame:**
   The **updateCameraBuffer** function produces both the current and previous frame's transformations in **mats.viewProj**, and **mats.priorViewProj**, so:
   ```
   vec4 screenH = (mats.priorViewProj * vec4(firstPos, 1.0)); //Project to prev buffers
   vec2 screen  = ((screenH.xy/screenH.w) + vec2(1.0)) / 2.0; //H-division and map to [0,1]
   ```
   Failure case 1: If **firsthit** is false, then **firstPos** is not valid.
   Failure case 2: If **screen.x** or **screen.y** are outside the [0,1] range, then **firsthit** projects off screen, so no previous-frame lookup can be performed.

2. **Calculate the previous frame's accumulation:**
   See (*) below for the details of calculating **vec4 P** as a weighted average of 4 (or fewer) previous frame accumulation values. Separate the result into
   ```
   oldAve = P.xyz;
   oldN = P.w;  // Beware: this will no longer be an integer. This is not a problem!
   ```
   Failure case 3: The calculation of P finds no valid pixels to average.

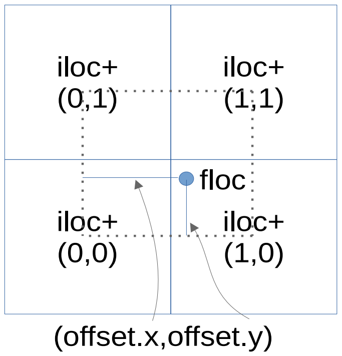
3. **Failure cases:** In any of the 3 failure cases mentioned above, restart the accumulation from scratch with
   ```
   oldAve = vec3(0.5);  // To soften the blackness of newly de-occluded pixels.
   oldN = 1;
   ```

4. **Sum the current frame's C value into that previous frame's accumulation.** As before
   ```
   newAve  = oldAve + (C – oldAve)/newN;
   newN = oldN + 1;
   ```

5. **Write the result to this frame's output image with**
   ```
   imageStore(colCurr, ivec2(gl_LaunchIDEXT.xy), vec4(newAve,newN));
   ```

Calculate the previous frame's accumulation P = <oldAve,oldN>

We calculate a previous accumulation value as a **selectively weighted bilinear interpolation** of four neighboring pixels in **colPrev** buffer. The projection may have failed for two reasons – see above for how to handle those cases. Here we assume the projection succeeded and the variable **screen** is a valid position on the previous screen.



The four neighboring pixels:

vec2 floc = screen * gl_LaunchSizeEXT.xy - vec2(0.5);
vec2 offset = fract(floc); // 0 to 1 offset between 4 neighbors.
ivec2 iloc = ivec2(floc); // (0,0) corner of the 4 neighbors.

The oldAve,oldN pair

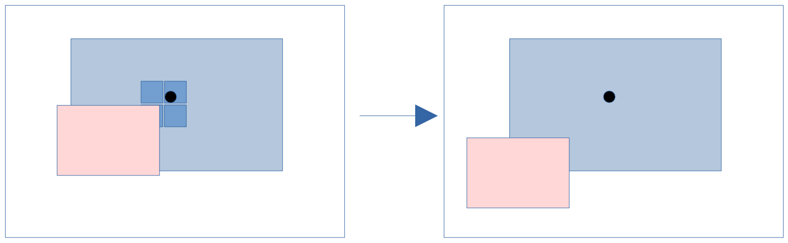
output from this step is calculated as the weighted average of four neighboring pixels:

$$P = <oldAve, oldN> = \frac{w_{0,0}\,P_{iloc+(0,0)} + w_{1,0}\,P_{iloc+(1,0)} + w_{0,1}\,P_{iloc+(0,1)} + w_{1,1}\,P_{iloc+(1,1)}}{w_{0,0} + w_{1,0} + w_{0,1} + w_{1,1}} \qquad (*)$$

where the weights $w_{i,j}$ are shown below, and the values $P_{iloc+(i,j)}$ are pixels read from **colPrev** via
imageLoad(colPrev, iloc+ivec2(i,j)) // Beware iloc+(i,j)) is something very different!

The selective weights:

The selective weights compare the current pixel's normal and depth (**firstNrm**, and **firstDepth**) with each of the 4 previous pixels recorded normal and depth, and forms a weight judging if the previous pixel should contribute to the weighted average.


Previous frame Current frame

Each weight $w_{i,j}$ is a product of three individual weights:

- $b_{i,j}$: a bilinear weight, one of
$$b_{0,0} = (1-offset.x)*(1-offset.y)$$
$$b_{1,0} = (\;\;offset.x)*(1-offset.y)$$
$$b_{0,1} = (1-offset.x)*(\;\;offset.y)$$
$$b_{1,1} = (\;\;offset.x)*(\;\;offset.y)$$
- a depth related weight: `if` $\text{if } |\text{firstDepth} - \text{prevDepth}| < d_{threshold} \text{ then 1 else 0}$
- a normal related weight: $\text{if}\,(\text{firstNrm}\cdot\text{prevNrm}) > n_{threshold} \quad \text{then 1 else 0}$

where

```
prevNd    = imageLoad(ndPrev, iloc+ivec2(i,j));
prevNrm   = prevNd.xyz;
prevDepth = prevNd.w;
```

Good values for the thresholds seem to be: $n_{threshold} = 0.95$ and $d_{threshold} = 0.15$

Denoising via the À-Trous algorithm

Note: New shader denoise.comp

This project is implemented in the compute shader **denoise.comp** which is distributed with this project. (The original distribution of the framework included a mostly intentionally empty placeholder version of **denoise.comp**. Replace it.)

References:

We will implement the (simple) first paper completely, and take one (simple) idea from the much more involved second paper.

1. Holger Dammertz, Daniel Sewtz, Johannes Hanika, Hendrik P.A. Lensch. Edge-Avoiding À-Trous Wavelet Transform for fast Global Illumination Filtering. 2010. Ulm University, Germany. https://jo.dreggn.org/home/2010_atrous.pdf

2. Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination 2017. https://cg.ivd.kit.edu/publications/2017/svgf/svgf_preprint.pdf

The last Vulkan initialization procedures in Vkapp::Vkapp

Uncomment the last three remaining initialization procedures, and follow any @@ comments:

```
createDenoiseBuffer();
createDenoiseDescriptorSet();
createDenoiseCompPipeline();
```

In **VkApp::drawFrame()**, uncomment the call to **denoise()**. You may want (or not) to make its execution conditional on an ImGUI boolean**.**

PushConstant for the denoise step

Replace the current contents of **struct PushConstantDenoise** (defined in **shaders/shared_structs.h**) with

- a single int value **stepwidth,** and
- several float parameters to fine-tune the À-Trous weight calculations. Initialized to these values as show. If you're in an experimental mood, hook them up to ImGUI sliders.
  - m_pcDenoise.normFactor = 0.003;
  - m_pcDenoise.depthFactor = 0.007;

```
struct PushConstantDenoise
{
   float normFactor, depthFactor;
   int  stepwidth;
};
```

The À-Trous denoise algorithm

The **denoise()** procedure makes **m_num_atrous_iterations** (=5) calls to the **denoise.comp** compute shader where one pass of the À-Trous algorithm will be implemented.

The features of these calls are:

for (int a=0; a<m_num_atrous_iterations; a++) // probably 5 iterations

- input image is **m_scImageBuffer**
- output image is **m_denoiseBuffer**
- **stepwidth** starts at 1 and increases by a factor of 2 each iteration
- a copy command follows the compute shader dispatch to copy **m_denoiseBuffer** back into **m_scImageBuffer** to setup for the next iteration

The À-Trous compute shader

The **denoise.comp** shader distributed with this project contains the outline of the algorithm and some @@ comments to lead you through the calculations.

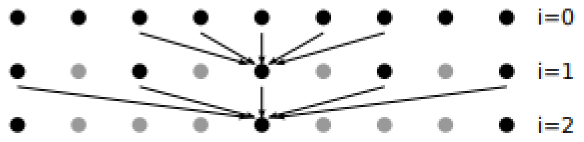


Figure 3: *Three levels of the À-Trous wavelet transform. Arrows indicate pixels that correspond to non-zero entries in the filter h_i and are used to compute the center pixel at the next level. Gray dots are positions that the full undecimated wavelet transform would take into account but that are skipped by the À-Trous algorithm.*

The algorithm: The shader is to implement one pass of the À-Trous algorithm, that being to calculate one output pixel as a selectively weighted average of 5x5 input pixels spread out (with holes) by a factor of stepwidth (= 1, 2, 4, 8, 16 in successive invocations).

All the capability of this algorithm is in a *very careful* choice of weights in the weighted average. Once again the weights are heuristic in nature and may not sum to one naturally.

$$O_P = \frac{\sum_{i=-2}^{2}\sum_{j=-2}^{2} w_{ij} I_{P+stepwidth*(i,j)}}{\sum_{i=-2}^{2}\sum_{j=-2}^{2} w_{ij}}$$

where $I_{...} = pDem$ the demodulated input pixel and the output O_P must be remodulated $O_P * \text{cKd}$

The calculation of $w_{i,j}$ makes use of two sets of values,

1. for the central pixel
   - at index P =ivec2(gl_GlobalInvocationID.xy)
   - cKd = read kdBuff at P, clamp value to vec3(0.1) or above
   - cVal = read .xyz of inImage at P; the pixel value to be denoised
   - cDem = cVal/cKd; **The pixel value demodulated**.
   - cNrm = read ndBuff .xyz at P
   - cDepth = read ndBuff .w at P
2. for the (i,j) pixel
   - at index P+offset; where offset=ivec2(i,j)*pc.stepwidth
   - pKd = read kdBuff at P+offset, clamp value to vec3(0.1) or above
   - pVal = read .xyz of inImage at P+offset; the pixel value to be denoised
   - pDem = pVal/pKd; **The pixel value demodulated.**
   - pNrm = read ndBuff .xyz at P+offset
   - pDepth = read ndBuff .w at P+offset

The weight $w_{i,j}$ is the product of these 4 weights:

1. h_weight: a Gaussian weight [1,4,6,4,2]/16 indexed by i+2
2. v_weight: a Gaussian weight [1,4,6,4,2]/16 indexed by j+2
3. d_weight: a depth related weight: $e^{-(\text{cDepth}-\text{pDepth})^2/\text{depthFactor}}$
4. n_weight: a normal related weight: $e^{-\|\text{cNrm}-\text{pNrm}\|^2/(\text{stepwidth}^2\ \text{normFactor})}$
5. The paper had a luminance based weight. Instead, we use de/re-modulation.